PS 390 DOCUMENT SET

TOOLS AND TECHNIQUES 1-10

EVANS & SUTHERLAND COMPUTER CORPORATION
P.O. Box 8700, 580 Arapeen Drive
Salt Lake City, Utah 84108

TOOLS AND TECHNIQUES

The *Tools and Techniques (TT1–10)* volume contains programming aids for the PS 390 user. It includes information such as application notes, helpful hints, how to use the various editors, using the GSRs, and using the ASCII-to-GSR converter.

TT1 Application Notes

This section contains a collection of applications for PS 390 users. Contributions to this section come from users inside and outside of Evans & Sutherland.

TT2 Helpful Hints

This section contains task-oriented information such as defining break keys, using the SITE.DAT file, and name suffixing. This section assumes a good working knowledge of the PS 390 and some programming experience.

TT3 Using the GSRs

This section is an introduction to using the graphics support routines (GSRs). The GSRs are a set of host-resident software routines that are the standard vehicle for communication to the PS 390 from the host. The GSRs can be used with the FORTRAN, Pascal and UNIX/C programming languages.

TT4 Function Network Editor

This section describes NETEDIT which permits the user to create a function network using a diagram on the PS 390 display rather than directly inputting commands to a file.

TT5 Function Network Debugger

This section describes NETPROBE which is used as a guide in preparing a user-written network debugging program.

TT6 Data Structure Editor

This section describes STRUCTEDIT which is a graphical display structure editor for the PS 390.

TT7 Character Font Editor

This section describes MAKEFONT which is a program that allows a user to edit an existing character font or create a new one.

TT8 ASCII to GSR Converter

This section describes the ASCII-to-GSR conversion program which allows a user to combine ASCII programming with the faster data communication speeds available through the GSRs.

TT9 Transformed Data and Writeback

This section provides information on how to retrieve transformed data such as a matrix or vector-list representation of transformation operations.

TT10 Crash Dump File

This section explains how to read back the crash dump file which is created when a system crash occurs.

TT1.

APPLICATION
NOTES

TT1. APPLICATION NOTES

CONTENTS

ILLUSTRATIONS

Section TT1

Application Notes

The PS 390 *Application Notes* is a collection of useful information and applications for PS 390 users. Contributions to the PS 390 *Application Notes* come from inside and outside of Evans & Sutherland. Each note includes the author's name and company. The notes are numbered arbitrarily for referencing only.

Users will develop ways of using the PS 390 that may be valuable for a wide range of applications. By publishing PS 390 *Application Notes*, Evans & Sutherland is acting as a clearinghouse to make your ideas and techniques generally available to other users. These notes have been written by PS 390 users and have not been rigorously tested. If you encounter errors or bugs in these *Application Notes* when you use them, please notify Evans & Sutherland.

These notes might describe an intricate function network that performs an important operation, show a new and useful way of structuring data, or they may provide something as simple as programming conventions or debugging methods that have helped you. In other words, almost any idea that you think may be useful for other PS 390 users is a candidate for the PS 390 *Application Notes*.

Please submit an Application Note for each idea you have to Evans & Sutherland. We will compile them and distribute them periodically to all PS 390 sites. We may, of course, not be able to publish every note submitted to us. Following is a description of the format we have used for this release of the *Application Notes*. Please follow the same format when you submit an Application Note. Submit your Application Notes and comments you may have to:

Neil Harrington
Evans & Sutherland
P.O. Box 8700
Salt Lake City, Utah 84108

Format Note

When actual code appears in an Application Note, the PS 390 commands are written with essential syntax in caps. Non-essential syntax is in lower case. For example, *BEGIN_Structure* indicates that it is necessary to enter only *BEGIN_S*.

PS 390 Application Note Form Instructions

(Title of Note Here)

Your Name
Department
Company
City
Date

Categories

List all possible categories for this note, for example: data structuring, function networks, command usage, host communications, animation, transformations, etc. Your note may fall into different categories.

Description

Briefly describe the function or application and tell why the new application is useful: what need it fills, what new thing it does, or what old thing it does in a new and easier way.

Implementation

Insert specific details about using the application. If the procedure is complex, describe it "top-down":

- First explain the "big picture" or supply a block diagram
- Then describe in detail each piece of the overall description or block in the diagram.

Do not start at the level of greatest detail unless the application is extremely simple.

Notes, Examples

Include an example of the application. For users, this could mean the difference between understanding and not understanding how to use your application when they can't find their way through the Description and Implementation.

Also put here any warnings or side notes you think might help someone to understand and use your application.

Application Note 1

Cursor Redefinition

Kerry Evans
Evans & Sutherland
Salt Lake City, Utah
December 1981

Categories

Screen cursor, data structuring

Description

This describes how the screen cursor may be redefined to be a symbol other than the default cursor, which is an "X".

Implementation

The default cursor is defined as a vector_list by a command of the form:

```
CURSOR := VECtor_list ITEMized n = 4
            P -.035, -.035
            L  .035,  .035
            P -.035,  .035
            L  .035, -.035;
```

To redefine the cursor as a square, simply redefine "CURSOR" in the following manner:

```
CURSOR := VECtor_list ITEMized n = 5
            -.035,  -.035
            -.035,   .035
             .035,   .035
             .035,  -.035
            -.035,  -.035;
```

Notes, Examples

The original cursor definition is lost until redefined in its original form by the user or until the PS 390 is powered on again. The INITialize command does not restore the default cursor definition.

Application Note 2

Defining a Dynamic Cursor

Kerry Evans
Evans & Sutherland
Salt Lake City, Utah
December 1981

Categories

Screen cursor, data structuring

Description

The screen cursor may be redefined to provide two different shapes - one when the data tablet pen tipswitch is up or open, and another when it is down or closed.

Implementation

Redefine the cursor as

```
CURSOR := BEGIN_Structure
UP_DOWN := SET conditional_BIT 1 OFF;
            IF conditional_BIT 1 is ON THEN C_SQUARE;
            IF conditional_BIT 1 is OFF THEN C_CROSS;
           END_Structure;

C_SQUARE := VECtor_list ITEMized n = 5
              -.035, -.035
              -.035,  .035
               .035,  .035
               .035, -.035
              -.035, -.035;

C_CROSS := VECtor_list ITEMized  n = 4
             P -.035, -.035
             L  .035,  .035
             P  .035, -.035
             L -.035,  .035;
```

Now connect the data tablet tipswitch Boolean value (output 2) to control which cursor symbol is displayed:

```
CONNect Tabletin<2>:<1>CURSOR.UP_DOWN;
```

Notes, Examples

The original cursor definition is lost until redefined in its original form by the user or until the PS 390 is powered on again. The INITialize command does not restore the default cursor definition.

Application Note 3

World-Space Rotations

Neil Harrington
Evans & Sutherland
Salt Lake City, Utah
February 1982

Categories

World-space rotations, object-space rotations, screen-space rotations

Description

A very desirable way of performing three-dimensional rotations is to know beforehand just what direction the object you are rotating is going to move. One way of doing this is to perform world-space (world-centered) rotations -that is, when you turn, say, the X rotation dial, you know the rotation will be about the world-space X axis. Likewise, of course, for Y and Z rotations. (See also object-space rotations.)

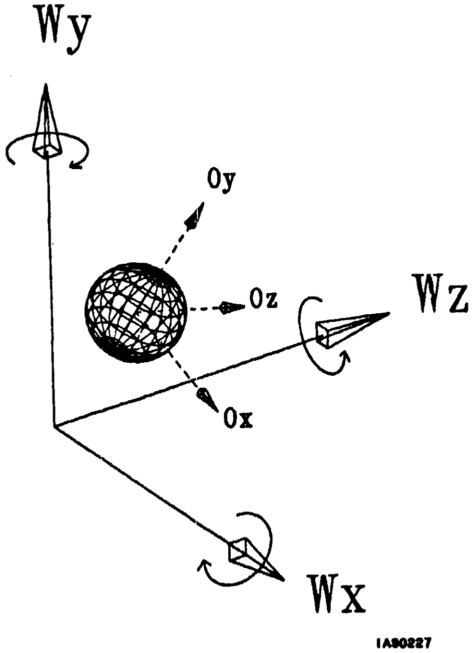
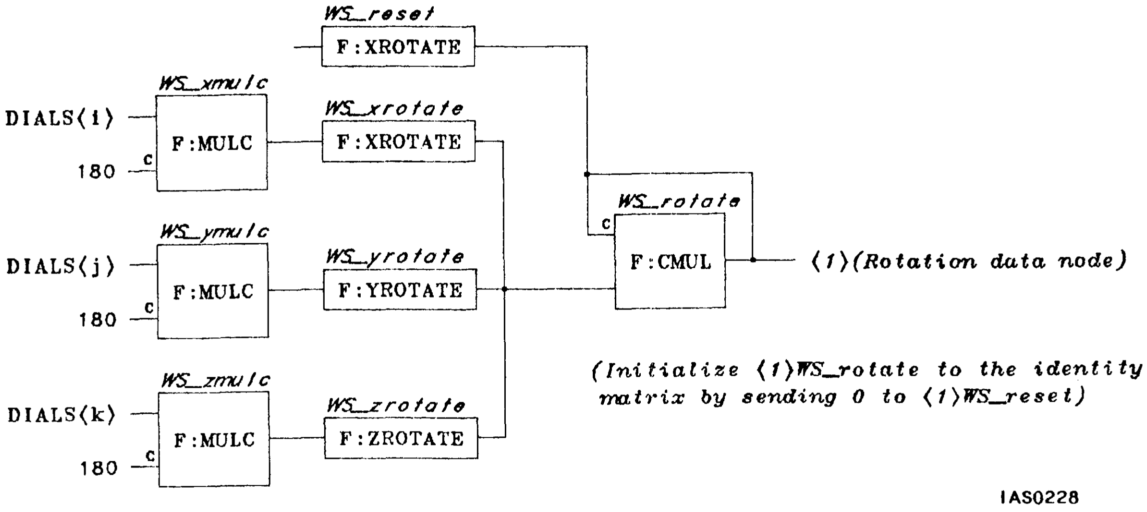


Figure 1-1. World-Space Rotations

Implementation

To get true world-space rotations, the rotations need to be processed in the order that they come in (that is, they need to be post-concatenated to the current matrix). The network to do this is shown in the diagram below.



(Initialize ⟨1⟩WS_rotate to the identity matrix by sending 0 to ⟨1⟩WS_reset)

Figure 1-2. World-Space Rotation Networks

The output of the last function should also be connected to the appropriate rotation node in the data structure.

Notes, Examples

If you are rotating an object about the world-space axes and viewing it from the negative Z axis, the space screen coordinates and world-space coordinates will coincide - the space-screen rotations will in effect be the same as the world-space rotations.

There should only be one rotation node in the data structure (not one each for X, Y, and Z rotations). This node can be created with the ROTate X, ROTate Y, or ROTate Z commands.

To reset the network and rotation node in the data structure, just put an identity matrix on input <1> of WS_Rotation. This can be done by connecting an instance of F:XROTATE to it and sending a 0 to input <1> of WS_Reset.

Application Note 4

Object-Space Rotations

Neil Harrington
Evans & Sutherland
Salt Lake City, Utah
February 1982

Categories

Object-space rotations, world-space rotations, data-space rotations

Description

A very desirable way of performing three-dimensional rotations is to know beforehand just what direction the object you are rotating is going to move. One way of doing this is to perform object-space (object-centered) rotations; that is, when you turn, say, the X rotation dial, you know the rotation will be about the X axis of the original object definition space. Likewise, of course, for Y and Z rotations. (See also world-space rotations).

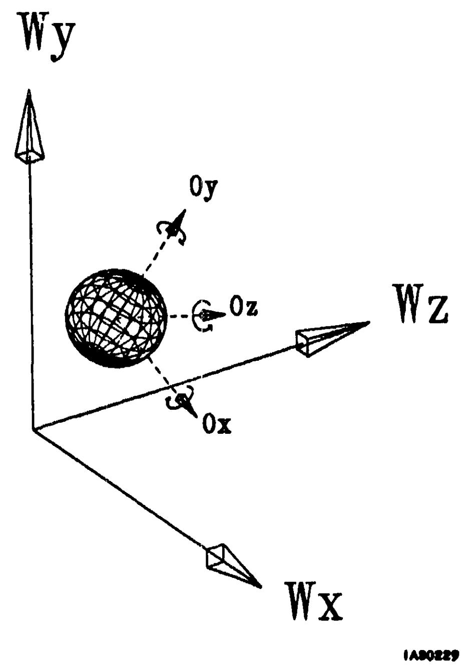


Figure 1-3. Object-Space Rotations

Implementation

To get true object space rotations, the rotations need to be processed in the reverse order that they come in (that is, they need to be pre-concatenated to the current matrix). The network to do this is shown in the diagram below.

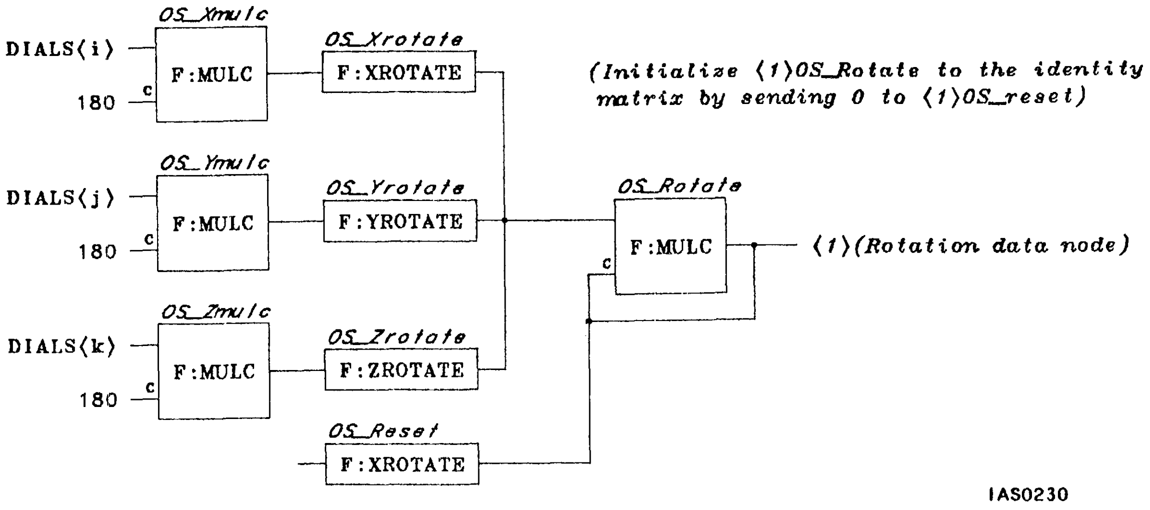


Figure 1-4. Object-Space Rotation Network

The output of the last function should also be connected to a rotation node in the data structure.

Notes, Examples

There should only be one rotation node in the data structure (not one each for X, Y, and Z rotations). This node can be created with the ROTate X, ROTate Y, or ROTate Z commands.

To reset the network and rotation node in the data structure, put an identity matrix on input <1> of OS_Rotation. This can be done by connecting an instance of F:XROTATE to it and sending a 0 to input <1> of OS_Reset.

Application Note 5

Rational Polynomial Command Usage

Marty Best, Bill Armstrong
Evans & Sutherland
Salt Lake City, Utah
April 1982

Categories

Circles, ellipses, curve generation, RATIONAL POLYNOMIAL

Description

The Polynomial commands that are available on the PS 390 offer a powerful means of building curve shapes without transmitting large numbers of vectors. Unfortunately, use of the Polynomial commands requires an understanding of curve generation and a routine for computing the curve parameters to be sent to the PS 390. Only users experienced in curve generation, for the most part, will find a specific use for them.

Some basic curve shapes, however, can be adapted to many applications and are simple to implement.

The command detailed below can be modified to draw a circle of a given radius, or an ellipse of a specified size. Of course, these primitives can be instanced by any other structure and translated, rotated, or scaled.

Implementation

A circle must be defined in two parts using a RATIONAL POLYNOMIAL command. It can then be included in a BEGIN_Structure...END_Structure and referenced as a single entity. The syntax is as follows:

```
CIRCLE := BEGIN_Structure
            RATional POLYnomial
             2r,   0,  0,  2
            -2r, -2r,  0, -2
              0,   r,  0,  1
            CHORDS=25;
            RATional POLYnomial
             2r,   0,  0, -2
            -2r, -2r,  0,  2
              0,   r,  0, -1
            CHORDS=25;
         END_Structure;
```

where r is the desired radius of the circle. The number of chords have been set at 25 to give a smooth appearance.

Notes, Examples

The two RATIONAL POLYNOMIAL commands given above define the right and left semicircles of the circle and can be made the top and bottom semicircles by exchanging the X and Y columns (Columns 1 and 2).

The above circle can be modified to give an ellipse as follows:

```
ELLIPSE := BEGIN_Structure
             RATional POLYnomial
              2a,  0,  0,  2
             -2a,-2b,  0, -2
               0,  b,  0,  1
             CHORDS=25;
             RATional POLYnomial
              2a,  0,  0, -2
             -2a,-2b,  0,  2
               0,  b,  0, -1
             CHORDS=25;
          END_Structure;
```

where a and b are the major and minor axes of the ellipse. Again the number of chords has been chosen for smoothness.

Application Note 6

Proportional Scaling

Neil Harrington
Evans & Sutherland
Salt Lake City, Utah
September, 1982

Categories

Scaling

Description

A dial is usually used to accumulate the scale factor in standard scaling networks. It's hard to control scaling this way, though, since the current scale factor becomes very small or very large in proportion to the new dial value. For example:

Current Scale Factor	New Dial Value	New Scale Factor	% Increase
0.01	.1	0.11	1000.0
100.00	.1	100.10	0.1

When the current scale factor is small, the effect of a turn of the dial is large, and vice versa.

The network shown below will correct this problem by making the effect of the dial proportional to the current scaling factor. Using this network the chart shown above will look like:

Current Scale Factor	New Dial Value	New Scale Factor	% Increase
0.01	.1	0.011	10
100.00	.1	110.00	10

Implementation

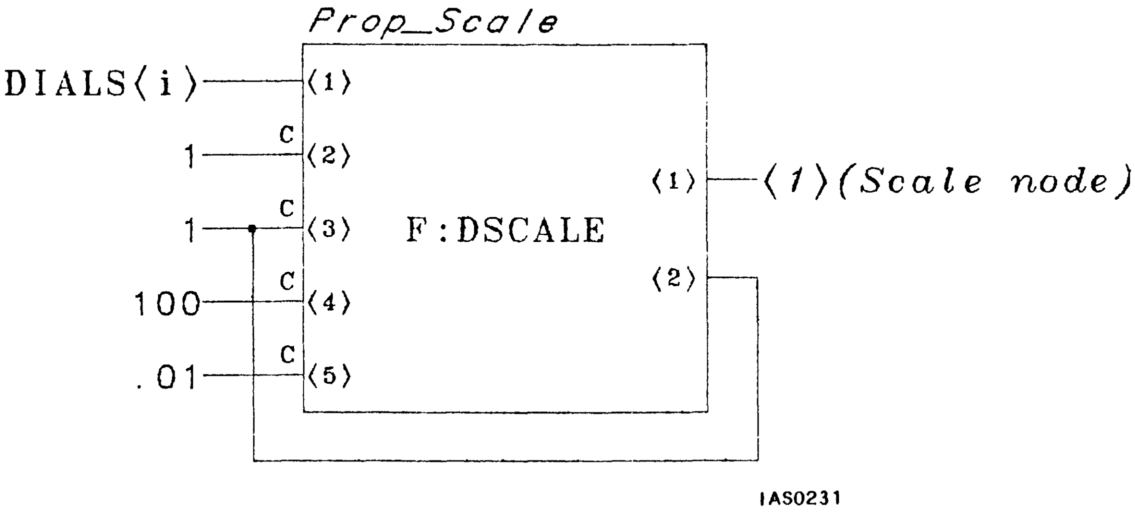


Figure 1-5. Proportional Scaling Network

Application Note 7

Local Inking of Tablet Coordinates

Neil Harrington
Evans & Sutherland
Salt Lake City, Utah
June 1982

Categories

Inking, F:XOR

Description

Inking is the technique of using an input device (usually the data tablet) to sketch "freehand." This application note describes a function network that will allow the user to do inking with the data tablet.

Implementation

The network is as follows. Data structure A should be DISPlayed and be created with a command such as:

```
A := VEC n=1000 0,0;
```

(The n=1000, or some other number, allocates a block of memory for the vector list).

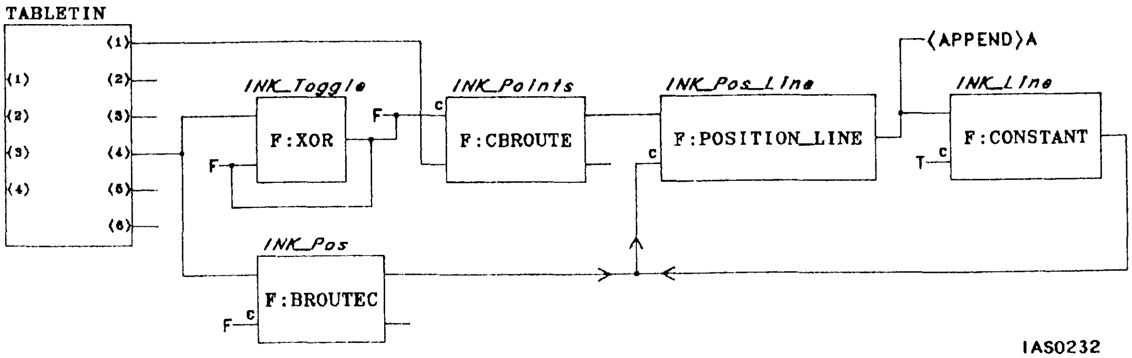


Figure 1-6. Inking Network

Notes

To use this, press and release the data tablet pen to start inking and then press and release it again to stop inking. Do this as many times as needed.

Application Note 8

Local Rubber Banding of Tablet Coordinates

Neil Harrington
Evans & Sutherland
Salt Lake City, Utah
June 1982

Categories

Rubber banding, grid banding

Description

This note describes a function network to do rubber banding using the data tablet.

Rubber banding is the technique of displaying a line segment that extends from some fixed point to the data tablet cursor and moves along with the cursor until some indication is given (such as pressing the data tablet pen switch) to fix the line segment at the current position. This way you can see the lie of the line before you finish positioning it.

Implementation

The network is as follows. Data structure A should be DISPlayed and be created with a command such as:

```
A := VEC n=1000 0,0;
```

(The n=1000, or some other number, allocates a block of memory for the vector list).

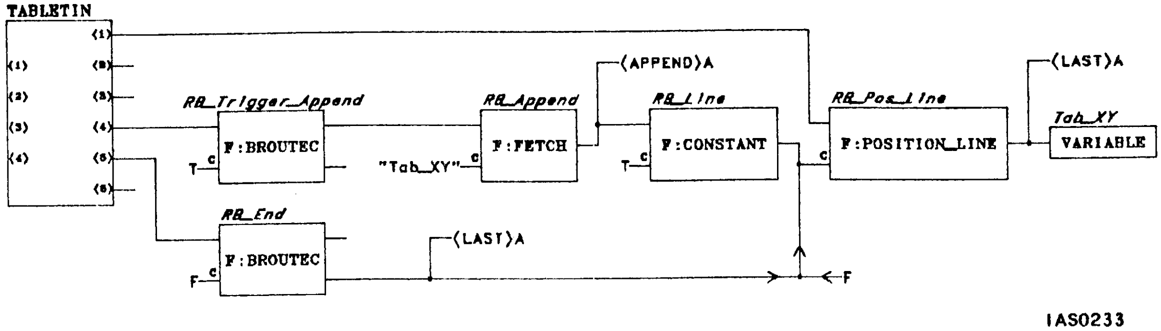


Figure 1-7. Rubber-Banding Network

Notes

To use this network, press and release the stylus on the data tablet to fix the first position. Moving the stylus around on the tablet now will create a rubber band line from the initial position to the cursor.

Pressing and releasing the stylus again will fix this line segment, and a new rubber band line will start from this last point to the next point you press down on and so on. To break this continuous line and start a new series of rubber band segments, you must move the stylus away from the tablet surface. This will cause the current rubber band line to disappear; a new one will start as soon as a new starting position is selected.

Application Note 9

Local Grid Banding of Tablet Coordinates

Kerry Evans
Evans & Sutherland
Salt Lake City, Utah
April 1982

Categories

Grid banding, rubber banding, function networks, data tablet

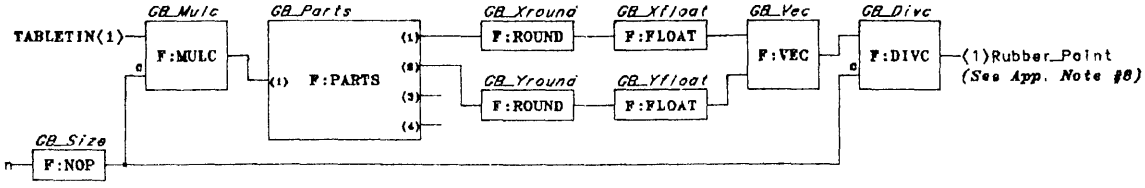
Description

This note describes a function network which takes 2D coordinates from the data tablet and constrains the points to fall on grid points of a user-defined grid - that is, it performs rubber banding to discrete points on a grid. We call this grid banding (see also Application Note 8.)

Implementation

Use the same network as that for rubber banding (Application Note 8), but instead of connecting the tablet xy position (TABLETIN <1>) to the POSITION_LINE Function directly, connect the output of the DIVC Function in the network shown below to the POSITION_LINE Function.

Specify the number of grid points per unit by sending a real to input <1> of the NOP function. For example, sending 10 causes the vectors output from the DIVC function to lie on grid points 0.1 unit apart in X and Y.



IAS0234

Figure 1-8. Grid-Banding Network

Notes, Examples

Use just like rubber banding (Application Note 8). This is an easy way of doing rubber banding without having to be as accurate with pen positioning, especially if you are doings things like schematics or block diagrams.

Application Note 10

Translation Network

Kerry Evans
Evans & Sutherland
Salt Lake City, Utah
April 1982

Categories

Translation, F:ACCUMULATE

Description

This application note shows an example of how the ACCUMULATE function may be used to build translation vectors from the dials. Since ACCUMULATE can accept real numbers or vectors, it is a simple matter for it to accumulate "position."

Implementation

The following function network allows Dials 1, 2, and 3 to control the X, Y, and Z components of the Translate vector, respectively.

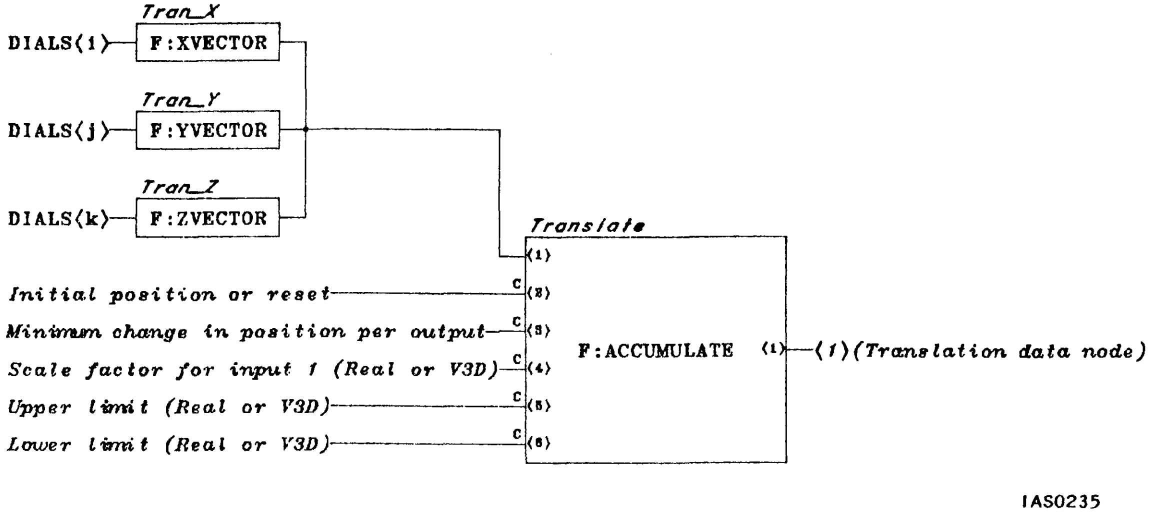

Tran_X
DIALS<i>
F:XVECTOR
Tran_Y
DIALS<j>
F:YVECTOR
Tran_Z
DIALS<k>
F:ZVECTOR
Translate
<1>
Initial position or reset
C <2>
Minimum change in position per output
C <3>
Scale factor for input 1 (Real or V3D)
C <4>
Upper limit (Real or V3D)
C <5>
Lower limit (Real or V3D)
C <6>
F:ACCUMULATE
<1>
<1>(Translation data node)
IAS0235

Figure 1-9. Translation Network

Notes, Examples

The X, Y, and Z vector functions build 3D vectors from the dial values, which get scaled by input <4> of ACCUMULATE and accumulated on input <2>. ACCUMULATE may be reset by sending the initial translation value to input <2>. But no output will be generated until input is received on <1>. This may be the result of turning a dial or sending a Boolean value to <1>.

Scale factor and upper and lower limit may be real numbers (if uniform scaling and limiting in X, Y and Z is desired) or they may be vectors, in which case the components are applied individually in each dimension. Input <3> specifies the amount by which the accumulated sum must change before an output is generated. This amount is a real number greater than or equal to 0.

Application Note 11

Animation Sequencing with CLOCK Function

Gary Cannon, Neil Harrington
Evans & Sutherland
Salt Lake City, Utah
May 1982

Categories

Animation, F:CLCSECONDS

Description

This shows a method for using a series of clock functions (F:CLCSECONDS, F:CLFRAMES, and/or F:CLTICKS) to run through a sequence of actions.

Implementation

A clock can control some motion for a given time span, then stop and trigger the start of the next clock in sequence, which controls some other motion.

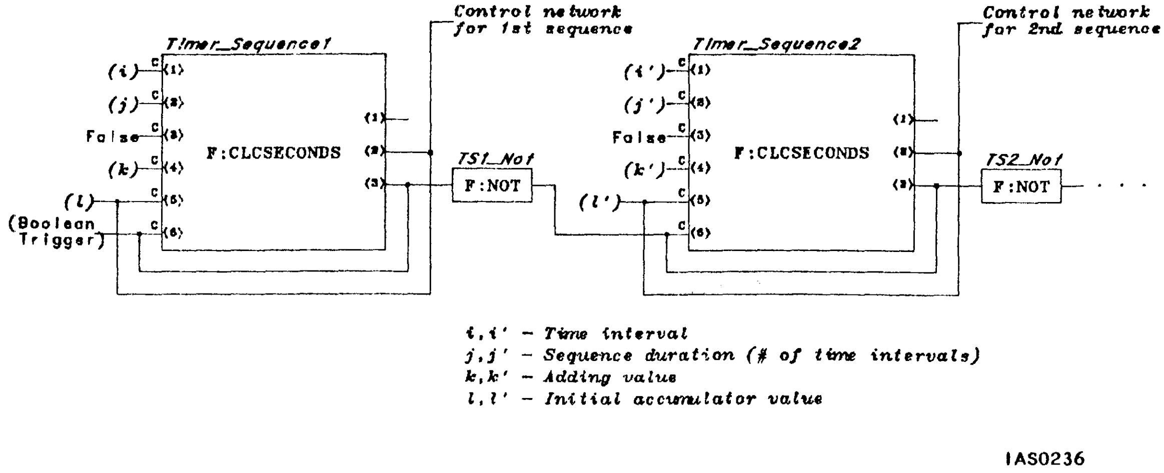


Figure 1-10. Animation Sequencing On-The-Fly

Notes, Examples

The actions best suited for this type of animation sequence are those that can use the summing outputs <1> or <2> to modify the currently displayed data structure. An example of this would be using output <2> to feed a rotation network that then modifies a rotation node in the displayed data structure. When output <3> of the clock generates a FALSE, a network could also be triggered to change the level of detail and change the data structure being viewed.

To cycle repetitively through the sequence, input <2> of each of the timers needs to be reset to the initial value. This could be done by having the NOT function of the last sequence trigger the following network (note that this network will also trigger the sequence to start over again "n" times).

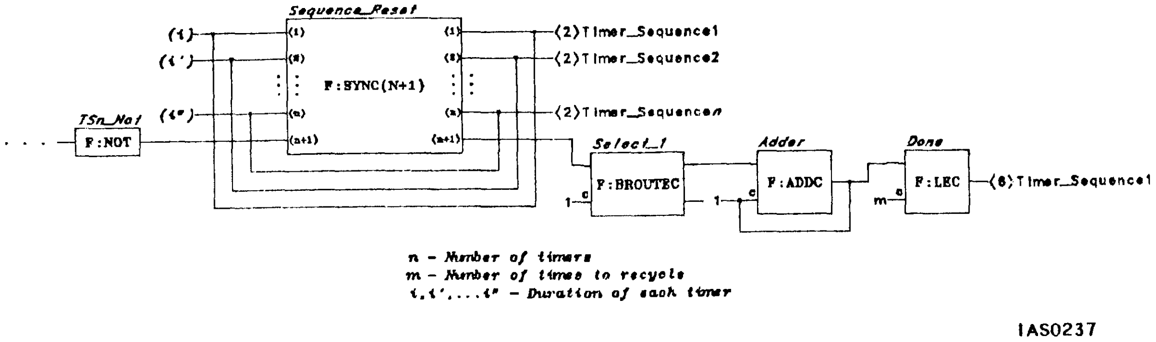


Figure 1-11. Animation Replay and Reset Network

Application Note 12

Frame-by-Frame Animation

Neil Harrington
Evans & Sutherland
Salt Lake City, Utah
August, 1982

Categories

Animation, F:CLCSECONDS, F:MODC, Level-of-Detail

Description

This shows a method for using a clock function (F:CLCSECONDS, F:CLFRAMES, or F:CLTICKS) to cycle through a series of previously calculated frames. Typically, each frame would consist of different transformations applied to the same objects. The modulo function allows for the animation to recycle indefinitely.

Implementation

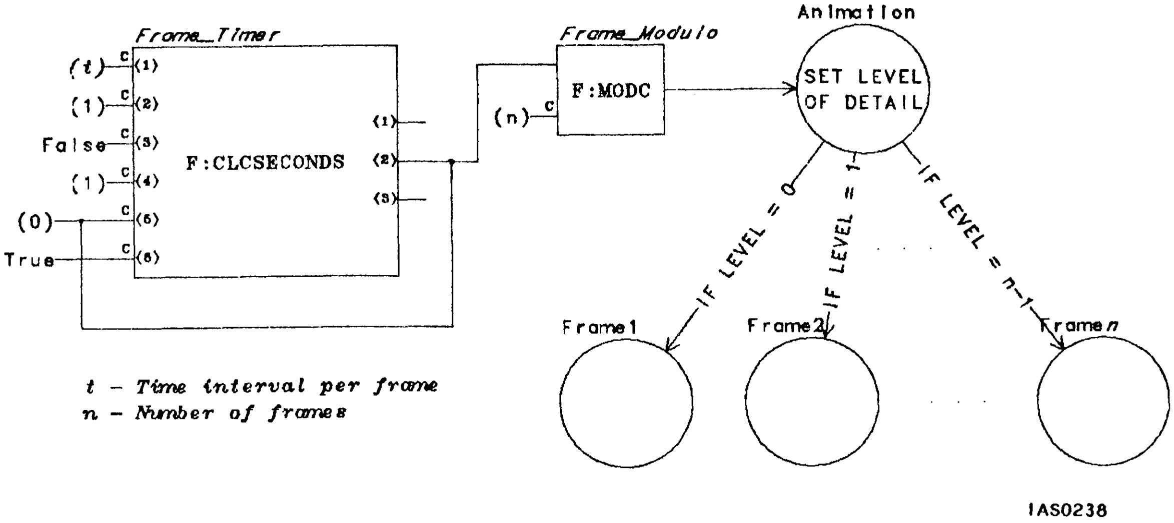


Figure 1-12. Frame-by-Frame Animation

Notes

1. Input <1> of Frame_Timer could be dynamically altered to change the speed of the animation sequence.
2. Input <4> of Frame_Timer could be dynamically altered to skip frames in the animation sequence.
3. The clock could be stopped and a value sent to input <1> of Frame_Modulo to look at a particular frame in the sequence.

Application Note 13

Menu Selection

Gary Cannon
Evans & Sutherland
Salt Lake City, Utah
August, 1982

Categories

Menus

Description

This function network allows you to do menu picking from a defined menu in a specific area of the screen. It uses simple math to produce a "box number" from the tablet X and Y coordinates.

The menu boundaries are shown below as they would appear in a full screen viewport on the screen:

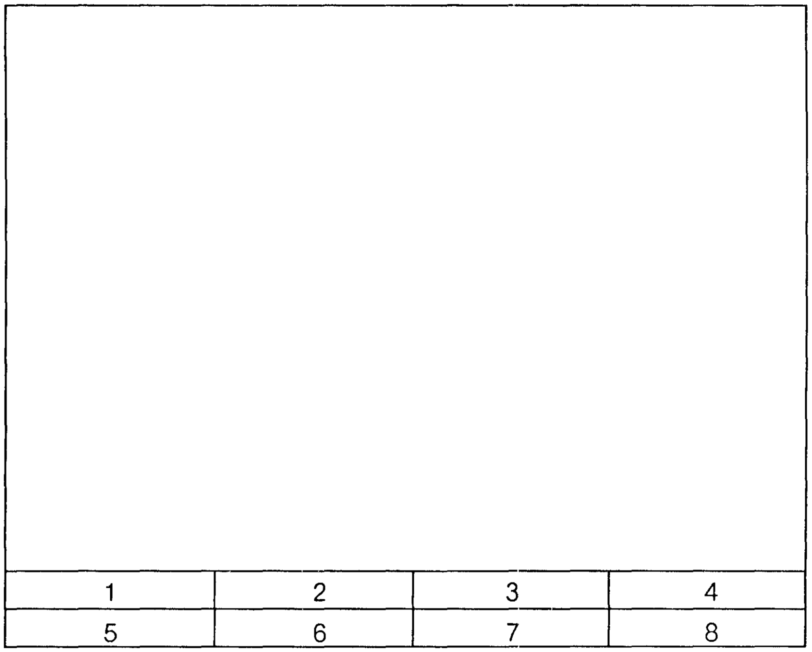
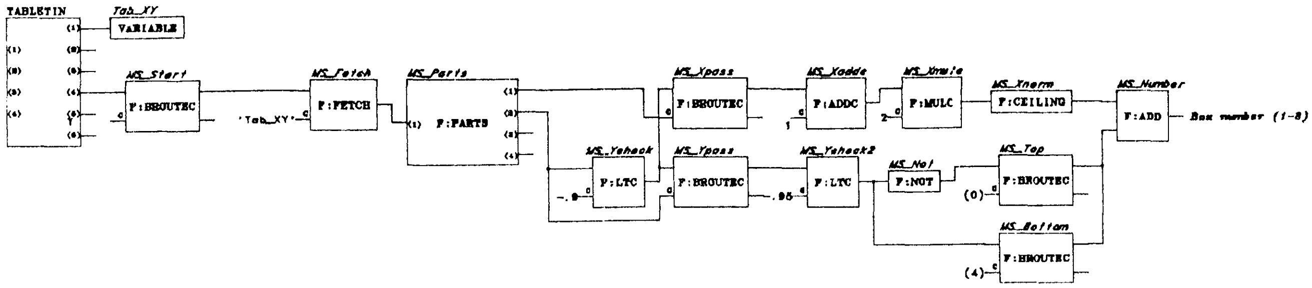


Figure 1-13. Menu

Each of the numbers shown is a value produced by the network when the screen cursor is in the menu box with that number and the stylus is pressed down on the tablet. Of course, these numbers should be replaced by descriptive names for the real menu.

Implementation

The menu selection network is shown below:



IAS0239

Figure 1-14. Menu Selection Network

Notes

With modifications to this network, menus of other sizes and shapes can easily be created.

Application Note 14

Rotary Switch

Carl Ellison
Evans & Sutherland
Salt Lake City, Utah
August 1982

Categories

Switching, Multiplexing, F:SYNC(n), F:ROUTE(n)

Description

The simplest form of multiplexing breaks a stream of items into a collection of streams by sending the first item to destination 1, the second to destination 2..., the nth to destination n, the (n+1) back to destination 1, and so on. As long as n<=20 the function network shown below can do that job.

Implementation

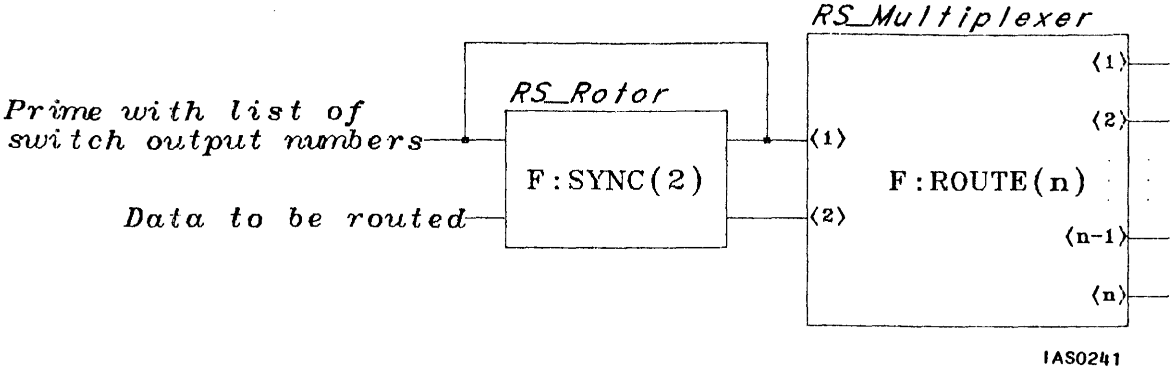


Figure 1-15. Rotary Switch Network

Application Note 15

Shift Register

Carl Ellison
Evans & Sutherland
Salt Lake City, Utah
August, 1982

Categories

Shift register, F:SYNC(n)

Description

F:SYNC(n) can be used to act as a shift register. It can be used, for example, to achieve scrolling by feeding character strings to CHARACTER display nodes.

Implementation

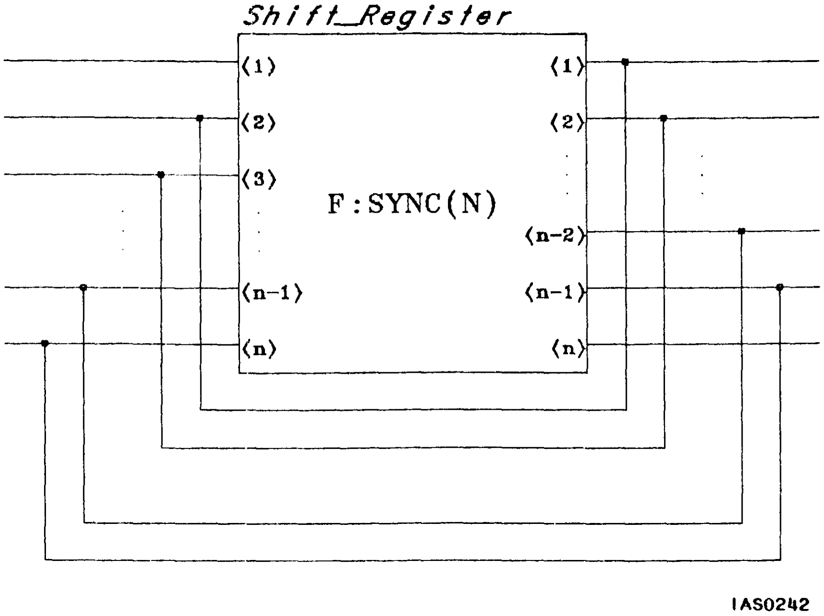


Figure 1-16. Shift Register Network

Notes

These F:SYNC functions could be cascaded to shift or scroll more than 20 lines if needed.

Application Note 16

Function Network Sequencing

Carl Ellison
Evans & Sutherland
Salt Lake City, Utah
August, 1982

Categories

Function network sequencing, F:SYNC(n), function loops, synchronization

Description

This application note describes how to control when a function network, or series of networks, runs. The sequencing schemes described here are based on the use of a "GO" token which is passed around among F:SYNC(n) functions, controlling the activation of sub-networks.

This type of network implementation can be safer, also, since it will not allow new tokens into a network until it has completely processed the current tokens.

Implementation

The network shown below has its inputs and outputs controlled by F:SYNC(n) functions. This allows the network to "execute" completely before accepting new inputs:

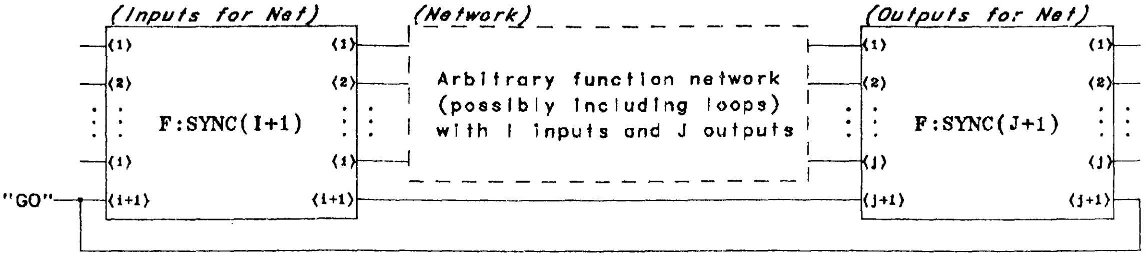


Figure 1-17. Function Network Sequencing

The network shown below shows an arbitrary number of sub-networks linked together. This forces the sequential processing of these networks.

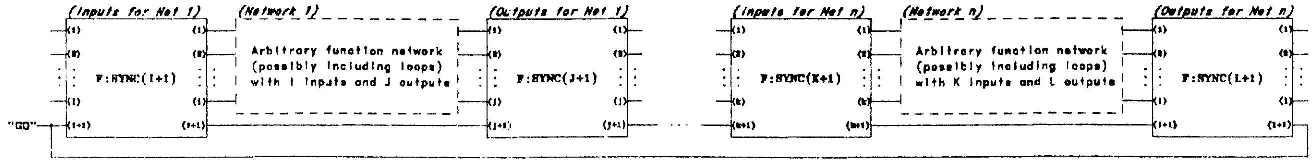


Figure 1-18. Sequential Executing Networks

Application Note 17

IF-THEN-ELSE Construct

Carl Ellison
Evans & Sutherland
Salt Lake City, Utah
August, 1982

Categories

IF-THEN-ELSE, Boolean switch, F:BROUTE

Description

This application note shows how to implement an IF-THEN-ELSE construct using PS 390 functions. It assumes "i" values are input to test some Boolean relation. The values are then routed to one of two networks depending on the Boolean value that is output from the expression. This implementation is similar to the general programming statement:

```
IF <expression> THEN statement1
     ELSE statement2;
```

Implementation

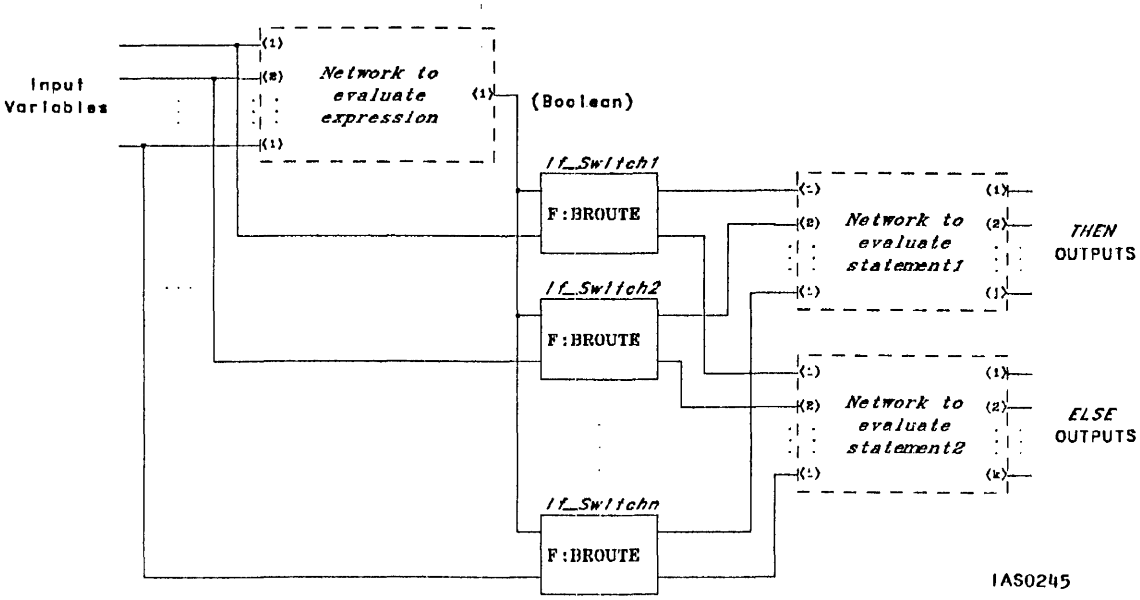


Figure 1-19. IF-THEN-ELSE Network

Note

This application shows how F:BROUTE can be used as a Boolean switch function.

Application Note 18

A Realtime Analog and Digital Clock

Patrick Fitzhorn, David Ferguson
Center for Computer-Assisted Engineering
Colorado State University
Ft. Collins, Colorado
October, 1982

Categories

F:CLTICKS

Description

Frequently it is useful to display a realtime clock on the PS 390 screen. The network described here has, as an end result, both an analog clock component (rotations in degrees for the hour, minute, and second hands) and a digital component.

Implementation

The network is based on F:CLTICKS with constant input of 120 on input <1>. This generates an integer at output <2> once per second, which is incremented by one each tick. The clock is based on a 12-hour cycle, so F:MODC resets the clock after 43,200 seconds.

An initialization network is provided that changes standard hour, minute, and seconds input into seconds. This value is then sent to input <5> of F:CLTICKS, which serves as a new starting value for the clock. The network diagram is shown in the following figure.

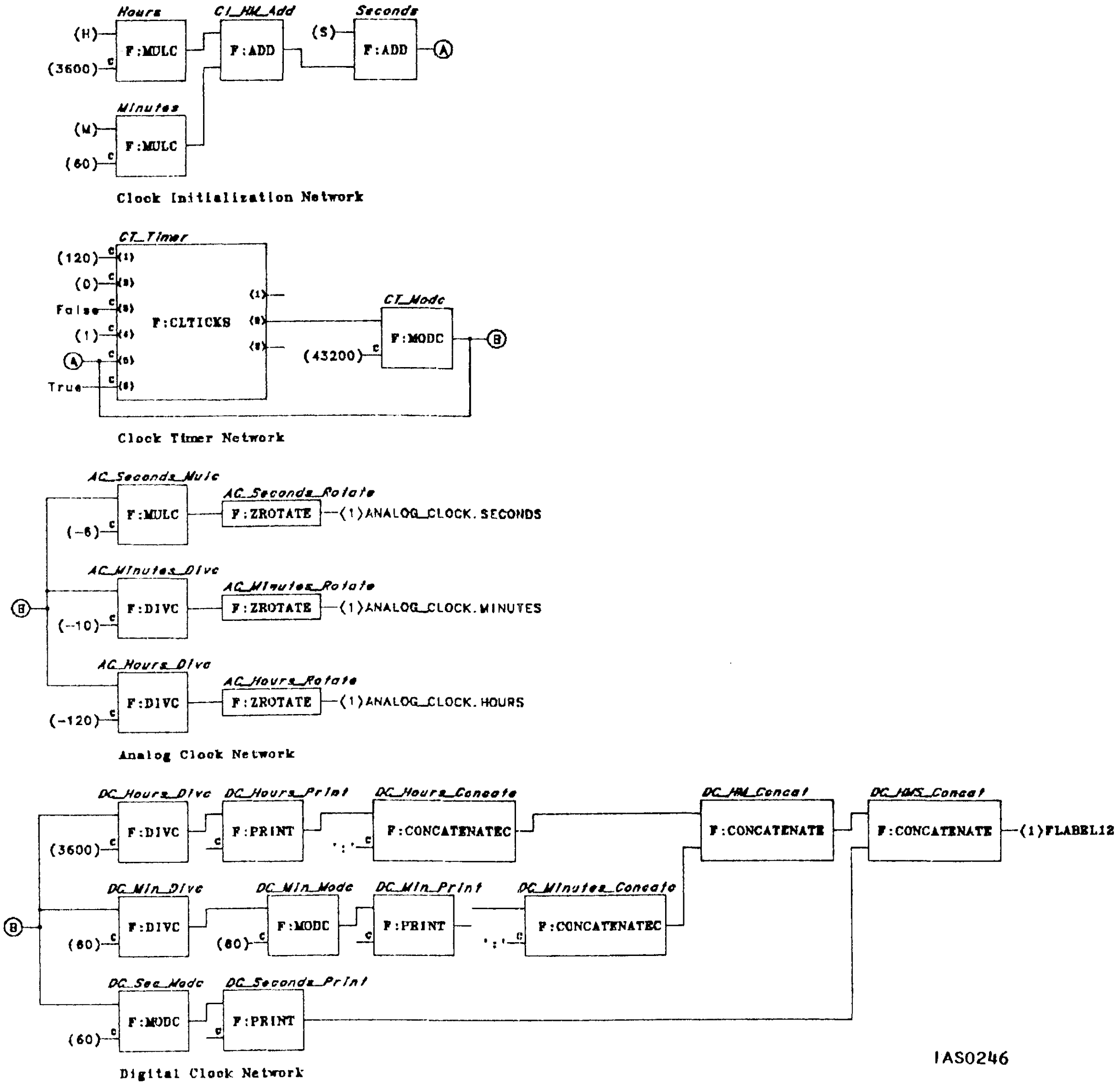


Figure 1-20. Clock Network

An example of the data structure for the analog clock face is:

```
ANALOG_CLOCK := BEGIN_Structure
                VIEWport HORizontal=-.1:.5454 VERtical=-1:1;
SECONDS := ROTate 0 := THEN SECOND_HAND;
MINUTES := ROTate 0 THEN MINUTE_HAND;
HOURS := ROTate 0 THEN HOUR_HAND;
                END_Structure;

SECOND_HAND := SCALE .025,1 THEN BASIC_HAND;
MINUTE_HAND := SCALE .05,.8 THEN BASIC_HAND;
HOUR_HAND := SCALE .075,.5 THEN BASIC_HAND;
BASIC_HAND := VECtor_list N=5  0,0  1,.25  0,1  -1,.25  0,0;
```

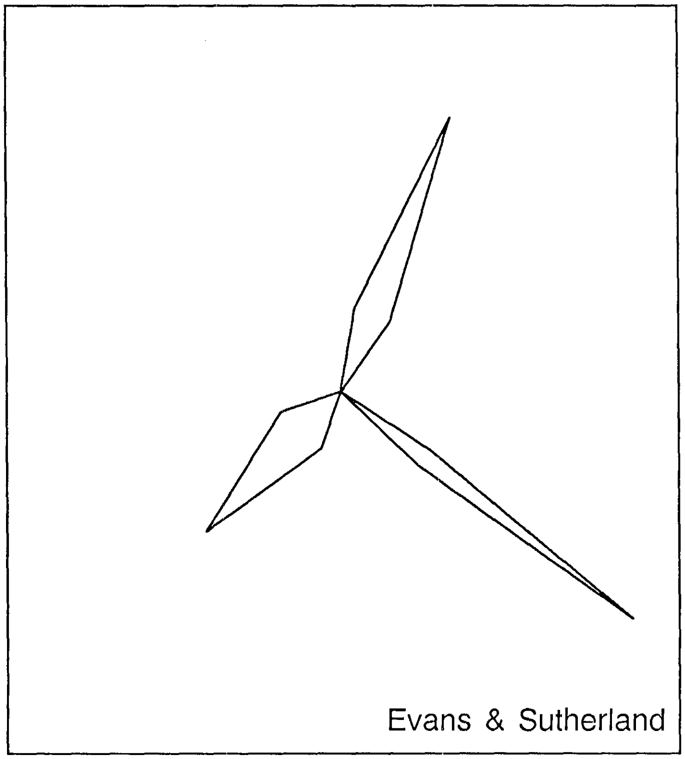


Figure 1-21. Analog Clock

Notes

The digital clock's display is of the form: (hours):(minutes):(seconds) with a maximum of eight digits.

The output can be connected to a character node in a display data structure or to a function key LED label, if so desired. In the current digital component, leading zeros for minutes and seconds do not appear, so that 9:05:05 is displayed as 9:5:5. This has not proved to be much of a hardship. If a standard 8-digit output is required, one could test the minute and second outputs and, if less than 10, concatenate a leading zero. The clock starts out at time 00:00:00. To set the clock, the following commands are used:

```
store fix(h) to <1>hours
store fix(m) to <1>minutes
store fix(s) to <1>seconds
```

where,

```
h    = integer between 1 and 12
m, s = integer between 1 and 60
```

Application Note 19

Blinking Using SET RATE EXTERNAL

Gary Cannon
Evans & Sutherland
Salt Lake City, Utah
January, 1985

Categories

Data Structuring, SET RATE EXTERNAL

Description

This data structure and function network allow blinking when appropriate without requiring two separate structure paths. This will use the SET RATE EXTERNAL command.

Implementation

The network is as follows:

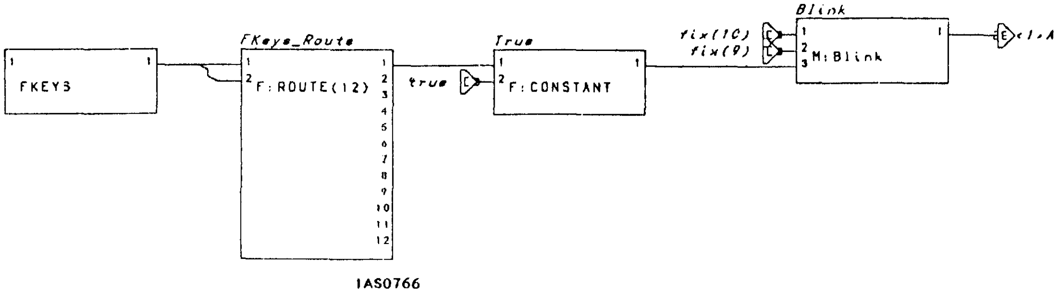


Figure 1-22. Blinking Network

The data structure should be displayed and be created with commands such as:

```
A := Set Rate External then B;
B := If Phase ON then OBJECT;
```

M:BLINK is a macro which is expanded in the following figure.

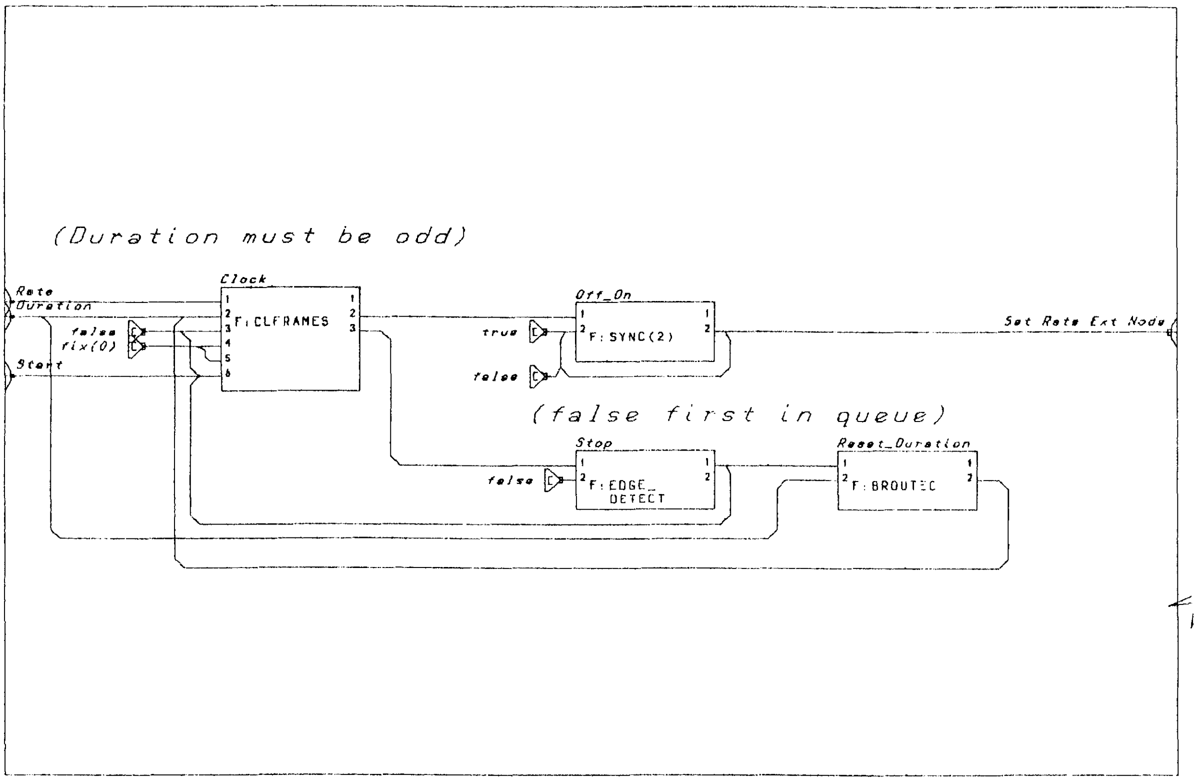


Figure 1-23. M:BLINK Macro

Application Note 20

Local Inking of Tablet Coordinates

Dan Harlin
Evans & Sutherland
Salt Lake City, Utah
December, 1982

Categories

Inking, F:EDGE_DETECT

Description

This application note is a variation of Application Note 7. To do inking with the network of Application Note 7, the user began by pressing and releasing the pen, and ended by pressing and releasing the pen again. With this network, the user presses the pen to begin, and releases the pen to end inking.

Implementation

The network is as follows:

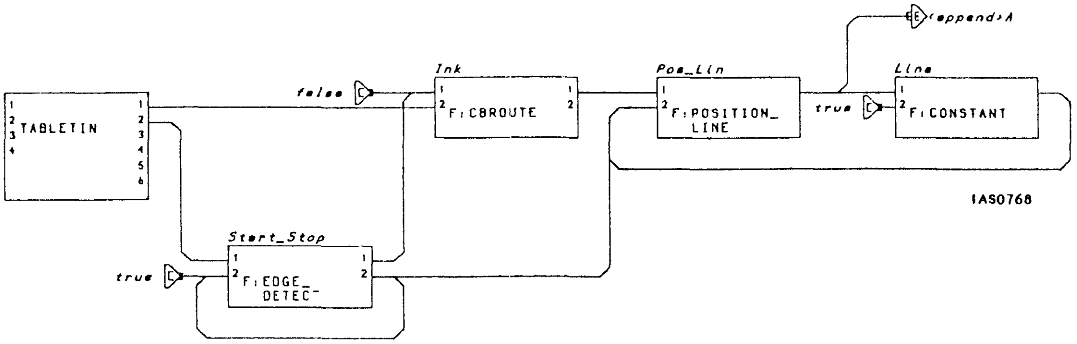


Figure 1-24. Inking Network

The data structure should be displayed and be created with a command such as:

```
A := VECTOR_LIST N=1000 0,0;
```

Notes, Examples

To ink, press the data tablet pen. Continue to press the pen while inking. To stop inking, release the pen.

Application Note 21

Integer Input Via Numeric Keypad

Michael F. Werner
Performance Analysis
Aerospace Corporation
El Segundo, CA
May 4, 1984

Categories

Function Keys, F:CHARCONVERT

Description

The most straightforward use of the PS 390 keyboard as an input device involves the use of the 12 function keys in conjunction with the FKEYS initial function instance. The integers 1 through 36 can be generated with the function keys and the SHIFT or CTRL keys on the keyboard.

Many PS 390 applications require the use of a large number of function keys. Use of the 12 keys becomes cumbersome in these applications. The function network below allows the numeric keypad on the keyboard to be used as an input device in a manner similar to the function keys.

Implementation

The network diagrammed converts a two-key sequence on the numeric keypad into an integer between 0 and 99 inclusive.

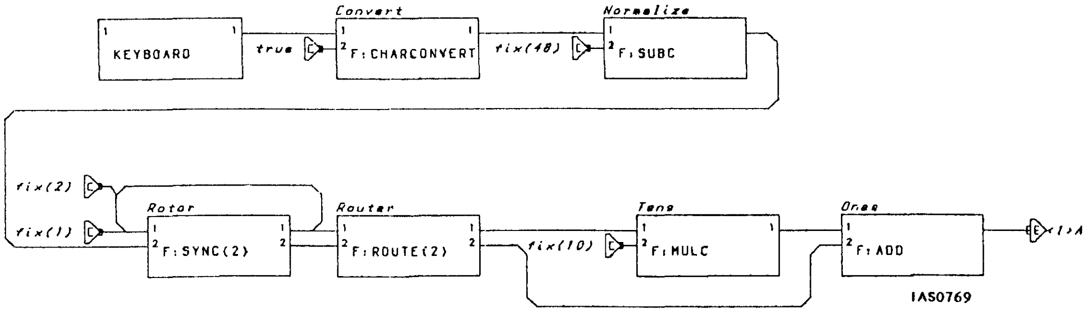


Figure 1-25. Keypad Network

Notes, Examples

This network can be used in applications where a large number of objects or perhaps 50 overlays of a single object must be toggled off and on. A list of 2 digit codes corresponding to each overlay must be provided until the codes are memorized. This approach is easier to use and requires less code than re-programming the function keys or developing a series of menus.

The user is cautioned that as long as the keyboard is enabled as an input device, this network will attempt to convert any 2 character ASCII sequence into an integer.

Note that this network is easily expandable to handle 3 digit sequences.

Application Note 22

Matrix Transpose

Thomas Hern
Department of Computer Science
University of North Carolina
New West Hall 035–A
Chapel Hill, NC 27514

and

Department of Mathematics & Statistics
Bowling Green State University
Bowling Green, OH 27514
January, 1985

Categories

Transformations, matrices

Description

This function network produces the transpose of the 3x3 matrix, which is the input to the network. Hence, the matrix which is the output of the network (F:MATRIX3, specifically) has as its rows the columns of the input. This network operates very quickly in this form.

Implementation

The network is as follows:

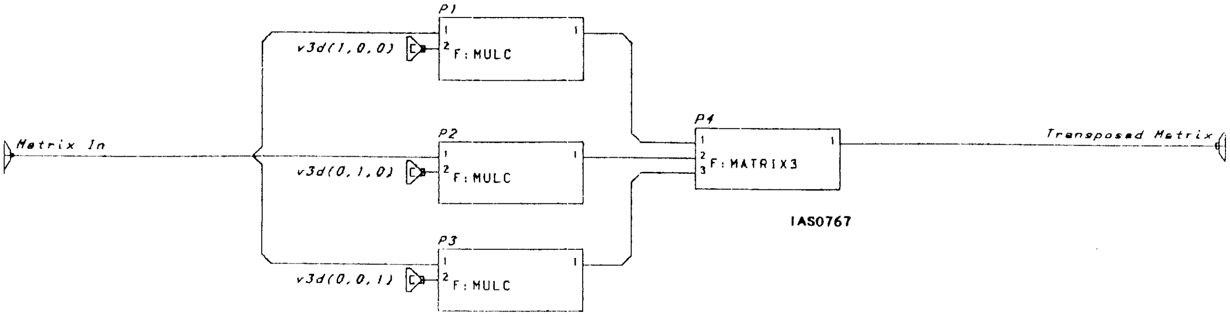


Figure 1-26. Matrix Transpose Network

Notes, Examples

The inverse of an orthogonal matrix is its transpose, so this network can often make calculating an inverse unnecessary. Such an inverse may be needed in some rotation nodes.

This implementation is for 3x3 matrices only, but adding (or deleting) COLn functions and changing the SEND statements will adapt the network to any size.

Application Note 23

Laser Disk Controller

Mike Grannan
Evans & Sutherland
Saint Louis, MO
January 10, 1985

Categories

Animation, PS 390 raster, laser disk

Description

A function network for automatic generation of separate PS 390 frames of an animation sequence for storage onto laser disk is documented. A Panasonic TQ-2022FC laser disk is attached to the PS 390. Recording commands are sent from the PS 390 to the laser disk via RS-232 communications; the picture is transmitted from the PS 390 to a color encoder, which converts an RGB signal to the NTSC format required by the laser disk.

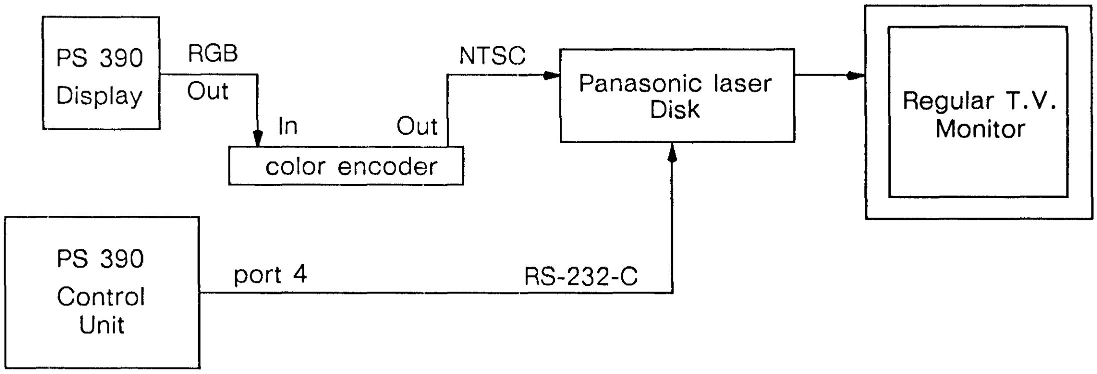


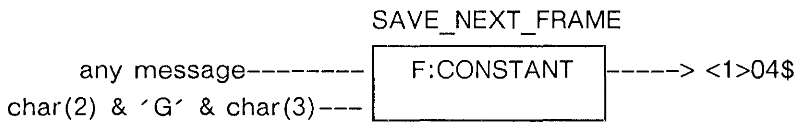
Figure 1-27. Hardware Diagram for PS 390/Laser Disk Configuration

The following function network can be used to enhance any PS 390 shading network to store multiple renderings onto laser disk for playback later. It is intended for purposes where high-quality shaded animation sequences are desired. Since the PS 390 uses a static raster display, each frame of the animation sequence is precalculated, and the laser disk is given the command to store each frame as it is generated.

Implementation

1. Laser disk communications

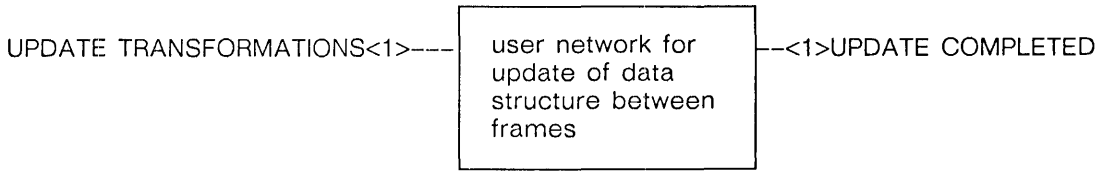
   The particular laser disk used is connected to port 4 of the PS 390 control unit, and requires configuring port 4's serial I/O parameters to 9600 baud, odd parity, and seven bits per character. A few simple functions can then be created for laser disk communications. The Panasonic laser disk accepts single character commands preceded by a byte with the value X'02' and followed by X'03'. For example, "G" is the command to record a frame at the current frame counter location, and increment the frame counter. Thus, a function can be created to save the next frame in a sequence (current picture on the raster display) when triggered in the following manner.



   Similarly, functions for ENABLE_RECORDING and DISABLE_RECORDING can be created. (The laser disk does transmit status messages, which will be ignored in this example for purposes of simplicity.)

2. Transformational updates between frames

   Of course, to obtain a moving sequence on laser disk, updates to data-structures will have to occur between renderings. Thus the controlling network will define two functions, UPDATE_TRANSFORMATION, whose first and only output fires after completion of a rendering, and UPDATE_COMPLETED, which must receive any message on its first input to signal completion of data-structure updates, so the network can continue and start rendering the next frame.


UPDATE TRANSFORMATIONS<1>---
user network for update of data structure between frames
--<1>UPDATE COMPLETED

3. Controlling network for multiple frame generation

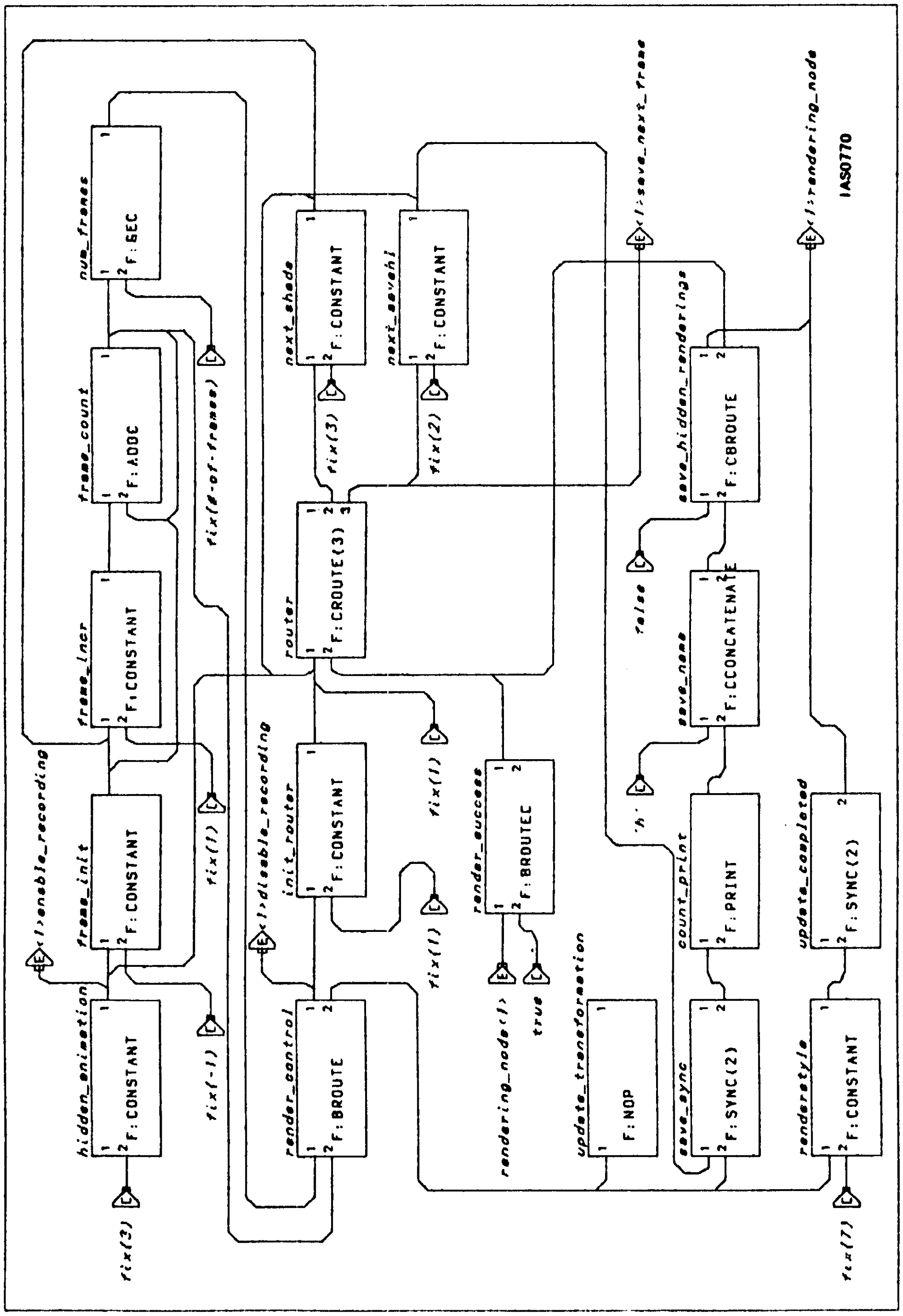


Figure 1-28. Laser Disk Controller Network

Notes

The controlling network can easily be incorporated into the shading network described in Section *GT13* by noting that the name of the rendering node in that network is WORLD.RENDERING. (This same network appears on PS 340 A1 firmware diskette B as the file "test340".) Just name the rendering node in the controlling network above WORLD.RENDERING when creating it.

```
CONN UPDATE_COMPLETED<2>:<1>WORLD.RENDERING;
CONN SAVE_HIDDEN_RENDERINGS<1>:<1>WORLD.RENDERING;
CONN WORLD.RENDERING<1>:<1>RENDER_SUCCESS;
```

Also, pass the current rendering style of the shading network (hidden, flat, etc.) as chosen by function key 3 to the controlling network.

```
CONN STYVAL<1>:<2>RENDERSTYLE;
```

Single frames can be rendered at will without affecting the controlling network. To invoke the animation control network, perform the following steps.

1. Make sure your user data-structure update network is in place and referencing the functions UPDATE_TRANSFORMATION and UPDATE_COMPLETED.

2. If the storage of each hidden-line picture generated during the rendering process as separate vector lists is desired, SEND TRUE TO <1>SAVE_HIDDEN_RENDERINGS. While the network is designed for raster animation, this mechanism is included to allow storage of the calligraphic renderings as vector lists. If used, the network will name these vector-lists H0,H1,...,Hn where n = #frames − 1. The frame by frame animation technique described in Application Note 12 could be used to implement a hidden-line animation sequence on the calligraphic display. BY DEFAULT, hidden-line pictures are not saved.

3. SEND FIX(#-of-frames) TO <2>NUM_FRAMES;

4. To invoke the frame generation process, SEND any-message TO <1>HIDDEN_ANIMATION;

Example

Once the controlling network is incorporated into a shading network, the only code remaining is the user data-structure update network. An example of a network to rotate a model 45 degrees (1 degree per frame) is listed below.

```
{the following network rotates the model about its own data-space}
{Z-axis}

TESTCONS:=F:CONS;
TESTADDC:=F:ADDC;
testzrot:=F:ZROTATE;

{cause update of rotation to occur when triggered by}
{UPDATE_TRANSFORMATION}

CONN UPDATE_TRANSFORMATION<1>:<1>TESTCONS;
CONN TESTCONS<1>:<1>TESTADDC;
SEND FIX(1) TO <2>TESTCONS;
SEND FIX(-1) TO <2>TESTADDC;
CONN TESTADDC<1>:<2>TESTADDC;
CONN TESTADDC<1>:<1>testzrot;
CONN testzrot<1>:<1>OBJECT;

{signal completion of update network by triggering UPDATE_COMPLETED}

CONN testzrot<1>:<1>UPDATE_COMPLETED;

{modify object manipulated to be model referenced by z rotation-matrix}

OBJECT:=ROT z 0 THEN scaled_mbb;
scaled_mbb:=scale by .0005 then smoothmbb;

{ render 45 ps340 frames }

SEND FIX(45) TO <2>NUM_FRAMES;

{After these commands are executed, the command to initiate the}
{rendering process is }

SEND TRUE TO <1>HIDDEN_ANIMATION;
```

Application Note 24

High Speed Communication

Erik K. Antonsson, Assistant Professor
Department of Mechanical Engineering
Division of Engineering and Applied Science
California Institute of Technology
Pasadena, CA 91125
August, 1985

Categories

Physical I/O using the parallel interface to a VAX/VMS host to send Qreals from a PS 300 variable to the Host.

Description

Frequently it is important to obtain a value from the PS 300 sent to the host very quickly. The normal routes of using HOSTOUT or HOST_MESSAGE are very slow for occasions where a single value is needed to close a computation or control loop at high speed or in real time. An example might be an aircraft flight simulation where a control dial input will be sent to the host to vary a parameter of the aircraft flight dynamics, and the simulation needs to obtain that single value and update the simulation (and the display structure) in real time, or near real time.

This solution exploits physical I/O using the Parallel Interface on a VAX 11/750 running VMS Version 4.1. The approach is to read the address and then the value at that address of a PS 300 Variable in PS 300 memory and send the value back to the host. This is done using VMS QIOs to find the address of the PS 300 named entity (Variable), then locate the address of the value, and then read the value. Since the address of the entity doesn't change, that address fetch need only be done once for each variable to be read. However, the structure of the PS 300 is such that when anything in memory is updated from a function network, the updated value is written into a free spot in PS 300 memory, and then the pointer to the value is updated in the block of data associated with the named entity. Thus two QIOs (physical I/O) are necessary for each "Read:" one to get the current address of the value in the block of data at the address and read the value. Occasionally the PS 300 updates the value in-between the two QIOs. When this happens, the value extracted is invalid. The subroutines shown here take that into account, and have a Re-Try parameter. The subroutines will re-try an operation (address fetch, or value read) up to the number of times specified in the Re-Try argument. Thus the time to get a change in value is unpredictable.

Getting single real value (PS 300 Qreal) using the serial interface and the HOSTOUT function instance, and a Fortran Read was timed to take approximately 20 milliseconds for 8 bytes. This does not include the time necessary to convert the Qreal to an ASCII string on the PS 300 nor that ASCII string into a real number on the host.

Getting a single real value using the GSRs and the Parallel Interface and the HOST_MESSAGE function instance took about 26 milliseconds for 7 bytes (also not including the ASCII conversions).

The subroutines shown here take 10.3 milliseconds per "read" of a PS 300 Qreal.

A future Application Note will describe an asynchronous version of these subroutines permitting several "reads" to be stacked up in a queue, or to allow other processing to occur in the 10 milliseconds required to get the real value.

Implementation

The subroutine set is based on the VMS system service $QIO and $QIOW (available from Fortran as QIO and QIOW). The subroutines are callable from Fortran, and adhere to the VMS calling standard. None of the values input to the subroutines are modified during the call. A test routine (PS3_PHY_TEST) is included showing how two different PS 300 Variables can be monitored, and a simple PS 300 function network to accumulate two dials into two variables is also included.

The subroutine's names are:

PS3_PHY_ATTACH
(Attaches an I/O channel to the PS 300 for physical I/O)

PS3_PHY_GET_ADDR
(Gets the address of a PS 300 named entity)

PS3_PHY_READ_VAR
(Reads the value of a variable at an address previously fetched)

PS3_PHY_DETACH
(Detaches the I/O channel used for Physical I/O)

The subroutines are heavily commented to explain the calling procedure and arguments as well as usage. The main test program demonstrates a typical calling sequence.

```
C  PS3 PHY TEST -- A test routine of Physical I/O Variable Reads
C  using the Parallel Interface.
C
C  Created: 3-August-1985 by:  Prof. Erik Antonsson, Caltech
C
          PROGRAM PS3 PHY TEST
C
C  Assign the Data Types required
C
        Integer*4 Ichan
        Integer*2 Iadrhi1,Iadrlo1
        Integer*2 Iadrhi2,Iadrlo2
        Character*80 Label1,Label2
C
    2   write(5,1002)7
 1002   format('$Enter the number of times to read each variable. ',
      & '[zero=indefinite]: ',1a1)
        read(5,1003,end=500,err=2)nloop
 1003   format(i6)
        if(nloop.lt.0)goto 2
C
C  Input the Character String Names of the PS300 Named Variables
C
    5   write(5,1005)7
 1005   format('$Enter the Name of the first PS300 Variable to
      & monitor: ',1a1)
        read(5,1010,end=500,err=5)label1
 1010   format(a)
   15   write(5,1015)7
 1015   format('$Enter the Name of the second PS300 Variable to
      & monitor: ',1a1)
        read(5,1010,end=500,err=15)label2
```

```
C
C    Attach to the PS300 for Physical I/O (ONCE)
C
         Call PS3 PHY ATTACH(Ichan)
         Iretry=10
C
C    Get the Address of the first PS300 Variable
C
         Call PS3 PHY GET ADDR(Label1,Iretry,Ichan,Iadrhi1,Iadrlo1)
C
C    If an invalid address is returned, Exit

C
          If(Iadrhi1.eq.0.and.Iadrlo1.eq.0)goto 450
C
C    Get the Address of the second PS300 Variable
C
          Call PS3 PHY GET ADDR(Label2,Iretry,Ichan,Iadrhi2,Iadrlo2)
C
C    If an invalid address is returned, Exit
C
          If(Iadrhi2.eq.0.and.Iadrlo2.eq.0)goto 450
C
            write(5,1020)
    1020    format(//)
            iloop=0
            Iretry=10
C
C    LOOP to READ the Variables Values
C
       25   Continue
            iloop=iloop+1
C
C    Read the First Variable
C
            Call PS3 PHY READ VAR(Ichan,Iadrhi1,Iadrlo1,
         &Iretry,Rvalue1,Ivalid)
C
C    Exit if Invalid data is returned
C
            If(Ivalid.le.0)goto 400
C
C    Read the Second Variable
C
            Call PS3 PHY READ VAR(Ichan,Iadrhi2,Iadrlo2,
         &Iretry,Rvalue2,Ivalid)
C
C    Exit if Invalid data is returned
```

```
C
      If(Ivalid.le.0)goto 400
C
C  Write the PS300 Variable's Values to the Terminal Screen
C
      write(5,1200)Iloop,Rvalue1,Rvalue2
 1200 format('+',I6,1PG20.10,1PG20.10)

C
C  LOOP again if the loop count (nloop) has not been exceeded
C
      if(nloop.eq.0)goto 25              !If nloop=0, Loop Forever
      if(iloop.ge.nloop)goto 500
      goto 25
C
C  ERROR Messages
C
  400 type *,'INVALID OR IMCOMPATABLE DATA RETURNED '
      goto 500
  450 type *,'ADDRESS FETCH FOR PS300 NAMED ENTITY FAILED '
      goto 500
C
C On EXIT be sure to Detach the PS300 from the Physical I/O Channel (ONCE)
C
  500 Call PS3 PHY DETACH(Ichan)
C
      call exit
      end
C
```

```
      {PS3_PHY_TEST.300};

      variable dial01;
      print1 := f:print;
      conn print1<1>:<1>message_display;
      sum1 := f:accumulate;
      conn sum1<1>:<1>print1;
      conn sum1<1>:<1>dial01;
      send 0. to <2>sum1;
      send 1. to <4>sum1;
      conn dials<1>:<1>sum1;
      send 0. to <1>sum1;
      variable dial02;
      sum2 := f:accumulate;
      conn sum2<1>:<1>dial02;
      send 0. to <2>sum2;
      send 1. to <4>sum2;
      conn dials<2>:<1>sum2;
      send 0. to <1>sum2;
```

```
C    PS3_PHY_ATTACH -- A Subroutine to Attach the PS300 Parallel Interface
C                      and Open a Channel and return a Channel Number
C
C    Created: 3-August-1985 by:  Prof. Erik Antonsson, Caltech
C
C    This routine should be called ONCE in a program that uses the other
C    PS3_PHY_* routines. This routine attaches the PS300 Parallel Interface
C    and assigns a channel number for subsequent communications. If the
C    user also plans to use the PS300 GSRs, the GSR routine PATTCH must
C    ALSO be called to attach the PS300 and open a channel for GSR
C    communication.
C
C    This routine Attaches to the Logical Device PS3PI: rather than
C    a physical device (PIA0:). So be sure to make the logical assignment
C    before executing this routine. This assignment allows the user to
C    execute the code on different devices (PIA0: or PIA1:, etc.) without
C    having to modify the code, only the logical assignment.
C
C    Note that all the PS3_PHY_* routines will work ONLY with the
C    PS300 PARALLEL INTERFACE.
C
C    Usage:
C
C        INTEGER*4 Ichan
C        CALL PS3_PHY_ATTACH(Ichan)
C
C    The argument Ichan is RETURNED by the routine, and contains the
C    channel number to use for subsequent PS3_PHY_* communication. It must
C    be declared INTEGER*4.
C
         SUBROUTINE PS3_PHY_ATTACH(Ichan)
C
         INTEGER*4 SYS$QIO,SYS$WAITFR
         INTEGER*4 SYS$ASSIGN,SYS$QIOW
         INTEGER*4 ICHAN,ISTATUS,IVALUE
         INTEGER*2 IOSB(4)
C
C    get a channel number
C
         ISTATUS=SYS$ASSIGN(%descr('PS3PI:'),ICHAN,,)
         IF(ISTATUS.EQ.1) GO TO 10
         TYPE *,'BAD ASSIGN! -- ',ISTATUS
         goto 500

C
C    Detach first for safety: 34 --> detach function code
C
```

```
     10     ISTATUS=SYS$QIOW(,%VAL(ICHAN),%VAL(34),IOSB,,,,,,,,)
C
C    Attach: 33 --> attach function code
C
            ISTATUS=SYS$QIOW(,%VAL(ICHAN),%VAL(33),IOSB,,,,,,,,)
            IF(ISTATUS.EQ.1) Return
            TYPE *,'BAD ATTACH! -- ',ISTATUS
C
C    Detach: 34 --> detach function code
C
      500   ISTATUS=SYS$QIOW(,%VAL(ICHAN),%VAL(34),IOSB,,,,,,,,)
            ichan=0
            return
            end

C    PS3_PHY_GET_ADDR -- A routine to get the address of a named entity
C                        in the PS300 memory using the Parallel Interface
C                        and Physical I/O.
C
C    Created: 3-August-1985  by: Prof. Erik Antonsson, Caltech
C
C    This routine will get the PS300 physical memory address of a PS300
C    named entity. The named entity must be set up in the PS300 BEFORE this
C    routine is called. If the named entity is not in PS300 memory before,
C    the routine will return an address of zero (0) (both High and Low 16
C    bits are zero).
C    This routine should be called ONCE per named entity whose address
C    is desired. As long as the named entity remains in the PS300 memory
C    its address will not change. This routine can be called multiple
C    times to access multiple entities in the PS300 memory. If several
C    addresses are required, different address variables must be used
C    for each one.
C
C    Usage:
C        CALL PS3_PHY_GET_ADDR(Label,Iretry,Ichan,Iadrhi,Iadrlo)
C
C    Arguments:
C        Label:  A Character String with the Name of the Entity.
C        Iretry: An Integer containing the number of times to retry
C                the address retreival if unsuccessful.
C        Ichan:  The I/O Channel Number (From PS3_PHY_ATTACH) [Integer*4]
C        Iadrhi: The high 16 bits of the PS300 entity Address [Integer*2]
C        Iadrlo: The low  16 bits of the PS300 entity Address [Integer*2]
C                Iadrhi and Iadrlo are RETURNED by this routine.
C
C    The Iretry argument is to allow the subroutine to retry the address
C    fetch if it is unsuccessful. If the Iretry argument is set to zero (0)
```

```
C    then PS3_PHY_GET_ADDR will retry the fetch indefinitely, until a
C    successful and valid address is returned.
C    If the subroutine is UnSuccessful in Iretry tries, it will return
C    zeros (0) in both Iadrhi and Iadrlo.
C    If Ichan is zero (0) when this routine is called, it will Return
C    and take no action.
C
         SUBROUTINE PS3_PHY_GET_ADDR(Label,Iretry,IchanA,Iadrhi,Iadrlo)

C
            INTEGER*4 SYS$QIO,SYS$WAITFR
            INTEGER*4 SYS$ASSIGN,SYS$QIOW
            INTEGER*4 ICHAN,ISTATUS,IVALUE
            INTEGER*4 ICHANA
            INTEGER*2 IOSB(4)
            integer*2 IrBUF(4),IrdBUF(8),Irdbu2(12)
            integer*2 iad2lo,iad2hi,iadrlo,iadrhi
            INTEGER*2 ILabel(40)
            BYTE      Blabel(80)
            Character*(*) Label
            Equivalence (Blabel,ILabel)
C
C           If(IchanA.eq.0)return
            lablen=len(label)
C
C    Truncate string length if greater than 80 characters
C
            if(lablen.gt.80)lablen=80
C
C    Move Character String into a Byte array
C
            do 12 iq=1,lablen
            blabel(iq)=ichar(label(iq:iq))
       12   continue
C
C    Strip Trailing blanks (etc) from String
C
            do 14 iq=lablen,1,-1
            if((blabel(iq).gt."40).and.(blabel(iq).lt."177))goto 15
       14   continue
            iq=0
       15   lablen=iq
C
C    Get the PS300 addresses of the entity to update
C    43 --> lookup named entities function code
C
       20   continue
```

```
            iloop=0
       21   Continue
            Ichan=IchanA
            ISTATUS=SYS$QIOW(,%VAL(ICHAN),%VAL(43),IOSB,,,ILabel,
           &%VAL(lablen),%VAL(1),,,)

C
            IF(ISTATUS.EQ.1.AND.IOSB(1).EQ.1.AND.
           &(IOSB(3).OR.IOSB(4)).NE.0) GOTO 24
            iloop=iloop+1
            if(iretry.le.0)goto 21
            if(iloop.lt.iretry)goto 21
            TYPE *,'BAD ENTITY ADDRESS FETCH! <STAT,IOSB> -- ',ISTATUS,IOSB
            goto 500
C
C    Get the address from out of the IO status block (IOSB)
C
       24   continue
            iadrlo=iosb(3)
            iadrhi=iosb(4)
            RETURN
C
      500   continue
C
C    Detach: 34 --> detach function code
C
C           Ichan=IchanA
C           ISTATUS=SYS$QIOW(,%VAL(ICHAN),%VAL(34),IOSB,,,,,,,,)
C           ichanA=0
C
            iadrlo=0
            iadrhi=0
            return
            end


C    PS3_PHY_READ_VAR -- A routine to retrieve and decode a Qreal number
C                        from a PS300 VARIABLE using the Parallel Interface
C                        and Physical I/O.
C
C    Created: 3-August-1985 by:  Prof. Erik Antonsson, Caltech
C
C    This routine will ONLY return a valid number if the entity whose
C    address is contained in Iadrhi and Iadrlo is a PS300 VARIABLE.
C    The PS300 Variable can be any named variable containing any real
C    value. However, if the variable contains a value other than a
C    PS300 Real (Integer, Character_String, Vector, Matrix) the subroutine
```

```
C    will retry the read until the Iretry argument count is reached, and
C    if no valid PS300 real data was returned, the subroutine will return
C    a zero (0.0) Rvalue and a negative Ivalid.
C    It is especially important to be sure that an initial Qreal is sent
C    to the PS300 Variable to be read so that on the first read, this
C    subroutine does not return an Invalid data condition.
C    The routine will return one REAL*4 Rvalue each time it is called.
C    This routine takes approximately 10 micro seconds to return a value.
C
C    Useage:
C           CALL PS3_PHY_READ_VAR(Ichan,Iadrhi,Iadrlo,Iretry,Rvalue,Ivalid)
C
C    Arguments:
C           Ichan:  I/O Channel Number from PS3_PHY_ATTACH
C           Iadrhi: High 16 bits of PS300 Entity (Variable) address
C                   [Integer*2]
C           Iadrlo: Low 16 bits of PS300 Entity (Variable) address
C                   [Integer*2]
C           Iadrhi and Iadrlo are from PS3_PHY_GET_ADDR
C           Iretry: The number of times to retry the read if unsuccessful
C                   (a zero argument here (0) will allow an unlimited
C                   number of retrys)
C           Rvalue: The REAL*4 RETURNED variable(with the PS300 Qreal
C                   value from the named Entity (Variable)).
C           Ivalid: An indicator of valid returned data (Rvalue)
C                   Less than zero  (0): INVALID Rvalue
C                   Greater than zero (0): VALID Rvalue
C
C    If Ichan is zero (0), or Iadrhi AND Iadrlo are both zero (0) this
C    routine will Return and take no action.
C
            SUBROUTINE PS3_PHY_READ_VAR(IchanA,Iadrhi,Iadrlo,
               &IretryA,RVALUE,Ivalid)

C
            INTEGER*4 SYS$QIO,SYS$WAITFR
            INTEGER*4 SYS$ASSIGN,SYS$QIOW
            INTEGER*4 ICHAN,ISTATUS,IVALUE
            Integer*4 IchanA
            Integer*4 Ia
            Integer*2 Ival2(2)
            INTEGER*2 IOSB(4)
            integer*2 IrBUF(4),IrdBUF(8),Irdbu2(12)
            integer*2 Iad2lo,Iad2hi,Iadrlo,Iadrhi
            Equivalence (Ivalue,Ival2)
C
            If(IchanA.eq.0)return
            If((Iadrlo.eq.0).and.(iadrhi.eq.0))return
            ia="17777777777
            Fia=float(ia)
            iloop=0
            Iretry=IretryA
C
       25   IrBUF(1)=1
            irbuf(2)=iadrlo
            irbuf(3)=iadrhi
C
C    Get 4 words (Actually will get 4 words + 8 bytes = 16 bytes = 8 words)
C
            irbuf(4)=4
C
C    Do a read phy At the Entity Address to get Address of Variable Value
C
C    39      --> Read Physical function code
C    IOSB    --> IO status block
C    IrBUF   --> Address buffer (actually address of buffer, by reference)
C    8       --> Address buffer BYTE count (4 words)
C    IrdBUF --> Data Buffer to Fill
C    16      --> Size of Read Buffer to Fill in Bytes (8 words)
C
            Ichan=IchanA
            ISTATUS=SYS$QIOW(,%VAL(ICHAN),%VAL(39),IOSB,,,
           &IrBUF,%VAL(8),IrdBuf,%VAL(16),,)
            IF(ISTATUS.EQ.1) GOTO 200
     150    TYPE *,'BAD POINTER READ! -- ',ISTATUS,IOSB
            goto 400

C
      200   continue
            IF(IOSB(1).NE.1) GO TO 150
            IF(IOSB(3).ne.16) GOTO 150
            IrdBUF(1)=1
            irdbuf(2)=irdbuf(7) !LOW Address
            irdbuf(3)=irdbuf(6) !HIGH Address
C
C   Get 8 words (Actually will get 8 words + 8 bytes = 24 bytes = 12 words)
C
            irdbuf(4)=8 !Get 8 words
C
C    Do a read phy At the Variable Value Address to get the Variable Value
C
            Ichan=IchanA
            ISTATUS=SYS$QIOW(,%VAL(ICHAN),%VAL(39),IOSB,,,
               &IrdBUF,%VAL(8),IrdBu2,%VAL(24),,)
            IF(ISTATUS.EQ.1) GOTO 205
      202   TYPE *,'BAD VALUE READ! -- ',ISTATUS,IOSB
            goto 400
C
      205   continue
            IF(IOSB(1).NE.1) GO TO 202
            IF(IOSB(3).ne.24) GOTO 202
C
C    Check to be sure Valid Data has been retrieved. A 4 always preceeds
C    a valid QReal number
C
            iloop=iloop+1
            if(irdbu2(7).eq.4)goto 300
            if(Iretry.le.0)goto 25
            if(iloop.lt.Iretry)goto 25
            goto 500
      300   continue
C
C    Convert from MC68000 Floating Long Format to VAX Real*4
C
C    Irdbu2(8)-1024 is the Exponent
C    Irdbu2(9)  is the  most significant 16 bits of the fraction
C    Irdbu2(10) is the least significant 16 bits of the fraction
C    Ivalue is an Integer*4 equivalenced to Ival2
C    A 16 bit approximation to the fraction can be obtained
C    using only Irdbu2(9) and dividing by "100000
C    Fia is a Real*4 equal to "17777777777

C
            Ival2(1)=Irdbu2(10)        !Swap Order of the Words
            Ival2(2)=Irdbu2(9)         !in the 32 bit Fraction
            Rvalue=(float(Ivalue)/Fia)*(2.0**float(Irdbu2(8)-1024))
            ivalid=1
            return
C
      400   continue
            write(5,1020)(iosb(iq),iq=1,4) ! Octal
            write(5,1021)(iosb(iq),iq=1,4) ! Decimal
     1020   format(' IOSB: ',4O8,' Octal ')
     1021   format(' IOSB: ',4I8,' Decimal ')
      500   Rvalue=0.0
            ivalid=-1
            return
            end
```

```
C    PS3_PHY_DETACH -- A Routine to Detach the PS300 Parallel Interface
C                      from the Open Channel and to Close the I/O Channel
C
C    Created: 3-August-1985 by:  Prof. Erik Antonsson, Caltech
C
C    This routine should be called ONCE in a program that uses the other
C    PS3_PHY_* routines. This routine detaches the PS300 Parallel Interface
C    and frees the channel number. If the user is also using the PS300
C    GSRs, the GSR routine PDETCH must ALSO be called to detach the PS300
C    from GSR communication.
C
C    Useage:
C
C           INTEGER*4 Ichan
C           CALL PS3_PHY_DETACH(Ichan)
C
C    The argument Ichan is Input to the routine, and contains the channel
C    number used for PS3_PHY_* communication. It must be declared
C    INTEGER*4. If this routine determines that the Ichan Argument is zero
C    (0) it will return, and take no action. Other PS3_PHY_* routines may
C    set Ichan to zero (0) if an error was detected, and the communication
C    is detached.
C
            SUBROUTINE PS3_PHY_DETACH(IchanA)
            INTEGER*4 ICHAN,ISTATUS
            Integer*4 IchanA
            INTEGER*2 IOSB(4)
C
C    Detach: 34 --> detach function code
C
            If(IchanA.eq.0)return
            Ichan=IchanA
            ISTATUS=SYS$QIOW(,%VAL(ICHAN),%VAL(34),IOSB,,,,,,,,)
            return
            end
```

TT2. HELPFUL HINTS

TT2. HELPFUL HINTS

CONTENTS

ILLUSTRATIONS

Section TT2
Helpful Hints

1. How to Make a SITE.DAT File

Categories:

SITE.DAT file

Description:

The SITE.DAT file is an optional command file created by the system manager. It can be used to tailor the PS 390 system default parameters to the specific requirements of a site. The commands in a SITE.DAT file are automatically read from the firmware diskette and executed when the system boots. The SITE.DAT file may be used to store the following types of information across power-up sequences:

- Host/PS 390 communications node address (DECNET or Ethernet interface)
- Host/PS 390 communications identity number (IBM interface)
- PS 390 Port Setup values
- Terminal Emulator keyboard and display features
- Special Site Configuration Commands

The SITE.DAT file is assumed to contain a string of ASCII commands. The file should be as compact as possible due to limited space on the diskette.

NOTE

You should make a backup copy of the graphics firmware diskette before the SITE.DAT file is downloaded to the diskette. If this is a new system, the PS 390 control unit, display and keyboard must be installed before you can create and download the SITE.DAT file.

Analysis And Implementation:

There are three methods for creating and installing the SITE.DAT file:

1. Creating the SITE.DAT file (an ASCII file) on the host and downloading to the PS 390 over an asynchronous line.
2. Entering the PS 390 command mode and typing the SITE.DAT commands directly to the diskette from the PS 390 keyboard. This method is not recommended since it involves writing directly to diskette and therefore does not provide for error correction.
3. Using GSRs.

NOTE

> All PS 390 commands in SITE.DAT must be terminated with a semicolon.
>
> The SITE.DAT file is executed in configure mode. This means the appropriate suffix must be attached to each function name.

Creating the SITE.DAT File on the Host and Downloading

You may create a text file on the host that contains the desired SITE.DAT commands and three special routing characters. The routing characters direct the PS 390 to write the ASCII commands to the SITE.DAT file on the firmware diskette, close the file and return control to the PS 390 Terminal Emulator.

The file you create on the host contains the following commands:

```
^\:                      The file must begin with the demuxing character
                         ^\ (control key and \ pressed simultaneously)
                         and the routing byte : which causes the ASCII
                         commands to be written on the firmware diskette.

PS390 command;           The command(s) you are using to configure your
                         system. Note that each command must end with a
                         semicolon.

^\;                      The demuxing character ^\ and routing byte ;
                         must precede the command to close the file
                         on the diskette.
```

```
CLOSE SITE;              The command that closes the file on diskette.

^\>                      The demuxing character ^\ and the routing byte >
                         restore output to the terminal emulator.
```

Once the SITE.DAT file exists on the host, boot the PS 390 from your backup disk and access Terminal Emulator mode, then enter the command to type the host file, as follows:

```
For VMS  -   type site.dat

For UNIX -   stty raw -echo; cat site.dat; stty cooked echo;
```

The routing bytes channel the commands to the SITE.DAT file on the diskette. You now reboot the system using the newly created SITE.DAT.

NOTE

> Your text editor controls the method you use to insert the control character ^\ in an ASCII file. For example, the VMS editor EDT uses the key sequence:
>
> ```
> gold-28-gold-specins
> ```
>
> or the EDT.INI command file line:
>
> ```
> define key control \ as "(28asc)"
> ```
>
> to define the ^\ character as ASCII 28.

The following is an example of a SITE.DAT file created on the host.

```
^\:
SEND ´042E´ TO <1>ei_o1$;
SEND ´any welcome message´ TO <1>ES_TE1;
SETUP INTERFACE PORT40/SPEED=2400/XON_XOFF;
^\;
CLOSE SITE;
^\>
```

The commands send the hexidecimal DECNET node address for a PS 390 JCP node # 1.46, and sets Port 4 to a baud rate of 2400 and enables it to use X_ON X_OFF.

Installing the SITE.DAT File Directly from the PS 390 Keyboard

This method is not recommended since it involves writing directly to diskette and does not provide for error correction. However, if you cannot communicate with the PS 390 over an asynchronous line, the SITE.DAT information required to use an Ethernet interface may be entered directly from the PS 390 keyboard as follows.

Boot the PS 390 from your backup diskette.

Access command mode on the PS 390 keyboard by simultaneously pressing the LINE LOCAL key and the CTRL key, then push RETURN. The command mode prompt @@ should appear. Enter the following command lines:

```
CONFIGURE A;
SEND ´SEND ´´042E´´ TO <1>EI_O1$;´ TO <1>WDA0;
SEND ´CLOSE SITE;´ TO <1>WDAC0;
FINISH CONFIGURATION;
```

Where:

- The A in the first line must be replaced by the appropriate password if one exists.
- The hexidecimal DECNET address for node #1.46 denoted by ´´042E´´ must have two single quotes before and after it.
- The O in EI_O1$ is the alphabetic character.
- The 0s in WDA0 and WDAC0 are zeros.

If you enter a command incorrectly from the keyboard and have not yet entered the line:

```
SEND ´CLOSE SITE;´ TO <1>WDAC0;
```

you can reboot the system and start over. However, if you have sent the CLOSE SITE command, you must delete the SITE.DAT that now exists on the diskette before you reboot and begin again.

Once the commands have been entered correctly, reboot the system using the diskette which now contains the SITE.DAT file.

The following commands demonstrate the same sequence to provide a UNIX Ethernet PS 390 JCP node address for a node # 192.6.10.8.

```
CONFIGURE A;
SEND ´SEND ´´C0060A08´´ TO <1>EI_01$;´ TO <1>WDA0;
SEND ´CLOSE SITE;´ TO <1>WDAC0;
FINISH CONFIGURATION;
```

Refer to the Customer Installation and User Manual for your particular interface for a discussion of node addressing.

Installing the SITE.DAT File Using GSRs

The GSR software source files must be loaded on the host and linked to each user application program.

The following examples demonstrate the basic GSR calls needed to create the SITE.DAT file using the Fortran and Pascal GSRs on a system with asynchronous communication. The GSR commands do the following:

1. Enable PS 390/host communications
2. Define the demuxing channel to diskette
3. Send one (or more) SITE.DAT commands out the demuxing channel
4. Define the channel to close the file
5. Close the file
6. Detach the PS 390 from host communications.

FORTRAN GSRs:

```
EXTERNAL ERR

CALL PATTCH(´LOGDEVNAM=TT:/PHYDEVTYP=ASYNC´)
CALL PMUXG(11,ERR)
CALL PPUTG(´SEND CHAR(65) TO <1>MESSAGE_DISPLAY1;´,37,ERR)
CALL PPUTG(´SETUP INTERFACE PORT40/SPEED=2400/XON_XOFF;´,44,ERR)
CALL PMUXG(12,ERR)
CALL PPUTG(´CLOSE SITE;´,11,ERR)
CALL PDTACH(ERR)
END

SUBROUTINE ERR(ERROR)

INTEGER*4 ERROR
```

```
WRITE(6,1) ERROR
FORMAT('ERROR = ',15)
STOP
END
```

PASCAL GSRs:

```
PROGRAM SITE(INPUT,OUTPUT);

CONST
%INCLUDE PROCONST

TYPE
%INCLUDE PROTYPES
%INCLUDE PROEXTRN

PROCEDURE ERR;

BEGIN
WRITELN('ERROR IS: ', ERROR);
HALT;
END;

BEGIN
PATTACH('LOGDEVNAM=TT:/PHYDEVTYP=ASYNC');
PMUXG(11,ERR);
PPUTG('SEND CHAR(65) TO <1>MESSAGE_DISPLAY1;',ERR);
PPUTG('SETUP INTERFACE PORT40/SPEED=2400/XON_XOFF;',ERR);
PMUXG(12,ERR);
PPUTG('CLOSE SITE;',ERR);
PDETACH(ERR);
END
```

Further Information:

Helpful Hints Topic 2, *How to Deal With Naming Conventions And Configure Mode.*

Section *RM10, Terminal Emulator*

2. How To Deal With Naming Conventions and Configure Mode

Categories:

naming conventions, configure mode

Description:

Naming conventions

When you instance a function or name something, the command interpreter assigns a specific suffix to that name corresponding to the suffix assigned to that instance of the command interpreter. Name suffixing is used to distinguish system level names and instances from user-defined names and instances. The following suffixing scheme is used for the PS 390:

```
0   suffix for system related functions associated with the user. The
    names with this suffix are not accessible in command mode.

1   suffix for user-defined and accessible names associated with The
    user. All names with this suffix are accessible to the user.

    (If the command interpreter is suffixed with a 0 or a 1, it
    will suffix names that it creates with a 1.)
```

Configure Mode

Name suffixing automatically occurs in command mode, but it does not automatically occur in configure mode. Configure mode is a privileged mode of operation that the command interpreter uses to create or modify system functions. You must suffix any function that is instanced whenever you are working in configure mode, whether you are using system-level or user-level names. You must also properly suffix the command interpreter to assure that other functions created by the command interpreter will have the appropriate suffix. Configure mode is provided as a way to protect system-level names and functions from being erroneously modified. Therefore, you cannot access any system-level names or functions without being in configure mode.

Analysis And Implementation:

Naming Conventions

Name suffixing is handled by the command interpreter when you are in command mode, and is completely invisible to you. For example, when you are in command mode on a single-user system and instance:

```
ADD := F:ADD;
```

The command interpreter creates an instance of the function F:ADD with the name of ADD1 (a suffix of 1 is used because ADD is a user-defined name created with the system command interpreter, called CI0). If you were to send:

```
SEND 2.5 TO <1>ADD;
```

to the function that you just instanced, the command interpreter would send the value 2.5 to the function ADD1.

Configure mode

You can only access system-level names and functions in configure mode, in which you have the capability of reconfiguring system functions. Use the following command to set up a password for configure mode:

```
@@ SETUP PASSWORD password;
```

Where **password** is the password to enter into configure mode. This command allows you to establish and modify the password required to enter configure mode. This command can be included in the SITE.DAT file, or may be set up at any time thereafter (there will be no password established prior to you issuing this command).

The commands that allow you to enter and exit configure mode are as follows:

`@@ CONFIGURE password;`(issue in command mode to enter configure mode)

Where **password** is the established string. If no password has been defined, any string can be entered for the password.

```
@@ FINISH CONFIGURATION;
```

This command takes the PS 390 out of configure mode back into command mode. This command must be entered after any modifications to system-level functions or names have been made.

When you boot your PS 390 with the appropriate firmware, a file called CONFIG.DAT is read from your firmware diskette. This file contains the initial instances of system-level commands and functions. While reading this file, the command interpreter is in configure mode.

The last thing that the CONFIG.DAT network does is search for a file called SITE.DAT on your firmware diskette. If a SITE.DAT file is found, it is read and processed. You must perform all name suffixing when you create a SITE.DAT file to be read from your firmware diskette. The SITE.DAT file is read by the CONFIG.DAT while in configure mode. For example, if you created a SITE.DAT file which had a command in it to change the background color to red, you would need to write:

```
@@ SEND V3D (120.0,1.0,1.0) TO <2>PS390ENV1;
@@ SEND TRUE TO <1>PS390ENV1;
```

While if you wanted to change the background color to red from command mode, the command would need to be:

```
@@ SEND V3D (120.0,1.0,1.0) TO <2>PS390ENV;
@@ SEND TRUE TO <1>PS390ENV;
```

Further Information:

Helpful Hints Topic 1, *How to Make a SITE.DAT File.*

3. Hardcopy Networks using Writeback

Categories:

writeback, plotters

Description:

Writeback allows you to receive picture information from the PS 390. The writeback node, defined by WB$, occurs at the top level of the display structure. Picture information is a description of the transformed vectors located beneath the writeback node and can be converted to instructions for a plotter. There are two ways to convert the information:

- Running a program from the host.
- Running a function net program on the PS 390.

Analysis And Implementation:

Running a program from the host

When you run the **plotinfo.exe** program, you create two files of picture instructions on the host. These files are talked about in the **plotinfo.doc** file. These host files are then converted into plotter commands for the appropriate plotter. The plot routines in the plot directory supports three plotters:

- Hewlett Packard 7550A
- Apple Laser Writer
- Tektronix 4510 rasterizer hooked to a 4691 jet plotter

The programs to do these conversions are described in the following files:

- hpplot.doc
- lwplot.doc
- tekplot.doc

Running a function net program on the PS 390

A network defined in **plot.fun** allows you to trigger the writeback feature with the PS 390 hardcopy key. This network has the WB$ node connected to a Plotinfo function. The Plotinfo function puts out generic picture commands for a plotter function.

NOTE

> The only plotter function supported by the PS 390 is the Hpplot user-written function for the Hewlett Packard hp7750a plotter.

The Hpplot function receives commands from the Plotinfo function and puts out commands to the port to which it is connected. The Plot.Fun file connects the plot function to output port 4. Both the Plotinfo and Hpplot function are User-written functions that need to be loaded down to the PS 390. The functions are defined in the Plotinfo.uwf and Hpplot.uwf files. These functions are described in the Plotinfouwf.doc and Hpplotuwf.doc files.

Further Information:

Section *RM2 Intrinsic Functions, RM14 GSR Internals*

4. How to Deal with XFORMDATA

Categories:

XFORM VECTOR, XFORM MATRIX

Description:

Before reading this, thoroughly read the Section *TT9 Transformed Data and Writeback*. You should be familiar with the information contained in that section.

Note the following restrictions on the use of transformed data on the PS 390:

- The PS 390 does not allow the display of transformed data (F:XFORMDATA outputs a non-displayable data type — vector-normalized vector list).
- A single-precision vector list is generated by F:XFORMDATA.
- Only three-dimensional data can be transformed.
- F:XFORMDATA can still be connected to F:LIST to enable the host to read the transformed data retrieved from the PS390.

XFORM VECTOR affects all transformations applied to the data node(s), whether these transformations are above or below the XFORM VECTOR node.

XFORM MATRIX affects only those transformations above the XFORM MATRIX node. XFORM MATRIX should be placed immediately above the data node(s) to include all transformations applied to the data node(s).

There is a limit of 2048 vectors that may be read back with single trigger of the XFORM VECTOR node.

Analysis And Implementation:

The following examples show how XFORM MATRIX and XFORM VECTOR might be used. As the code is, an output is not produced because the LIST<1> is not connected to another function. Output of LIST<1> can be verified in one of three ways:

- LIST<1> can be connected to a debug port (if available)
- LIST<1> can be connected to the function ES_TE
- LIST<1> can be connected to the function HOST_MESSAGE, and messages may be polled from HOST_MESSAGE via FORTRAN/Pascal GSR programs.

You can verify the XFORM VECTOR example is functional by removing the displayed data and displaying the transformed data:

```
@@ REMOVE XFORM;
@@ DISPLAY XDATA;
```

Example Of XFORM MATRIX

```
init;

xform := begin_structure
       x := set conditional_bit 1 on;              {Set up conditional    }
            if conditional_bit 1 is on then view;  {referencing so that   }
            if conditional_bit 1 is off then tran;{the viewing, etc. can }
            end_s;                                 {be replaced by the    }
                                                   {identity matrix.      }

tran := begin_structure
            matrix_4x4 1,0,0,0 0,1,0,0 0,0,1,0 0,0,0,1; {structure to be }
            instance of obj;                            {transformed- no }
        end_s;                                          {viewing, etc.   }
                                                        {applied         }

view := begin_structure
            window x = -5:5 y = -5:5;     {structure to be displayed- }
            instance of obj;              {all viewing, etc. applied  }
        end_s;

obj  := begin_structure
            trans := translate by 0.2,0.4,0.0;
            xform_request := xform matrix;    {xform matrix request - }
            instance of data;                 {note that it is just   }
       end_s;                                 {prior to the data node }
```

```
data := vec item n=2 -.5,.5,0 .5,-.5,0;

xformdata := f:xformdata;
sync2 := f:sync(2);
list := f:list;
conn sync2<1>:<1>xformdata;
conn xformdata<1>:<1>list;
conn list<2>:<2>sync2;
conn list<2>:<1>xform.x;
send 'go' to <2>sync2;
send 'obj.xform_request' to <2>xformdata;
send 'xdata' to <3>xformdata;
display xform;



{The following network sets up function key 1 such that it performs the }
{two commands:                                                           }
{              @@ send false to <1>xform.x;                              }
{              @@ send false to <1>sync2;                                }
{                                                                        }
{which triggers the conditional bit 1, and then triggers the xform      }
{vector node to return the transformed data.                            }


sync := f:sync(2);
false := f:constant;
route := f:route(12);
send 'xform' to <1>flabel1;
send false to <2>false;
conn fkeys<1>:<1>route;
conn fkeys<1>:<2>route;
conn route<1>:<1>false;
conn fkeys<1>:<1>sync;
conn false<1>:<1>xform.x;
conn false<1>:<2>sync;
conn sync<2>:<1>sync2;
```

This example should produce the following from LIST<1>:

```
 XDATA := MATRIX_4X4 1.000000E+0,0.000000E+0,0.000000E+0,0.000000E+0
                     0.000000E+0,1.000000E+0,0.000000E+0,0.000000E+0
                     0.000000E+0,0.000000E+0,1.000000E+0,0.000000E+0
                     1.999999E-1,3.999999E-1,0.000000E+0,1.000000E+0;
```

Example Of XFORM VECTOR

```
init;

xform := begin_structure
       x := set conditional_bit 1 on;             {Set up conditional     }
            if conditional_bit 1 is on then view; {referencing so that    }
            if conditional_bit 1 is off then tran; {the viewing, etc. can }
         end_s;                                   {be replaced by the     }
                                                  {identity matrix.       }

tran := begin_structure
            matrix_4x4 1,0,0,0 0,1,0,0 0,0,1,0 0,0,0,1; {structure to be }
            instance of obj;                            {transformed- no }
        end_s;                                          {viewing, etc.   }
                                                        {applied         }

view := begin_structure               { structure to be displayed- }
            instance of obj;          { all viewing, etc. applied  }
        end_s;

obj := begin_structure
           xform_request := xform vector; {xform vector request - note  }
           instance of data;              {that it is just prior to the }
       end_s;                             {data node                    }

data := vec item n=2 -.5,.5,0 .5,-.5,0;  {Only three-dimensional data can }
                                         {be transformed on a PS390       }

xformdata := f:xformdata;
sync2 := f:sync(2);
list := f:list;
conn sync2<1>:<1>xformdata;
conn xformdata<1>:<1>list;
conn list<2>:<2>sync2;
conn list<2>:<1>xform.x;
send 'go' to <2>sync2;
send 'obj.xform_request' to <2>xformdata;
send 'xdata' to <3>xformdata;
display xform;

{The following network sets up function key 1 such that it performs  }
{the two commands:                                                    }
{                  @@ send false to <1>xform.x;                       }
{                  @@ send false to <1>sync2;                         }
{                                                                     }
{which triggers the conditional bit 1, and then triggers the xform   }
{vector node to return the transformed data.                          }
```

```
sync := f:sync(2);
false := f:constant;
route := f:route(12);
send 'xform' to <1>flabel1;
send false to <2>false;
conn fkeys<1>:<1>route;
conn fkeys<1>:<2>route;
conn route<1>:<1>false;
conn fkeys<1>:<1>sync;
conn false<1>:<1>xform.x;
conn false<1>:<2>sync;
conn sync<2>:<1>sync2;
```

This example should produce the following from LIST<1>:

```
XDATA := VECTOR_LIST ITEMIZED N=2 P --0.500000,0.500000,0.000000 I=0.992188
                                  L 0.500000,-0.500000,0.000000 I=0.992188;
```

Further Information:

Section *TT9 Transformed Data and Writeback*

5. How to Render Spherical and Line Data Types

Categories:

rendering

Description:

Spherical renderings on the PS 390 use input <5> of the surface/solid rendering node and transformed data node information. To change definitions of the spherical renderings in the attribute table, you need to create a tabulated vector list that contains spherical definitions.

Analysis And Implementation:

To render a sphere, perform the following:

1. Place an instance of the surface/solid rendering node in the main structure of the object to be rendered. The rendering node should be placed below the instance of the vector data to be used for the spherical rendering.
2. Create spherical data definition using tabulated vector lists where each X,Y,Z coordinate specification represents the spherical center of the rendered sphere. The tabulated value represents the spherical attributes associated with the table value as loaded in the attribute table. The following is an example of spherical vector data definition with the default system attribute table:

```
raster_spheres := vector list tabulated n = 3
                      P 1,0,0  t=5   {yellow sphere}
                      L 1,1,0  t=4   {blue sphere}
                      L 0,1,0  t=2   {red sphere}
```

Because the transformed data function uses vector normalized vector data, two definitions of the spherical data must be established in mass memory that are identical in value. One list is used for display information and is block normalized.The other list follows a trigger of the ALLOW_VECNORM function and is a vector normalized vector list in mass memory. This vector list is not displayed since the PS 390 does not display vector normalized vector lists.

3. Setup transformed data node to provide transformed data information to the rendering node and connect output <1> of the xformdata node to input <4> of the rendering node.
4. Send the name of the spherical vector list data to input <5> of the rendering node.
5. Trigger the rendering xformdata node and the rendering node to produce the rendering.

NOTE

> To avoid timing problems, use a SYNC(2) function from the function keys to simultaneously trigger the rendering xformdata node and the rendering node. Connect sync function output <1> to the xformdata function and output <2> to the rendering node.

Reloading the Attribute Table

To change the definitions of the spherical renderings in the attribute setup a vector list similar to the following:

```
ATTRIBUTE_TABLE_VEC := VECTOR LIST TABULATED N = 6
                       0,1,1      2,.5,4     T=5
                       120,1,1    4.0,0.8,9  T=6
                       240,1,1    3.0,0.3,2  T=7;
```

Where each table entry is specified by two X, Y, Z vector components. The first X, Y, Z, component is for hue, saturation, and intensity, and the second X, Y, Z component is for radius, diffuse, and specularity of that entry.

Update the table by entering the command:

```
@@ send 'attribute_table_vec' to <14>shadingenvironment;
```

NOTE

> When rendering spherical data, the sphere will not be rendered if the vector list is defined or transformed such that the front or back clipping planes clip the radius of the rendered sphere. Be sure front and back clipping planes are setup so as not to interfere with the spherical rendering.

The following is an example of basic raster_spheres rendering network.

```
init;

 reserve_working_storage 300000;

  allow := f:allow_vecnorm;

 display render_sphere;

 render_sphere := begin_s
   set depth_clipping on;
   set contrast 0;
   window x = -5:5 y=-5:5 front = -10.0 back = 10;
   instance of disp_sphere;
   rendering := surface_rendering;
   {instance of polygon objects}
 end_s;

  disp_sphere := begin_s
   linexform := xform vector;
   { viewing, windowing, etc. }
   { rotate, scale, translate }
   instance of sphere_vecn,sphere_blockn;
 end_s;

  sphere_blockn := vector  tabulated n=3
        P 1,0,0 t=2    {red sphere}
        L 1,1,0 t=5    {yellow sphere}
        L 0,1,0 t=4;   {blue sphere}

 {allow following vector definition to be vector normalized }
 send true to <1>allow;
 give_up_cpu;
 give_up_cpu;
 give_up_cpu;
 give_up_cpu;

 sphere_vecn := vector  tabulated n=3
        P 1,0,0 t=2    {red sphere}
        L 1,1,0 t=5    {yellow sphere}
        L 0,1,0 t=4;   {blue sphere}

 send false to <1>allow;

 { setup xformdata node }
 linexformdata := f:xformdata;
 send ´disp_sphere.linexform´ to <2>linexformdata;
```

```
conn linexformdata<1>:<4>render_sphere.rendering;
send 'sphere_vecn' to <5>render_sphere.rendering;

{ setup network to turn on the display upon render completion }
con := f:constant;
send fix(0) to <2>con;
conn render_sphere.rendering<1>:<1>con;
conn con<1>:<1>turnondisplay;

{setup render trigger to be any function key}
rsyn := f:sync(2);
conn fkeys<1>:<1>rsyn;
setup cness true <2>rsyn;
send fix(7) to <2>rsyn;
conn rsyn<1>:<1>linexformdata;
conn rsyn<2>:<1>render_sphere.rendering;

{ initialize shadingenvironment function }
send v3d(120,0,500) to <3>shadingenvironment;
send fix(1) to <5>shadingenvironment;
```

Further Information:

Section *TT9 Transformed Data and Writeback*

6. Physical I/O GSR issues

Categories:

Ethernet interface, parallel interface, I/O commands

Description:

The physical I/O commands permit the host to directly access the internal contents of any node or PS 390 structure. The I/O commands take full advantage of the speed of the Ethernet or parallel interfaces to modify the contents without any node swapping, pointer juggling, memory management, or command interpretation. For a more complete description of the physical I/O commands refer to Section AP4 of the *PS 390 Document Set* and the appropriate Customer Installation and User Manual for your interface.

Analysis And Implementation:

The following is a high level description of a physical I/O programming example. This programming example can be run over the Ethernet or parallel interfaces. To run this program read the file **phy_example.doc** located on the host software tape in the **miscellaneous** sub-directory.

The source for this program is located in the **phy_example.for** file.

The program updates two matrices that rotate two objects, one around the Y axis, the other around the Z axis. These two matrices are double buffered. The double buffering is accomplished by having two paths through the display structure.

Path1 has one copy of the Y and Z matrices, path2 has the other copy of the matrices. The names of the Y rotation matrices are **path1.yrot** and **path2.yrot**. The names of the Z rotation matrices are **path1.zrot** and **path2.zrot**. The conditional **pivots.bit** controls whether path1 matrices or path2 matrices are used.

The program first gets the physical address of these five entities and then continuously switches updates of path1 and path2 matrices. Each time an update is done the conditional bit is also updated so the appropriate path is taken in the display structure. The result is smooth rotations of the objects around the Y and Z axis.

Further Information:

Advanced Programming, AP1–9 and the Customer Installation and User Manual for your interface

7. Host Communication Data Flow — How the Interfaces Deal with Runtime

CATEGORIES:

communication interface, runtime environment

DESCRIPTION:

The host system communicates with the PS 390 runtime environment via a communication interface. The communication interface uses two types of communication packets, count mode and escape mode.

Count mode

Count mode packets begin with the start of packet (SOP) character (6), followed by two bytes of count data, followed by the data. The first byte of data should be a muxing byte that tells the PS 390 where the data goes.

As an example, a sample count mode packet, in hex, might look like:

```
06 00 06 30 49 4E 49 54 3B
```

Where:

06 signifies the beginning of a packet

00 06 signifies there are 6 bytes following

30 signifies that this is an ASCII command which needs to go to the chopper/parser

49 4E 49 54 3B are the ASCII character codes for "INIT;"

This type of packet is generated automatically by the GSRs and is the standard method of communication with the PS 390.

Escape mode

Escape mode packets start with an ASCII FS character (Hex 1C), followed by the muxing byte, followed by data. Since there is no count associated with an escape mode packet, the system assumes that all data is part of the current packet until another SOP character is received. The escape mode packet is commonly used when you place the FS and the mux bytes directly in the data file destined for the PS 390 and use a host system command such as TYPE or COPY to transfer the file to the PS 390 over the asynchronous line.

Analysis And Implementation:

The runtime communication environment can be conceptually divided into five function boxes.

```
    ==================      ================        ===============
    | input handling |      |   depacket   |------->|   ciroute   |---->
    |    function    |      |   function   |        |   function  |---->
 -->| accepts data   |----->| breaks host  |        | routes data | .
    | from host      |      | data into    |        | to various  | .
    |                |      | Qpackets and |----    |   system    | .
    |                |      | Qmorepackets |   |    |  functions  |---->
    ==================      ================   |    ===============
                               DEPACKET0       |       CIROUTE0
                                               |
------------------------------------------------
|
|   ==================      ================
|   |    another     |      |   another    |---->
--->|    depacket    |----->|   ciroute    |---->
    |    function    |      |   function   | .
    |                |      |              | .
    |                |      |              | .
    |                |      |              |---->
    ==================      ================
        DEPACKET20              CIROUTE20
```

The input handling function is a generic description of a function that accepts data from the host and passes it to the next function in line: DEPACKET0.

DEPACKET0 is a count mode version of the F:DEPACKET function. When DEPACKET0 receives data that is not a count mode packet, it sends the data to the second F:DEPACKET function: DEPACKET20. When DEPACKET0 receives a count mode packet, it consumes the SOP and count bytes and sends out a Qpacket, followed by Qmorepackets, to CIROUTE0.

CIROUTE0 looks at incoming data packets and does one of two things; if a QPACKET is received, CIROUTE0 looks at the first byte, which should be a muxing byte, then changes the output path to match the mux byte. Data following the mux byte goes out this path. If a Qmorepacket is received, CIROUTE0 does not change the output path, and sends the data out the current output path. The various paths out of CIROUTE0 are connected to such system functions as a chopper/parser to handle ASCII commands, as well as a path to handle binary data commands.

DEPACKET20 is similar to DEPACKET0, except it looks for escape mode packets. DEPACKET20 consumes the SOP character from an escape mode packet and breaks the data up into Qpackets and Qmorepackets and sends it out output 1 to CIROUTE20. If the data is not an escape mode packet, DEPACKET20 sends the data out output 2.

CIROUTE20 functions like CIROUTE0, including a complementary set of output paths.

The appropriate packets must be built on the host side before sending to the PS 390. The common way to send ASCII PS 390 commands across an asynchronous link is to build a file using the editor which has the desired commands and required FS characters followed by mux bytes. The file is then sent to the PS 390 with an operating system command such as TYPE or COPY.

However, the standard way to communicate with the PS 390 is via the GSR library. The GSR library builds packets for all PS 390 commands in binary and ASCII format, handles efficiency considerations such as buffering of data, and knows how to deal with all the supported communication interface options such as asynchronous, parallel, Ethernet, and IBM 3278.

To reset CIROUTE and CIROUTE3 to their previous configuration type in the following commands:

```
Configure a;

disconnect ciroute0<10>:<1>rasstr0;
disconnect ciroute0<11>:<1>hpolystr0;
disconnect ciroute30<10>:<1>rasstr30;
disconnect ciroute30<11>:<1>hpolystr30;
send fix(4) to <4>ciroute0;
send fix(4) to <4>ciroute30;

Finish configuration;
```

FURTHER INFORMATION:

Section *RM5 Host Communications* and Section *RM7 Local Data Flow*

8. How to Copy Files Between the Host and the PS 390

Categories:

Description:

You can copy files from the host to a PS 390 diskette and from the PS 390 diskette to the host. You need to connect a debug terminal to your system and use the UTILITY routines booted from the diagnostic utility diskette except when downloading a file over an asynchronous line.

Analysis And Implementation:

Copying Files from Host to PS 390 Diskette

There are two methods for copying an ASCII file from the host to the PS 390:

1. Adding appropriate routing bytes to the ASCII file on the host and downloading to the PS 390 over an asynchronous line.
2. Booting from a PS 390 diagnostic disk and using the UTILITY command TRANSFER.

Downloading an ASCII File from the Host Using Routing Bytes

You must add three special routing bytes to your text file on the host. These routing bytes direct the PS 390 to write the host file to the firmware diskette, close the new file and return control to the PS 390 terminal emulator.

The file you create on the host contains the following special characters:

`^\:` The file must begin with the demuxing character ^\ (control key and \ pressed simultaneously) and the routing byte : which causes the following lines of the file to be written on the firmware diskette.

`FILE BODY` The file lines are assumed to be ASCII.

`^\;` The demuxing character ^\ and routing byte ; must precede the command to close the file on the diskette.

```
CLOSE<filename>;      The PS 390 command that closes the file on
                      diskette. Note you DO NOT ADD A FILE EXTENSION
                      HERE. When you use this method, the system will
                      automatically append the extension .dat to the file
                      name you give here.

^\>                   The demuxing character ^\ and the routing byte >
                      restore output to the terminal emulator.
```

Once the routing bytes have been added to the file that exists on the host, boot the PS 390 and enter terminal emulator mode. Give the command to type the host file:

```
For VMS  -   type <filename.>
For UNIX -   stty raw -echo; cat <filename.>; stty cooked echo;
```

The routing bytes channel the commands to the **filename.dat** file on the diskette.

NOTE

> You must insert a space at the start of each line since the first character of each line is lost. The method you use to insert the control character ^\ in an ASCII file depends on the text editor you are using. For example, the VMS editor EDT uses the sequence:
>
> ```
> gold-28-gold-specins key
> ```
>
> or the **edt.ini** command file line:
>
> ```
> define key control \ as "(28asc)"
> ```
>
> to define the ^\ character as ASCII 28.

The following example shows a file containing the necessary routing bytes.

```
^\:
These two lines of text will be copied onto the PS 390 diskette.
The file will be labeled: SAMPLE.DAT.
^\;
CLOSE SAMPLE;
^\>
```

Using Utility Routines for Uploading and Downloading

You use the TRANSFER utility routine to copy files from the host to a PS 390 diskette, and the SENDBACK routine to copy files from the PS 390 diskette to the host. You need to connect a debug terminal to your system to fully use the UTILITY routine screen prompts and messages.

To transfer a file, boot your system from the diagnostic utility diskette. If you have a data tablet connected to your system, be sure the puck is not on the tablet when booting. When the debug terminal indicates the diagnostics have completed through 'O', hold the control key down while you slowly type the letter 'p' 5 or 6 times. The system should respond by identifying the operating system, disk name and the message: type HELP for additional help, followed by the = prompt. When you see the = prompt type:

```
utility <cr>
```

You then get the utility> prompt and are ready to begin the file transferring process.

Checking Host Communications Settings

The first step in checking the communication settings is to ensure the default communication parameters between your host and the PS 390 are compatible. The MODIFY utility routine displays the current settings and allows you to modify those that are not correct. At the utility> prompt type:

```
modify <cr>
```

The MODIFY routine lists a menu of all the setting selections when you enter the number 0. Select menu number 1 to see the current default settings for all the parameters.

NOTE

> Generally, you only need to verify that the baud rate (#4) is appropriate, and that the sendback (#7) and transfer (#9) strings and terminator strings (#8, #10) are compatible with your host. For example, on VMS, the appropriate transfer string:

```
TYPE <fn><cr>
```

is the VMS command that will cause the host to send the file <fn> to the PS 390. The appropriate terminator string is the VMS prompt "$ " which indicates that the file transfer has been completed.

If the default settings are compatible with your host, you may exit from the MODIFY utility by typing a carriage return and proceed with the file copying as described in the next sections. If one or more of the default settings needs to be changed, refer to the section on Modifying Host/PS 390 Communication Parameters under this topic before attempting the copy procedures.

Copying from Host to PS 390 Using the TRANSFER Utility

If the communication parameters displayed by the MODIFY utility are appropriate for your host, you may proceed to copy a file of up to 30,000 bytes from the host to the PS 390 diskette as follows:

1. At the Utility> prompt, type:

   ```
   terminal <cr>
   <cr>
   ```

   to log onto the host account containing the file to be transferred. This places the PS 390 in terminal emulator mode and causes the system to prompt for your login.

2. After you have logged onto the appropriate account, set the terminal to NOECHO. If you fail to turn the echo off, your newly copied file will contain the transfer string as its first line. The commands to turn character echoing off are:

   For VMS - `set term/noecho`
   For UNIX - `stty raw -echo`

3. Simultaneously press the control key and letter A to return to the PS 390 utility prompt. At the utility> prompt type:

   ```
   transfer <cr>
   ```

4. The transfer utility will prompt you for:
   - The name of the file exactly as it appears on the host.

- The name of the file to be created on the PS 390. File names may contain 1 to 8 alphanumeric characters plus an extension .DAT (data), .TXT (text) or .COM (communication).
- The type of transfer (ASCII or S-record).
- The date in the format dd-mmm-yy.

The file transfer starts when you have supplied the above information. A dot appears for each line transferred, then a message stating the transfer was successful appears when the transfer is complete. At the utility> prompt and you can repeat the process for another file. If you return to terminal emulator mode, you should set the character echo back on as follows:

For VMS - set term/echo
For UNIX - stty cooked echo

Copying from the PS 390 to the Host Using the SENDBACK Utility

If the communication parameters displayed by the MODIFY utility are appropriate for your host (including the XON/XOFF characters), you may copy a file from the PS 390 diskette to the host as follows.

1. At the Utility> prompt, type:

   ```
   terminal <cr>
   <cr>
   ```

   to log onto the host account to which the file will be transferred. This places you in terminal emulator mode and causes the system to prompt for your login.

2. After you have logged onto the appropriate account, set the terminal to host synchronization. If the host computer does not send XON/XOFF signals data may be lost. Setting the host synchronization assures that the buffers of the host computer do not overflow. The VMS command to turn host synchronization on is:

   ```
   SET TERM/HOSTSYNC
   ```

3. To return to the utility prompt, type:

   ```
   <CONTROL> A
   ```

4. At the utility> prompt type:

   ```
   sendback <cr>
   ```

5. The sendback utility prompts you for:
   - The name of the file exactly as it appears on the PS 390.
   - The name and extension of the file to be created on the host, including the full path name if the file is to be copied to a directory other than the one where you are currently logged on.
   - The type of transfer (ASCII or S-record)

The file transfer starts when you have supplied the above information. A message stating that the transfer was successful appears when the transfer is complete. At the utility> prompt you can repeat the process for another file. When you return to terminal emulator mode, you should set VMS host synchronization off as follows:

```
SET TERM/NOHOSTSYNC
```

Modifying Host/PS 390 Communication Parameters

Successful file transfers require that the default communication parameters between your host and the PS 390 are compatible. The utility routine MODIFY displays the current settings and allows you to modify those that are not correct. At the utility> prompt type:

```
modify <cr>
```

The MODIFY routine lists a menu of all the setting selections when you enter the number 0. Select menu number 1 to see the current default settings for all the selections.

Enter the number 0 to return to the selection menu at any time or enter the menu number of the specific setting you wish to change.

When you select a specific parameter, the MODIFY utility shows the current setting for the parameter and prompt you step by step through the modification process.

As an example, the following steps describe how to modify the sendback string (#7) to shorten the transfer string to TY instead of TYPE and choose the # delimiter.

1. The system prompts:

   ```
   Enter a number
   ```

   You type:

   ```
   9 <cr>
   ```

   The current transfer string is displayed and you are prompted to enter the new transfer string:

   ```
   Current transfer string is:  "TYPE <fn><cr>"
   ```

   Enter a delimiter (any character but <cr>), then the message, and the delimiter again.

2. Then type:

   ```
   #TY (CONTROL F) <cr>#
   ```

   The change is made as soon as you type the final delimiter.

3. You are prompted to enter another number and repeat the process, or you can return to the main menu by typing a 0. Verify your changes by entering number 1. If they are not correct, repeat the process.

4. Exit the MODIFY utility by typing a carriage return and proceed with the file copying as described in the previous sections.

9. Routing Bytes

CATEGORIES:

routing bytes, F:CIROUTE

DESCRIPTION:

A routing byte (also called a mux byte) is that portion of a PS 390 communication packet that determines where the commands and/or data are sent. It is one byte in length. The intrinsic function F:CIROUTE uses the routing byte to determine where data following the routing byte is sent. Outputs of F:CIROUTE are connected to other functions such as CHOP to process ASCII commands, READSTREAM to process binary commands, and SREC_GATHER to process user written function code.

ANALYSIS AND IMPLEMENTATION:

There are three ways to specify routing bytes:

1. Insert the routing byte within an ASCII file with an editor. This byte must be preceded by the FS character (hex 1C) and all data following the routing byte is routed to the designated path until another FS routing byte sequence is encountered.

2. Use the special GSR routines PMuxG and PPutG. PMuxG sets the generic output channel according to the value of its parameter and subsequent calls to PPutG send data to the generic output channel.

3. The most common technique is to let the GSR's implicitly build communication packets and set the routing bytes for you.

Following is a table showing the F:CIROUTE function, its output connections and the routing bytes necessary for each output path. The connections of CIROUTE0 are subject to change.

		routing bytes	
CIROUTE0	output connection	ascii /	pmuxg
<1> <1>	reserved	N/A	N/A
<2>	<1>BADROUTE0	N/A	N/A
<3>	<1>H_CHOP0	0	1
<4>	<1>READSTREAM0	1	2
<5>	<1>SIXTOEIGHT0	2	3
<6>	<1>RESET_RS1	3	4
<7>	unused	4	5
<8>	unused	5	6
<9>	<1>SREC_GATHER0	6	7
<10>	<1>RASSTR0	7	8
<11>	<1>HPOLYSTR0	8	9
<12>	unused	9	10
<13>	<1>WDA0	:	11
<14>	<1>WDAC0	;	12
<15>	<1>WDBC0	<	13
<16>	unused	=	14
<17>	<1>ES_TE1	>	15
<18>	<1>TRIGGER_CONVB1	?	16
<19>	<1>WHO1	@	17
<20>	unused	A	18
<21>	<1>RASSTR0	B	19

FURTHER INFORMATION

Helpful Hint Topic 7, *Host Communication Data Flow/How the Interfaces Deal with Runtime*

10. How to do Patterned and Textured Vector Lists

CATEGORIES:

patterned vector lists, textured vector lists

DESCRIPTION:

Textured vector lists

The texture command is used to apply a hardware generated texture to a vector list. Textures are drawn in the same direction as the vector list was specified.

Patterned vector lists

A pattern may be applied to a vector list by specifying up to 32 integers between 0 and 128 that represent the relative lengths on the pattern segments (lines and spaces).

ANALYSIS AND IMPLEMENTATION:

Textured vector lists

The texture command has the form:

```
Name := set line_texture [around_corners] pattern [applied to x]
```

Where **pattern** is an integer in the range of 1 to 127 specifying a texture to be applied to a vector list. The texture applied is calculated by the binary representation of the of the integer given in the lower 7 bits. All textures start with a one (dash) in bit 8 of the texture.

Example 1:

```
Plain_vec   := vector_list -1,0,0 1,0,0;
texture_vec := set line_texture 122 then plain_vec;
Display texture_vec;

Integer 122 which is 01111010 in binary
would produce a texture  ----- - ----- - ----- -
over 3 intervals.       111110101111101011111010
                        |  1    |   2   |   3   |
```

Example 2:

```
Plain_vec   := vector_list -1,0,0 1,0,0;
 texture_vec := set line_texture 25 then plain_vec;
Display texture_vec;

Integer 25 which is 00011001 in binary
would produce a texture  -  --  --  --  --  --  -
over 3 intervals.        100110011001100110011001
                         |   1   |   2   |   3   |
```

Patterned vector lists

The command to apply a pattern to a vector list is:

```
name := pattern i [around corners][match/nomatch] length r;
```

Where **i** represents the integers that specify the pattern and **r** represents the interval over which the pattern is to be repeated. The pattern is applied to the vector list with the PATTERN WITH command.

Example:

```
lineX    := vector_list -1,0,0 1,0,0;
Pattern1 := pattern 5,2,2,5 length 1;
Pattern linex with pattern1;
Display linex;
```

Produces a patterned vector list over three intervals as shown below:

```
-----  --     -----  --     -----  --
5  2  2   5   5  2  2   5  5   2  2   5
```

11. Discussion of inputs to display structures

CATEGORIES:

vector nodes, character nodes, label nodes

DESCRIPTION

The vector, character, and label data structures can be updated via inputs to their node in the display structure. Following are examples of updates. For complete documentation on inputs for vector, character, and label nodes, refer to Section *RM1*, *Command Summary*.

ANALYSIS AND IMPLEMENTATION:

The following program uses an update to delete the top vector of a box. The third item in the vector list is changed from a move command to a position command.

```
vector := vec n=4
               0,0,0
               0,.5,0
               .5,.5,0
               .5,0,0
               0,0,0;

 display vector;
 send false to <3>vector;
```

The following program uses an update to replace the last character (g) of a string with another character (a).

```
char1 := char 'test string';
 scal := scale by .05,..05 applied to char1;
 display scal;
 send 'a' to <last>char1;
```

The following programming example uses an update to replace the last string in a label block (string1) with another string (new string).

```
lab := labels 0,0,0 ´string 1´
                 0,-1.5,0 ´string 2´
                 0,-3,0 ´string 3´;
 scal := scale by .05,..05 applied to lab;
 display scal;
 send ´new string´ to <3>lab;
```

FURTHER INFORMATION:

RM1, Command Summary

12. How to do Run Length Encoded Programming

CATEGORIES:

raster programming

DESCRIPTION

The PS 390 raster system outputs static images to a 1024 (column) by 864 (row) pixel raster display. Each pixel is 24 bits deep for addressing into a red–green–blue color lookup table that is 24 bits deep. The PS 390 accepts raster data from the host in run–length encoded format. A description of this data format and how to address the various pixels on the PS 390 is contained in *Section GT14, PS 390 Raster Programming*. The following is an example from *GT14*. The program displays a flag of red, white, and blue blocks.

ANALYSIS AND IMPLEMENTATION:

1. Load a black background color. This is done through the PRasEr call with the red, green, and blue entries having a value of 0.
2. Display a red rectangle. This rectangle will be 200 pixels wide by 440 pixels high. The block is located between pixel number 20 and 219 in the X direction and pixel number 20 and 459 in the Y direction. The location of the block is given in the PRasLd call. The number of pixels and the color of these pixels are stored in the matrix MAT. This matrix is used in the PRasWP call for writing down pixel information.
3. Display a white rectangle. This rectangle will be 200 pixels wide by 440 pixels high. The block is located between pixel number 220 and 419 in the X direction and pixel number 20 and 459 in the Y direction. The location of the block is given in the PRasLd call. The number of pixels and the color of these pixels are stored in the matrix MAT. This matrix is used in the PRasWP call for writing down pixel information.

4. Display a blue rectangle. This rectangle will be 200 pixels wide by 440 pixels high. The block is located between pixel number 420 and 619 in the X direction and pixel number 20 and 459 in the Y direction. The location of the block is given in the PRasLd call. The number of pixels and the color of these pixels are stored in the matrix MAT. This matrix is used in the PRasWP call for writing down pixel information.

5. Exit program.

FURTHER INFORMATION:

GT14, Raster Programming

13. How to Define a Break Key

CATEGORIES:

break key

DESCRIPTION:

A break sequence is useful for getting the attention of the host across an asynchronous line for such purposes as requesting a logon prompt. PS 390 users have control over which key is defined as the break key, as well as the duration of the break sequence.

Any key may be used as a break key except the following:

- SETUP
- Function Key F1
- GRAPH Key
- TERM Key
- LINE/LOCAL Key

Any other function key, HARDCOPY, CLEAR/HOME, or any key on the right-hand keypad can be designated as the break key. The break key can be designated as the key, the shift value of the key, or the control value of the key.

ANALYSIS AND IMPLEMENTATION:

Defining the break key

You must be in setup mode to define the break key. To define a break key, perform the following:

- Press Function Key F10
- Press the key your designating as the break key
- Press Function Key F1 to indicate the break key has been selected

After entering this sequence press the setup key to exit setup mode.

Setting the duration of the break sequence

Use the SETUP INTERFACE command to set the duration of the break sequence. For example, to set the break time on port 1 to be 50 centiseconds, enter the following command:

```
SETUP INTERFACE PORT10/BREAK_TIME=50;
```

The default break time is 10 centiseconds and the maximum is 127. All values are in centiseconds.

NOTE

> The break key is only functional in the terminal emulator mode of operation.

14. How to Debug a Function Network

CATEGORIES:

DESCRIPTION:

There are two ways to debug the function network; use the PS 390 Debug, or tap into the function network in different places and examine the data to see if it is correct at that point.

ANALYSIS AND IMPLEMENTATION:

PS 390 Debug

It is suggested that you do not attempt to use Debug except when other methods for debugging a function network have failed. To use Debug, you must be familiar with PS 390 data types and formats. For more information on the PS 390 debugger, refer to *AP7.*

Tapping into the Function Network

Print function

The F:PRINT function converts any data type to string format; it performs an inverse of the operation that occurs when an ASCII string is input to the PS 390 and is converted to one of the data types.

Most of the data retrieved from a function network must first be translated by the F:PRINT function to an ASCII string so that it can be printed in a readable form.

Unprintable to printable data

The F:NPRT_PRT function converts strings containing nonprintable characters to strings with printable characters, as in:

```
^L to <FF>
```

If the F:NPRT_PRT input is connected to the function receiving input from the host, and the F:NPRT_PRT output connected to the terminal emulator, it allows all data that enters the PS 390 from the host to be printable.

Debug port: O3$

If you have a debug terminal, you can dump function network data to the terminal by making a connection (in configure mode) to O3$.

NOTE

This is the letter O, not the number zero.

O3$ is a system function that corresponds to output port 3 on your PS 390. Port 3 is initially configured to be a debug port, as follows:

- 9600 baud
- 8 bits per character
- 1 stop bit
- No parity
- Nontransparent mode that accepts all X_ON and X_off protocol characters
- 8 48–byte buffers with 0 STOP buffers and 1 GO buffer
- Debug break enabled

This configuration may be changed by using the SETUP INTERFACE command.

For example, if your transformed data network is not returning the expected messages to your host program from HOST_MESSAGE, look at the data that is returned from the LIST function (the LIST function sends the data to the HOST_MESSAGE function). Use the following commands to look at the list function data on the debug terminal.

```
@@ CONFIGURE A;
@@ CONN LIST1<1>:<1>O3$;
@@ FINISH CONFIGURATION;
```

Note that F:LIST is a special function that converts the output of F:XFORMDATA into a PS 390 ASCII command string suitable for storage on the host computer. The data output by F:LIST does not need to be sent through F:PRINT.

ES_TE and MESSAGE_DISPLAY initial function instances

ES_TE is the terminal emulator display handler that displays the input on the PS 390 screen. MESSAGE_DISPLAY is a function that corresponds to the bottom line of the PS 390 display and is used to display error messages and information messages. Both of these functions are user-accessible and may be connected to any printable output of any function.

ES_TE uses the next available line on the screen and then scrolls this line as does a terminal emulator. However, MESSAGE_DISPLAY is restricted to one line, so output to this function overrides what is already on the bottom line of the display.

The following are possible networks for debugging your function network:

```
PS390 printable data --> F:PRINT--> ES_TE, MESSAGE_DISPLAY, or O3$

PS390 non-printable data--> F:NPRT_PRT--> ES_TE, MESSAGE_DISPLAY, or O3$

ASCII data ------------> ES_TE, MESSAGE_DISPLAY, or O3$
```

FURTHER INFORMATION:

RM1, Command Summary

Helpful Hint Topic 2, How to deal with configure mode and naming conventions.

AP7, Advanced Programming

15. Intensity Settings on the PS 390

CATEGORIES:

SET INTENSITY node

DESCRIPTION:

When you display an object consisting of many lines drawn closely together on the PS 390 you occasionally observe a "roping effect" on the lines. The roping effect gives lines a jagged appearance similar to that of a braided rope. The roping effect is due to the intensity saturation of pixels as lines are drawn very close to each other. You can restore the line crispness by lowering the overall intensity level of the display; however, this results in a trade off between crisp lines and bright intensity.

To minimize the effect of lower display intensity, you can build flexible intensity settings into the display structure. Then, if the roping effect occurs, you can decrease the intensity in stages until crispness is achieved or return it to full brightness when desired.

ANALYSIS AND IMPLEMENTATION:

To create a flexible display intensity, place a SET INTENSITY node at the very top of the display structure and use input from a function key or dial to cycle through several levels of intensity values.

The following PS 390 commands could be added to a PS 390 program to cycle through five levels of intensity using the F11 function key. The number of intensity levels and the actual intensity range values should be modified to meet your requirements.

NOTE

This example is not intended as a complete program and will not run independently. For demonstration purposes, it assumes your top level display structure is named IMAGE and that it includes an enabled SET INTENSITY node named INTENS at the top. Each time the F11 key is pushed, the intensity range of the IMAGE.INTENS node is modified and the intensity is cycled to the next level.

```
{ Instance and connect the necessary functions. INTENS_SELECT and    }
{ INTENS_LABEL will hold cycling queues of maximum intensity values  }
{ to be sent on input from the F11 key. INTENS_RANGE will send a 2-D }
{ intensity range vector of ( 0.0, current maximum ) to the          }
{ IMAGE.INTENS node.                                                 }

INTENS_SELECT := F:SYNC(2);
INTENS_LABEL  := F:SYNC(2);
INTENS_RANGE := F:CVEC;
KEY_ROUTE := F:ROUTEC(12);

CONN FKEYS <1>:<1> KEY_ROUTE;
CONN KEY_ROUTE <11>:<1> INTENS_SELECT;
CONN INTENS_SELECT <2>:<2> INTENS_RANGE;
CONN INTENS_RANGE <1>:<2> IMAGE.INTENS;

{ Set maximum intensity values on a cycling queue. }
SEND .15 TO <2> INTENS_SELECT;
SEND .25 TO <2> INTENS_SELECT;
SEND .4 TO <2> INTENS_SELECT;
SEND .7 TO <2> INTENS_SELECT;
SEND 1  TO <2> INTENS_SELECT;
CONN INTENS_SELECT <2>:<2> INTENS_SELECT;

{ Label the F11 key with the current maximum intensity. }
CONN KEY_ROUTE <11>:<1> INTENS_LABEL;
CONN INTENS_LABEL <2>:<1> FLABEL11;
SEND ´INT=.15´ TO <2> INTENS_LABEL;
SEND ´INT=.25´  TO <2> INTENS_LABEL;
SEND ´INT=.4´  TO <2> INTENS_LABEL;
SEND ´INT=.7´  TO <2> INTENS_LABEL;
SEND ´INT=1´   TO <2> INTENS_LABEL;
CONN INTENS_LABEL <2>:<2> INTENS_LABEL;
SEND ´INT_SEL´ TO <1> FLABEL11;

{ Set minimum intensity to a constant 0. }
SEND 0 TO <1> INTENS_RANGE;

{ Accept F11 key output }
SEND TRUE TO <2> KEY_ROUTE;
```

16. Softlabels

CATEGORIES:

softlabels, function key labels, dial labels

DESCRIPTION:

The softlabels network redefines the dynamic viewport on the PS 390 to allow the left edge of the display to be used as a display area for the function key labels (FLABELS) and dial labels (DLABELS).

ANALYSIS AND IMPLEMENTATION:

The network is loaded from diskette B with the following command:

```
@@ SEND 'SLABEL' TO <1>READASCII;
```

This command can be added the the SITE.DAT file so that the network is automatically loaded at boot time, as follows:

```
SEND 'SLABEL' TO <1>READASCII1;
```

There is also a copy of the softlabel network on the host distribution tape under the the PS 390 subdirectory.

The softlabels are accessed by sending a character string to <1> of FLABEL 1-12 as well as <1> of DLABEL 1-8. FLABEL 0 is not supported by the softlabel network.

The softlabels network allows for two lines of descriptive characters for each label block. The second line is accessed by sending a character string to <1> of the FLABEL 1-12h or DLABEL 1-8h functions.

An INITIALIZE command clears the displayed labels. The underlying network is protected from the initialize command.

You can toggle the display of the labels on and off by pressing the LEFT ARROW key when then the system is in application mode.

17. CPK Rendering

CATEGORIES:

Rendering

DESCRIPTION:

Enhanced CPK firmware includes new shading and lighting options for CPK renderings and the integration of lines as a raster primitive.

With this release, you can now specify multiple, colored, and movable light sources. The color of each light source may be specified by the parameters hue, saturation, and intensity. The direction of the light source may be specified as any 3D vector. Previous to this release, you were only allowed to define one light source, which was fixed at the eyepoint and assumed to be white.

The specification of diffuse and specular attributes for spheres is also a new capability now available with enhanced CPK firmware. These two attributes can be adjusted to produce the appearance of different types of surfaces. A high diffuse value and a low specular value produces a dull, plastic appearance. A low diffuse value and a high specular value produces a shiny, metallic appearance.

As before, you use an attribute table with color and radius for each atom in your CPK rendering. A default attribute table is provided with the firmware. You are also allowed to define your own attribute table.

ANALYSIS AND IMPLEMENTATION:

Spherical rendering and raster lines are represented as vector lists instead of an explicit PS 390 data type. Spheres are shaded consistent with the Phong shading style, allowing multiple colored light sources, specular reflections, and depth cueing. Hidden-element removal has been accomplished with a common z-buffer algorithm.

17.1 DEFINING SPHERES AND LINES

A table for rendering attributes for 127 atom types is referenced by the rendering node. The seven-bit intensity field or the TABulated field for a vector is used as an index into the attribute table. Thus, you must store an appropriate value in the field depending on which atom type a particular vector represents. Either the intensity or the TABulated option may be used to specify this field in the vector.

The ITEMized option in the vector list command allows specifying this intensity field as a real number between 0 and 1. For example, to specify that an atom is to have index 12, you would need to divide 12 by 128 (0.09375) and use the result in the "i = " clause of the vector list command:

```
v := vec ITEMized n = 1
0,0,0 i = 0.09375;    { for atom type 12 (12 / 128) }
```

The TABulated option in the VECtor_list command allows specifying this seven bit value as an integer between 0 and 126:

```
v := vec TABulated n = 1
0,0,0 t = 12;    { for atom type 12 }
```

Both of the above examples create the same vector list and either option may be used in conjunction with CPK renderings. If you have been using the ITEMized option, there is no need to switch to the tabulated option.

These vector lists must be tied to F:XFORMDATA functions whose output is used by the PS 390 SOLID_RENDERING node to perform CPK renderings. Previous CPK firmware used the F:CPK function to perform renderings. The SOLID_RENDERING node, which was previously used only for rendering polygons, is now also used to render raster spheres and lines. (Refer to section 17.3 for more details on the SOLID_RENDERING node.)

NOTE

> Since there are no explicit PS 390 data types for representing spheres or raster lines, you do not place sphere or raster-line data under a rendering operation node.

17.2 Specifying Attributes for Spheres and Lines

The attributes for spheres (radius, diffuse, specular, color) and lines (color) are stored in a default table created at system boot up. This table can be modified via input <14> of the SHADINGENVIRONMENT function.

The table has the following components:

Hue Saturation Intensity Radius Diffuse Specular

Hue is a real number in the range 0 to 360. Saturation and intensity are real numbers in the range 0 to 1. Radius is a real number greater than 0. Diffuse is a real number in the range 0 to 1. Specular is an integer in the range 0 to 255. The table is initialized as follows:

INDEX	Hue	Sat	Intensity	Radius	Diffuse	Specular
0	0	0	0.5	1.8	0.7	4 (Gray)
1	0	0	1	1.2	0.7	4 (White)
2	120	1	1	1.35	0.7	4 (Red)
3	240	1	1	1.8	0.7	4 (Green)
4	0	1	1	1.8	0.7	4 (Blue)
5	180	1	1	1.7	0.7	4 (Yellow)
6	0	0	0.7	1.8	0.7	4 (Gray)
7	300	1	1	2.15	0.7	4 (Cyan)
8	60	1	1	1.8	0.7	4 (Magenta)

Spheres use all six components per entry. Lines use only the hue, saturation and intensity components.

The t field of each 3D tabulated vector is used as an index into this table. The table contains 127 entries (0–126). Entries 9–126 are not initialized.

For example, the following vector list represents three spheres with the color indicated.

```
Sphere:= VECtor_list TABulated N = 3
    P 1.866,1.5,0    t = 5  {yellow sphere}
    L 1.787,2.833,0  t = 6  {gray sphere}
    L .822,3.282,0   t = 7  {cyan sphere}
  ;
```

The following example represents a square with sides of the indicated colors.

```
Rasterline := VECtor_list TABulated N = 5
   P 0,1,0
   L 0,0,0  t = 5  {yellow}
   L 1,0,0  t = 2  {red}
   L 1,1,0  t = 3  {green}
   L 0,1,0  t = 4  {blue}
  ;
```

NOTE

> Lines use the tabulated index of the point drawn "to" and not the point drawn "from."

The attribute table may be updated by encoding the table entries into a PS 390 tabulated vector list and then sending the name of the vector list to input<14> of SHADING ENVIRONMENT. The six table components are encoded into two consecutive 3D vectors of the vector list. Hue, saturation, and intensity are encoded into the first X,Y,Z respectively. Radius, diffuse, and specular are encoded into the second X,Y,Z respectively. The table index is encoded into the t field of the second vector. When an entry is updated, each of the six components must be specified. For example, the following vector list could be used to update attribute table entry 5:

```
ATTRIBUTE_TABLE := VEC TAB N = 2
 150,0.5,1          { Hue, Saturation, Intensity }
 5.0,0.3,2  t = 5   { Radius, Diffuse, Specular, Index }
;
```

Updating would be accomplished by the command:

```
@@ SEND 'ATTRIBUTE_TABLE' TO <14>SHADINGENVIRONMENT;
```

More than one table entry may be encoded into a vector list. The following vector list would be used to update attribute table entries 5, 6, and 7:

```
ATTRIBUTE_TABLE := VEC TAB N = 6
 0,.1,.1   2.0,0.5,4  t = 5
 120,1,1   4.0,0.8,9  t = 6
 240,1,1   3.0,0.3,2  t = 7
;
```

17.3 Rendering Spheres and Lines

The rendering node created with the PS 390 SOLID_RENDERING command is used to perform the rendering of raster spheres and lines. The rendering node has five inputs and acts somewhat like a PS 390 function.

Input <1> accepts a fix(7) to trigger the rendering. Input <2> is only used for polygons and has a default value which is adequate for raster spheres and lines. Input <3> accepts a transformed vector list and interprets the vectors as "moves" and "draws" for raster-line renderings. Similarly, input <4> of the rendering node accepts a transformed vector list and interprets each vector as an X,Y,Z spherical center for raster rendering. Input <5> of the rendering node accepts the name of the original vector list representing the spherical data to be rendered. The rendering node must have access to the original data to enable accurate scaling of the sphere(s). The name is represented as a string in single quotes.

17.4 Function Network Considerations

A function network to display a CPK rendering must contain at least the following:

- Instance of F:XFORMDATA to transform the data
- Instance of SOLID_RENDERING
- Instance of F:SYNC(n) to guarantee that the rendering node is not triggered until all the constant inputs of the rendering node are updated.

In some situations, an instance of F:XFNORM or F:CONCATXDATA(n) is required.

If a non-cubical window is used, an instance of F:XFNORM is required to normalize the coordinates. F:XFNORM is a user-written function described in section 17.9.

If multiple instances of F:XFORMDATA are required for rendering either spheres or lines, an instance of F:CONCATXDATA(n) must be used to create a single vector list for spheres or a single vector list for lines.

The function network for CPK renderings must also accommodate potential timing problems triggering the rendering node and certain window and viewing restrictions. These function network considerations are detailed in the following sections.

The diagram in Figure 17-1 shows a general flow of data through a function network to the rendering node.

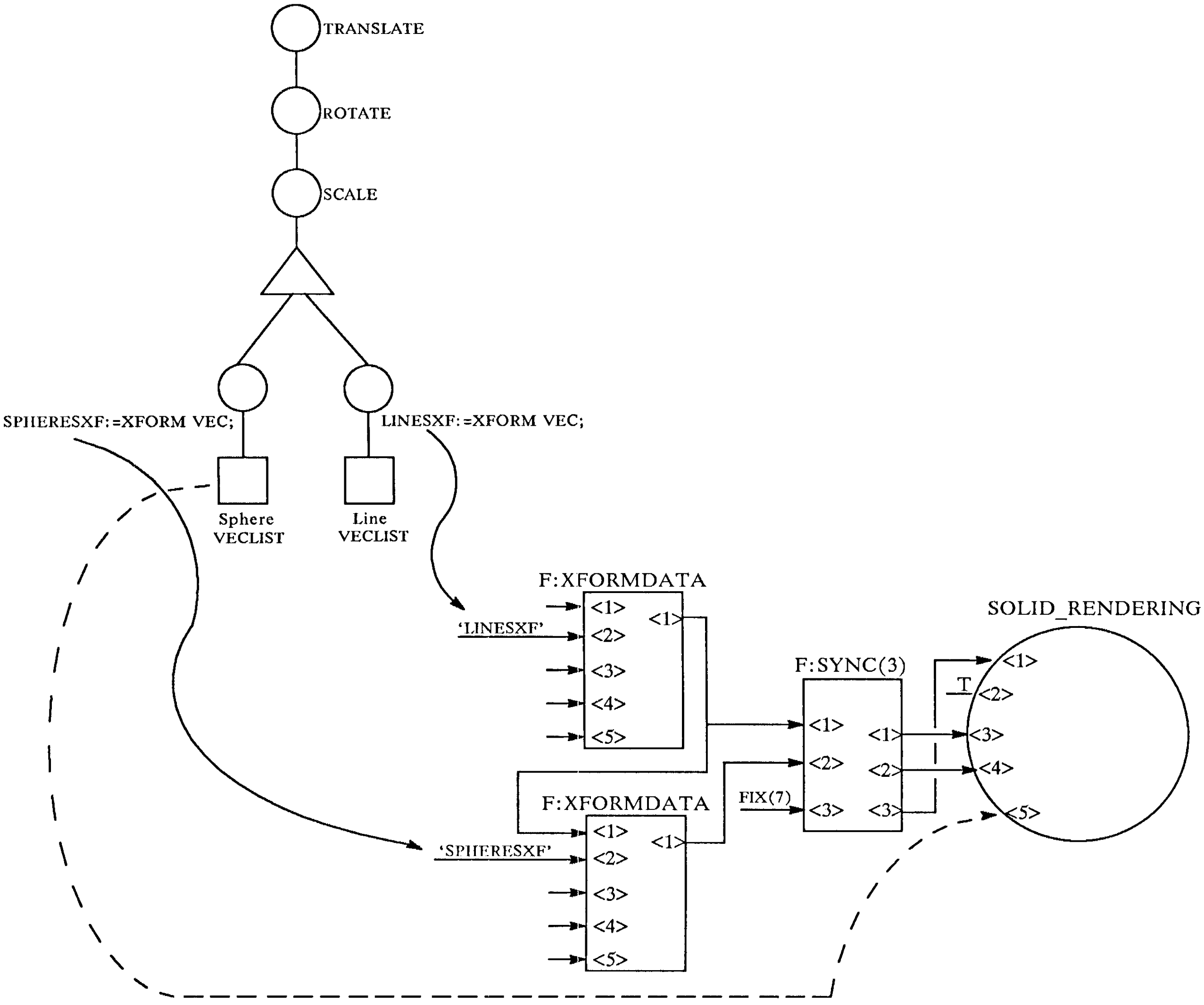


Figure 17-1. Function Network Diagram

17.5 Triggering the Rendering Node

Two potential timing problems exist with triggering the rendering node. Input <1> of the rendering node is the only active input. Inputs <3> and <4> accept transformed data to render lines and spheres. Since inputs <3> and <4> are constant inputs, you must guarantee that they have been updated before the trigger is sent to input <1> of the rendering node. This is accomplished using the F:SYNC(n) function.

The second potential timing problem deals with the triggering of F:XFORMDATA. An instance of F:XFORMDATA must not be triggered while it or any other instance of F:XFORMDATA is still active. Thus, when using multiple instances of F:XFORMDATA, one instance should be used to trigger the next. This is explained in more detail in section 17.7.

17.6 Notes on Using F:CONCATXDATA(n)

F:CONCATXDATA(n) accepts up to 127 transformed vector lists (output from XFORMDATA functions) and concatenates them into a single transformed vector list. This function is used to avoid the maximum vector restriction imposed on the output of F:XFORMDATA. The XFORMDATA function will return a maximum of 2048 vectors. This restriction passes on to the rendering node since the output of the XFORMDATA function is normally connected directly to the rendering node. To obtain a rendering of greater than 2048 vectors or (spheres), the output of multiple instances of XFORMDATA must be concatenated into a single transformed vector list, which can then be sent to the rendering node (see Figure 17-2). Note that the number of inputs to an instance of F:CONCATXDATA is specified in the parameter (n) when the function is instanced.

17.7 Notes on Using F:XFORMDATA

As previously mentioned, when multiple instances of F:XFORMDATA are used to provide input for F:CONCATXDATA(n), they must be connected in a way which ensures that one instance completes before the next one commences. This synchronization is accomplished by linking instances of XFORMDATA together so that the output of the first instance triggers the second instance, and the output of the second instance triggers the third, and so forth. For example, assume that in the following network, the vector list SPHERES contains 5,000 vectors.

```
FORCPK := BEGIN_STRUCTURE
           GETXF := XFORM VEC;
           INSTANCE OF SPHERES;
          END_STRUCTURE;
```

One instance of XFORMDATA could retrieve the first 2048 transformed vectors of SPHERES (vectors 1–2048). A second instance of XFORMDATA could retrieve the second 2048 transformed vectors (vectors 2049 - 4096). A third instance of XFORMDATA could retrieve the last 904 vectors (vectors 4097 - 5000). Figure 17-2 shows an illustration of this network.

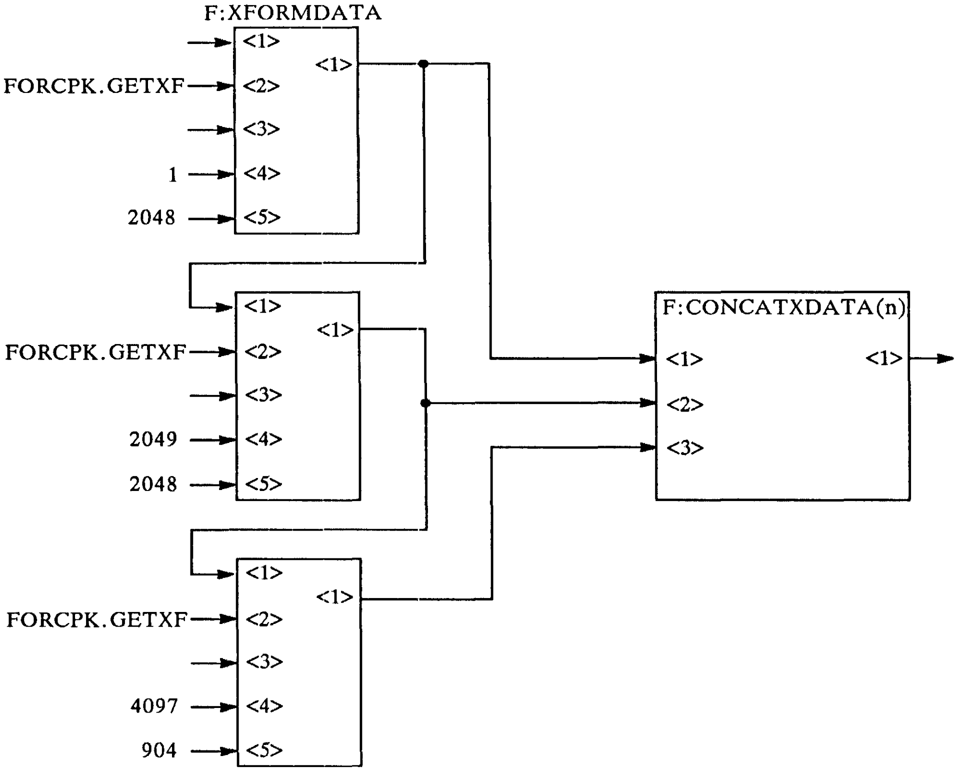


Figure 17-2. Synchronization of XFORMDATA

There is one other restriction that you must be aware of when using F:XFORMDATA. Input <3> of F:XFORMDATA typically allows you to specify a name for the transformed data. However, when using F:XFORMDATA in conjunction with the rendering node, this input MUST be left blank.

CAUTION

> Naming the transformed data and then sending it to a rendering node, will result in a system failure.

17.8 Window and Viewing Restrictions

The following window and viewing restrictions apply to CPK renderings.

The rendering node assumes that the transformed data it receives on input <4> comes from an orthographic projection (i.e., PS 390 WINDOW command). Each sphere is rendered with the radius specified in the attribute table, regardless of the sphere's distance from the viewpoint.

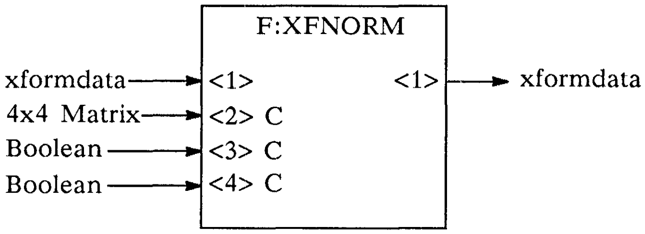
Spherical renderings with perspective projections (FIELD_OF_VIEW or EYE BACK) should not be used. Non-cubical windows for spheres are allowed but require special handling. If non-cubical windows are allowed by the applications program, F:XFNORM must be used to ensure correct mapping of the coordinates. This new function is explained in the following section.

17.9 Using F:XFNORM

This function filters, maps, and clips transformed sphere and line data for enhanced CPK renderings. If the PS 390 WINDOW in effect for a rendering is non-cubical, transformed data will return non-uniformly mapped coordinates which result in incorrect renderings. The F:XFNORM function compensates for this non-uniform mapping, by applying an inverse mapping.

Spheres which are outside of the Z-clipping planes are rejected before mapping. Under user control, the function either rejects or clips lines which have exactly one endpoint outside of the Z-clipping planes before mapping.

Following is a summary of F:XFNORM.



Input <1> accepts transformed data.

Input <2> accepts the output of F:WINDOW. The WINDOW function used to update the WINDOW node in the display structure should be connected to input <2> of F:XFNORM.

Input <3> accepts a Boolean value indicating whether the transformed vector list is to be used for rendering spheres or for rendering lines. A True indicates that the vector list is to be used for spheres. A FALSE indicates that the vector list is to be used for lines.

Input <4> only pertains .o transformed vector lists used for rendering lines (i.e. when input <3> is FALSE). Input <4> is ignored for vector lists used for rendering spheres, but must have a Boolean value in order for the function to trigger. Input <4> accepts a Boolean value indicating whether lines with exactly one endpoint outside of the Z clipping planes are to be clipped or rejected entirely.

17.10 Mapping

F:XFNORM first compares the WINDOW's *dx* against its *dy*. If *dx* is not equal to *dy*, then the window is scaled down in the appropriate dimension to correct the non-uniform transformation. Then if *dz* is not equal to the new normalized *dx/dy* value, the WINDOW is scaled in *Z* to correct it in that dimension. In doing so, some possible problems are introduced.

In some instances, when F:XFNORM scales up and transforms the data, it places some points out of the clipping planes which were previously between the clipping planes. Because of this, certain spheres and lines may be clipped-out in the rendering which ARE NOT Z-clipped in the wireframe model, even with depth clipping enabled. There is little that can be done about this.

Similarly, in other situations, .XFNORM scales down and transforms a point into the range between clipping planes which was previously outside of the clipping planes. Because of this, certain spheres and lines may be rendered which are Z-clipped in the wireframe display of the model. This problem can be solved, however. For spheres, those points which are Z-clipped (before mapping) are tagged, and are not rendered.

One additional problem exists. Assume that a model would "fit" in a cubical window but is being viewed in a window with Z-planes that are scaled out some distance. Since there is a non-uniform WINDOW mapping in Z, F:XFNORM will compensate in this case by scaling up the vectors' Z-components. Assume also, that the model is near either the front or back clipping plane. Since the Z-components of the model's vectors are already near a clipping plane, scaling them up will place them outside of the clipping plane and nothing will be rendered. To reduce this problem, the function also centers the model in Z. Figure 17.3 shows a function network using F:XFNORM.

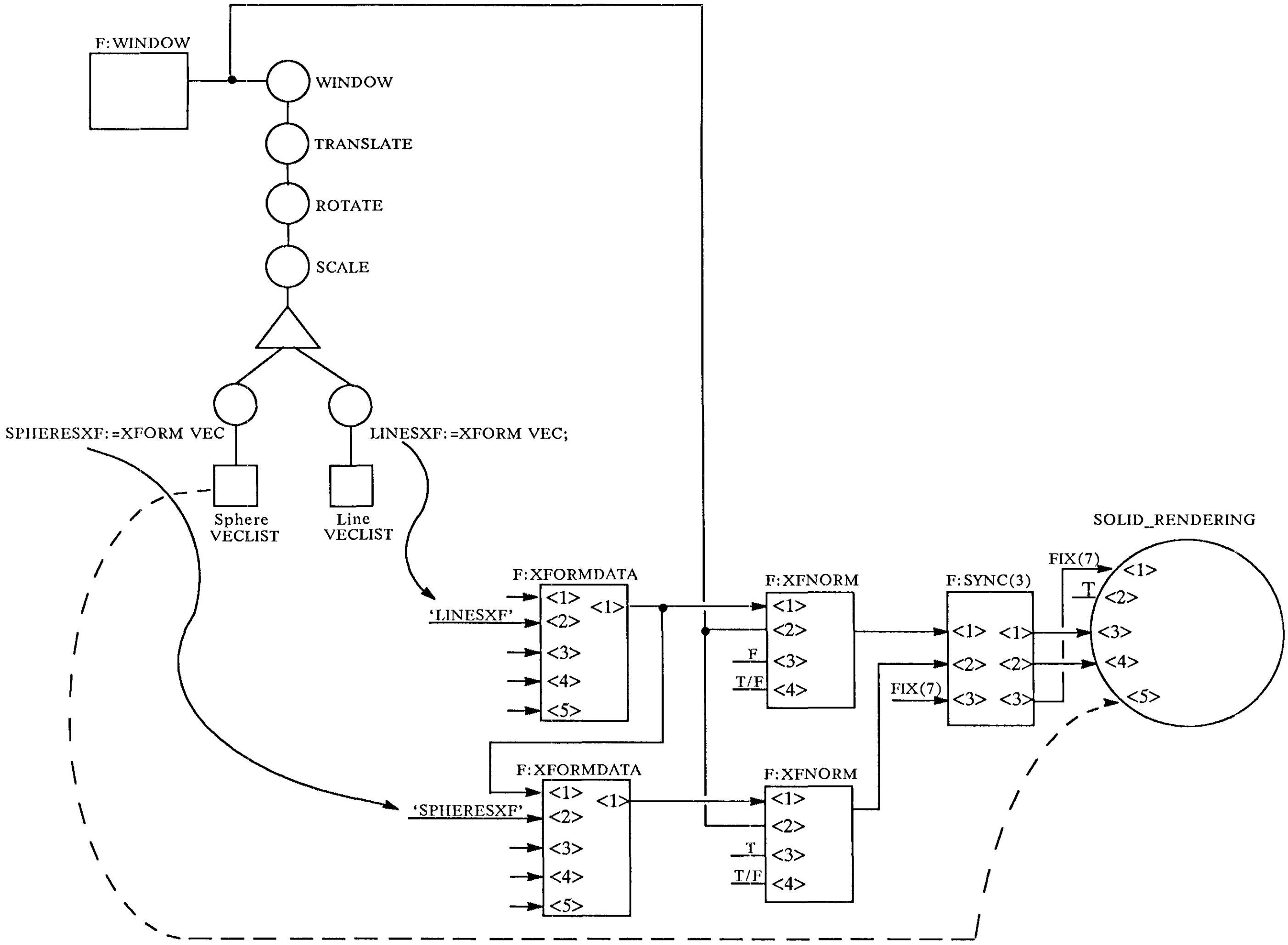


Figure 17-3. F:XFNORM Network for Window Scaling

17.11 Viewport Considerations

On the PS 340, only the following raster viewport should be used.

```
Xleft = 64        Xright = 575
Ybottom = -32     Ytop = 479
```

The command "SEND V3D(64,-32,511) TO <3>SHADINGENVIRONMENT;" will set this raster viewport.

On the 390, the following viewport for CPK renderings must be used.

```
    Xleft = 256      Xright = 767
    Ybottom = 176    Ytop = 687
```

The command "SEND V3D(256,176,511) TO <3>SHADINGENVIRONMENT;" will set this raster viewport.

17.12 Using CPK on the PS 390

The PS 390 cannot display vector-normalized data, so by default, the system converts it to block-normalized data. Enhanced CPK firmware is dependent on vector-normalized data to perform renderings, so some programming adjustments have to be made to account for this.

An intrinsic user function, F:ALLOW_VECNORM allows vector-normalized vector lists to be created locally or downloaded from the host to the PS 390. Function networks for the display of CPK renderings need to have a node for both the block-normalized vector (to allow display of the data on the PS 390) and a vector- normalized vector list of the same data (required by the XFORMDATA function). The function network in figure 17-4 gives an example of this.

Following is a summary of F:ALLOW_VECNORM.

A Boolean TRUE sent to input <1> of F:ALLOW_VECNORM allows vector-normalized data to be created by the PS 390. A Boolean FALSE on input <1> will reset the PS 390 and cause vector-normalized data to be converted to block-normalized data. The Boolean TRUE sent from output <1> of F:ALLOW_VECNORM when the function has run to completion may be connected to user function networks.

Because F:XFORMDATA cannot selectively choose only vector-normalized vector lists, any block-normalized vector lists that are sent to the rendering node unintentionally will be discarded.

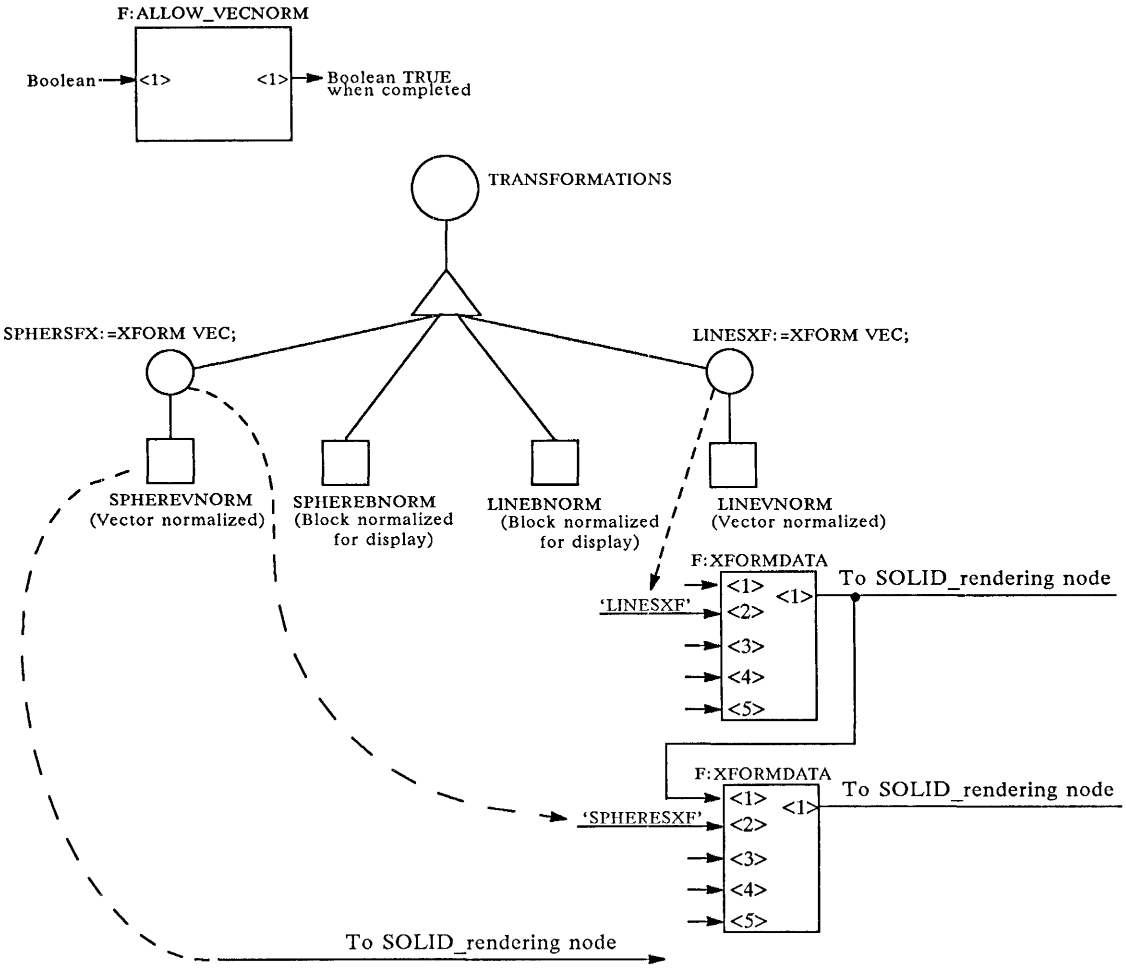


Figure 17-4. Function Network for PS 390 Display and CPK Renderings

There is a potential timing problem when using the F:ALLOW_VECNORM function: if the Boolean TRUE to trigger F:ALLOW_VECNORM is sent from the host and then immediately followed by a vector list to be defined as a vector-normalized vector list, F:ALLOW_VECNORM may not have executed before the vector list definition begins. If this should happen, the resulting vector list would be the wrong type. To prevent this from occurring, the command GIVE_UP_CPU; may be used. GIVE_UP_CPU causes the command interpreter to terminate execution temporarily, allowing other functions to be activated. To ensure that other functions are activated, GIVE_UP_CPU should be sent four times after sending a value to F:ALLOW_VECNORM.

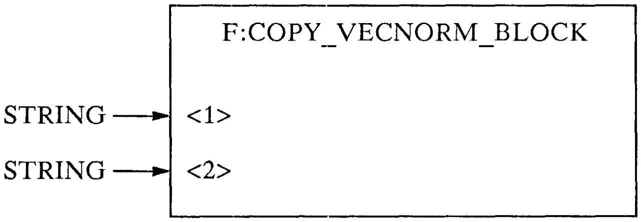
The sequence that should normally be followed when downloading vectors lists to the PS 390 is:

1. Download the vector lists from the host to be displayed (block-normalized vector lists).
2. Send Boolean TRUE to input <1> of an instance of F:ALLOW_VECNORM.
3. Execute the GIVE_UP_CPU command four times.
4. Download the vector lists from the host for CPK renderings (PS 390 will create vector-normalized vector lists).
5. Send Boolean FALSE to input <1> of the same instance of F:ALLOW_VECNORM (resets the default condition of converting to block-normalized data).
6. Execute the GIVE_UP_CPU command four times to make sure F:ALLOW_VECNORM is reset to the default condition before any more vector lists are sent.

NOTE

> Although vector-normalized lists will not be displayed on the PS 390, they can still be transformed using F:XFORMDATA and sent to the rendering node without any additional modification.

17.13 Using F:COPY_VECNORM_BLOCK



The user-written function F:COPY_VECNORM_BLOCK accepts the name of an existing vector-normalized vector list on input <1> and creates the corresponding block-normalized vector list with the name specified on input <2>.

This function is to be used in conjunction with the intrinsic user function F:ALLOW_VECNORM for enhanced CPK renderings on the PS 390. CPK renderings require the use of vector-normalized vector lists, but the PS 390 will only display block-normalized vector lists. To avoid having to transmit both types of vector lists from the host, the user may transmit only the vector-normalized list and use F:COPY_VECNORM_BLOCK to locally generate a block-normalized vector list.

17.14 MASS MEMORY REQUIREMENT

The RESERVE_working_storage command must be issued at least once to enable enhanced CPK renderings. This command reserves a block of mass memory to perform the rendering. (Refer to the RESERVE_working_storage command summary at the back of this manual for more details.) For polygonal renderings, this number is typically large, at least 100000 bytes. For CPK or ball-and-stick renderings 1000 bytes is sufficient.

FURTHER INFORMATION:

RM1, Command Summary
RM2, Intrinsic Functions
RM3, Initial Function Instances

TT3. USING THE GSRs

TT3. USING THE GSRS

CONTENTS

Section TT3
Using The Graphics Support Routines

The Graphics Support Routines (GSRs) are a set of host resident software routines that are the standard vehicle for communication to the PS 390 from the host. They are a collection of FORTRAN, Pascal or UNIX/C routines that preparse and package data on the host computer. Typically, the routines are used for the following applications:

- Attach to the graphics device
- Create and modify display structures
- Create, connect and modify function networks
- Receive data from the graphics device

This section is a guide to the FORTRAN, Pascal and UNIX/C GSRs. It contains information on the conventions and definitions used in the GSRs. Section *RM4* contains the FORTRAN, Pascal and UNIX/C GSRs listed in alphabetical order according to the FORTRAN GSR. The GSRs corresponding to a PS 390 command are grouped together. The GSRs are listed in the following order: VAX and IBM FORTRAN, VAX Pascal, IBM Pascal and UNIX/C. A description of the GSR, the PS 390 command syntax and cross references follow the listing of the GSRs.

Section *RM4* also contains error tables which define the possible error codes used to identify warning, error or fatal error conditions that may arise while using the GSRs.

1. VAX and IBM FORTRAN Graphics Support Routines

The PS 390 VAX FORTRAN GSRs are written in FORTRAN-77 and require a FORTRAN-77 compiler. The GSRs are supported under PS 390 Graphics Firmware Release A2.V02 and higher. There are no specific hardware requirements.

The PS 390 IBM FORTRAN GSRs are written in VS FORTRAN and are compatible with IBM VM/CMS and TSO environments. The GSRs are supported under PS 390 Graphics Firmware Release A2.V02 and higher. There are no specific hardware requirements.

The UNIX/C routines are written in C and are supported under PS 390 Graphics Firmware Release A2.V02 and higher and UNIX BSD 4.2 running on a DEC VAX host system.

Appendix A contains a FORTRAN--77 network creation example program, and *Appendix B* contains a VS FORTRAN network creation example program. Both programs contain an error handling routine.

1.1 FORTRAN GSR Conventions

The FORTRAN-77 GSRs make extensive use of the following data type definitions:

```
Boolean = Logical value true/false, generally LOGICAL*1.
Integer = Integer value always INTEGER*4.
Real    = Real ( floating point ) number generally REAL*4.
String  = Character string, CHARACTER*N.
```

The VS FORTRAN GSRs make extensive use of the following data type definitions:

```
Boolean = Logical value true/false, generally LOGICAL*1.
Integer = Integer  value generally INTEGER*4.
Real = Real (floating point) number generally REAL*4.
String = Character string, CHARACTER*N.
```

For the FORTRAN version of the GSRs, character strings require a delimiter character for length determination. Double quote (") is the default delimiter. This delimiter may be changed using the PDELIM routine. A description of PDELIM is found in Section *RM4*. The GSRs use LEN (String) to determine the maximum length of a string. Therefore, if the delimiter is not specified, all characters up to LEN (String) will be used. Because of this, quoted strings may be used without delimiters, i.e. `THIS´ is treated the same as `THIS"´.

1.2 Utility and Application Routines

Utility Routines are specific to the operation of the GSRs. These routines are used to attach the PS 390, set the string delimiting character, select multiplexing channels, send and receive messages, and detach.

Application Routines correspond almost one for one with the standard PS 390 Commands. In most cases, the names for the application routines were derived by choosing an abbreviation of the PS 390 commands and prefixing it with a **P**. Parameter ordering generally coincides with the PS 390 commands as well. Examples of some of the application routines are shown below.

EXAMPLE 1

For commands which build operate display structures, such as

```
Name:= operate parameter1,parameter2,..., then apply;
```

the routine call is:

```
CALL Poper('name',parameter1,parameter2,...,'apply', ErrHnd)
```

where:

oper is an abbreviated form of the PS 390 command such as rotate in x — Protx

'name' is a character string containing the name to be associated with the operate

parameter1,parameter2,..., are the parameters to be used in computing the operation. These may be logicals, integers, reals, vectors, or matrices.

'apply' is a character string containing the name of the object to which this operate applies.

ErrHnd is the user-defined error handler routine.

EXAMPLE 2

For commands to send to functions or display structures, such as

```
Send datum to <input>dest;
```

the routine call is:

```
CALL PSNtyp(datum,input,'dest', ErrHnd)
```

where:

typ is an abbreviated form of the PS 390 command such as PSNFIX, PSNM2D,...

datum is what is to be sent. It may be logical, integer, real, character string, vector, or a REAL*4 two-dimensional array.

input is an integer which specifies which input of the destination is being sent to.

'dest' is a character string containing the name of the display structure or function.

ErrHnd is the user-defined error handler routine.

Note that the function names in the GSRs are specified without the "F:" prefix that is used in the standard PS 390 command language.

EXAMPLE 3

For commands which create functions and connections such as:

```
Name := f:genfcn;
Name := f:genfcn(n);
Conn name<output>:<input>dest;
Disc name<output>:<input>dest;
```

the routine calls are:

```
CALL PFN      ( 'name', 'genfcn', ErrHnd )
CALL PFNN     ( 'name', 'genfcn', n, ErrHnd )
CALL PCONN    ( 'name',output,input,'dest', ErrHnd )
CALL PDI      ( 'name',output,input,'dest', ErrHnd )
```

where:

'name' is a character string containing the name associated with the function instance.

'genfcn' is a character string containing the name of the system generic function.

n is an integer specifying the number of input/outputs for this function instance.

output,input are integers specifying the output and input numbers.

'dest' is a character string containing the name of the display structure or function.

ErrHnd is the user-defined error handler routine.

1.3 Exceptions

To be fully specified using GSRs, three PS 390 commands require three separate calls to routines. The commands are LABEL, VECTOR_LIST, and POLYGON.

For example, to create, specify and complete a label block, the user must call:

```
PLaBeg - To create and open a label block
PLaAdd - May be called multiple times to add to a previously
         opened label block
PLaEnd - To complete the creation of a label block.
```

Together these three routines implement the PS 390 command:

```
 Name   := LABELS  x, y, z, 'string'
                      .
                      .
                   x, y, z, 'string';
```

In the same way, the user must call PVcBeg to begin a vector list, PVcLis to send a piece of a vector list, and PVcEnd to end a vector list.

An example of a call that varies slightly from the PS 390 command is the PBSPL call. In the BSPLINE command, some of the parameters are optional. In the routine they are all required. This is also the case for the PRBSPL, PPOLY, and PRPOLY routines.

The PS 390 syntax allows for instancing multiple display entities and for creating multiple variables. In the PS 390 command language the commands would be:

```
NAME := INSTANCE a,b,c,d;
```

for instancing multiple display entities, and

```
VARIABLE s,y,z,w,t,q;
```

for multiple variables.

To perform the equivalent instancing of multiple display entities or for creating multiple variables, the following GSRs should be used.

For the multiple instance case:

```
CALL PINST(´NAME´, ´A´, ErrHnd)
CALL PINCL(´B´, ´NAME´, ErrHnd)
CALL PINCL(´C´, ´NAME´, ErrHnd)
CALL PINCL(´D´, ´NAME´, ERRHND)
```

For the multiple variable case:

```
CALL PVAR (´S´, ERRHND)
CALL PVAR (´Y´, ERRHND)
CALL PVAR (´Z´, ERRHND)
CALL PVAR (´W´, ERRHND)
CALL PVAR (´T´, ERRHND)
CALL PVAR (´Q´, ERRHND)
```

1.4 Error Handling

An error handling scheme has been employed to catch errors detected by the GSRs. Examples of errors detected by the GSRs are:

- Prefix not followed by an operate.
- Follow not followed by an operate.
- Multiple calls to PVcLis for block-normalized vector list data.
- Invalid characters in a name.

Command Interpreter errors and warnings are not detected by the GSRs. Examples of these errors are:

- Destination does not yet exist.
- Message rejected by destination.
- Connection not made.

Error checking will be performed within the GSRs to ensure that only valid characters are sent within names, and that routines are called in the proper order, in cases where order is required. No attempt has been made to capture errors and/or warnings from the Command Interpreter.

Each routine call includes an argument that specifies the user-written error handler. This error handler is of the form:

```
Routine ERRHND (ercode)
```

where **ercode** is an integer error code corresponding to one of the errors.

CAUTION

> It is critical that the user specify the error handler as EXTERNAL in all routines that make calls to the GSRs. Otherwise, the address of a real variable will be passed as a routine address and unpredictable results will occur if the error handler is called.

It is the responsibility of the user to provide an error handling routine to decide what action should be taken when an error is detected. The GSRs do not attempt to terminate execution or log errors.

The name, description, and error code of each detectable error is given in tables in Section *RM4*. An example error handler routine appears in the example programs in Appendix A and Appendix B. It is a sophisticated error handler that may be incorporated by the user into an error handling scheme, or used as an example of what an error handler should look like.

1.5 Programming Suggestions

The file PROCONST.FOR contains definitions for constants used by the FORTRAN–77 GSRs. The file PROCONSF FORTRAN contains definitions for constants used by the VS FORTRAN GSRs. It is often convenient to think of these constants by name rather than by remembering numbers. Specifically, in the usual PS 390 command syntax, inputs to display structures are often referred to by name such as <append> and <clear> for vector_lists and <position> and <step> for character strings. There are also <delete>, <last>, and others. Other useful constants such as values for conditional tests for level of detail, and vector list class are obtainable from PROCONST.FOR or PROCONSF FORTRAN. PROCONST.FOR or PROCONSF FORTRAN also contain a complete set of error/warning code definitions. These values are given in the error table in Section *RM4* and may be referenced by name by the user routine if PROCONST.FOR or PROCONSF FORTRAN is included in the routine.

The following is an abbreviated list derived from PROCONST.FOR or PROCONSF FORTRAN of the constants which should be most useful to the user.

GSR constant declarations:

Name	Meaning
PIAPP:	<Append> input number.
PIDEL:	<Delete> input number.
PICLR:	<Clear> input number.
PISTEP:	<Step> input number.
PIPOS:	<Position> input number.
PILAST:	<Last> input number.
PISUBS	<Substitute> input number.
PCLES:	"Less" level of detail comparison operator.
PCEQL:	"Equal" level of detail comparison operator.
PCLEQL:	"Less-equal" level of detail comparison operator.
PCGTR:	"Greater" level of detail comparison operator.
PCNEQL:	"Not-equal" level of detail comparison operator.
PCGEQL:	"Greater-equal" level of detail comparison operator.
PVCONN:	Vector list "Connected" class type.
PVDOTS:	Vector List "Dots" class type.
PVITEM:	Vector List "Itemized" class type.
PVSEPA:	Vector List "Separate" class type.

```
INTEGER*4
                     PIAPP,  PIDEL,  PICLR,
      &              PISTEP, PIPOS,  PILAST, PISUBS, PCLES,
      &              PCEQL,  PCLEQL, PCGTR,  PCNEQL, PCGEQL,
      &              PVCONN, PVDOTS, PVITEM, PVSEPA

PARAMETER
                     PIAPP =  0, PIDEL = -1,
      &              PICLR = -2, PISTEP= -3, PIPOS = -4, PILAST= -5,
      &              PISUBS = -6, PCLES = 0, PCEQL =  1, PCLEQL=  2,
      &              PCGTR =  3, PCNEQL=  4, PCGEQL=  5, PVCONN=  0,
      &              PVDOTS=  1, PVITEM=  2, PVSEPA=  3, PVTAB=  4,)
```

The following example illustrates the use of PROCONST.FOR.

Send to a vector list.

```
      PROGRAM TEST
      INCLUDE ´ PROCONST.FOR ´
      LOGICAL*1 PL (100)
      DIMENSION VECS( 4,100 ), AVEC( 3 )
      REAL*4  VECS, AVEC
C
C     Always declare user error handler external
C
      EXTERNAL ERRHND
      .
      .
      .
C
C     Create a vector list named VLIST containing 100 connected vectors
C              PVCONN is defined in PROCONST.FOR
C
      CALL PVCBEG ( 'VLIST', 100, .FALSE., .FALSE., 3, PVCONN, ERRHND )
      CALL PVCLIS ( 100, VECS, PL, ERRHND )
      CALL PVCEND ( ERRHND )
C
C     Send a 3d vector to <append> of vecs.
C               PIAPP is defined in PROCONST.FOR.
C
      CALL PSNV3D ( AVEC, PIAPP, 'VLIST', ERRHND )
C
C     Delete 2 vectors from VLIST.
C     PS 390 command: Send fix(2) to <delete>vlist;
C             PIDEL is defined in PROCONST.FOR.
C
      CALL PSNFIX ( 2, PIDEL, 'VLIST', ERRHND )
      .
      .
      .
      END
```

The following example illustrates the use of PROCONSF FORTRAN.

Send to a vector list.

```
      PROGRAM TEST
      INCLUDE ( PROCONSF FORTRAN )
      LOGICAL*1 PL (100)
      REAL*4 VECS( 4,100 ), AVEC( 3 )
C
C     Always declare user error handler external
C
      EXTERNAL ERRHND
      .
      .
      .
C
C     Create a vector list named VLIST containing 100 connected vectors
C            PVCONN is defined in PROCONSF FORTRAN
C
      CALL PVCBEG ( 'VLIST', 100, .FALSE., .FALSE., 3, PVCONN, ERRHND )
      CALL PVCLIS ( 100, VECS, PL, ERRHND )
      CALL PVCEND ( ERRHND )
```

```
C
C     Send a 3d vector to <append> of vecs.
C            PIAPP is defined in PROCONSF FORTRAN.
C
      CALL PSNV3D ( AVEC, PIAPP, 'VLIST', ERRHND )
C
C     Delete 2 vectors from VLIST.
C     PS 390 command: Send fix(2) to <delete>vlist;
C            PIDEL is defined in PROCONSF FORTRAN.
C
      CALL PSNFIX ( 2, PIDEL, 'VLIST', ERRHND )
      .
      .
      .
      END
```

2. VAX and IBM Pascal Graphics Support Routines

The PS 390 VAX Pascal GSRs are written in Pascal V2 and are supported only in a VAX/VMS environment. The GSRs are supported under PS 390 Graphics Firmware Release P5.V03 and higher. There are no specific hardware requirements.

The PS 390 IBM Pascal GSRs are written in IBM Pascal/VS and are compatible with IBM VM/CMS and TSO environments. They require an IBM Pascal/VS compiler, Release 2.0. The GSRs are supported under PS 390 Graphics Firmware Release P5.V03 and higher. There are no specific hardware requirements.

Appendix C contains a Pascal V2 network creation example program, and Appendix D contains a Pascal/VS network creation example program. Both programs contain an error handler routine.

2.1 Pascal GSR Conventions

2.1.1 Pascal V2

The Pascal V2 version of the GSRs make use of the following program-defined Pascal TYPE definitions.

```
P_VaryingType    = VARYING [P_MaxVaryingSize] OF CHAR;
P_VaryBufType    = VARYING [P_MaxVaryBufSize] OF CHAR;
P_KnotArrayType  = ARRAY [1..P_MaxKnots] OF REAL;
P_MatrixType     = ARRAY [1..4, 1..4] OF REAL
```

```
P_VectorType     = RECORD
                   Draw : BOOLEAN;
                   V4   : ARRAY [1..4] OF REAL;
                   END;
P_VectorListType = ARRAY [1..P_MaxVecListSize] OF
                   P_VectorType;
P_PatternType    = ARRAY [1...32] of INTEGER;
```

2.1.2 Pascal/VS

The Pascal/VS version of the GSRs make use of the following program-defined Pascal TYPE definitions.

```
P_KnotArrayType  = ARRAY (.1..P_MaxKnots) OF SHORTREAL;
P_MatrixType     = ARRAY (.1..4, 1..4.) OF SHORTREAL;
P_VectorType     = RECORD
                   Draw : BOOLEAN;
                   V4   : ARRAY (.1..4.) OF SHORTREAL;
                   END;
P_VectorListType = PACKED ARRAY (.1..P_MaxVecListSize) OF
                   P_VectorType;
P_PatternType    = ARRAY (.1...32.) of INTEGER;
P_MaxKnots       = 10;  This parameter can be changed by the user
                        to any appropriate value WITHOUT
                        recompiling the GSRs.
P_MaxVecListSize = 100; This parameter can be changed by the user
                        to any appropriate value WITHOUT
                        recompiling the GSRs.
```

The Pascal V2 and Pascal/VS versions of the Graphic Support Raster Routines make use of the following program-defined Pascal CONSTANT definitions:

```
P_MaxRunclrSize = User specified maximum length run color array

P_ColorType     = RECORD
                  RED   : INTEGER;
                  GREEN : INTEGER;
                  BLUE  : INTEGER;
End;
```

```
     P_RunColorType   = RECORD
                        COUNT : INTEGER
                        RED   : INTEGER;
                        GREEN : INTEGER;
                        BLUE  : INTEGER;
   End;

   P_RunClrArrayType  =  ARRAY [1..P_MaxRunclrSize] of P_RunColorType;
```

The following parameters can be changed by the user to any appropriate value without having to recompile the GSRs:

```
P_MaxKnots           =    10
P_MaxVecListSize     =   200
P_MaxVaryingSize     =   255
P_MaxVaryBufSize     =   512
```

2.2 Utility and Application Routines

Utility Routines are specific to the operation of the GSRs. These calls are used to attach the PS 390, select multiplexing channels, send and receive messages, and detach.

Application Routines correspond almost one for one with the standard PS 390 Commands. In most cases, the names for the Application Routines were derived by choosing an abbreviation of the PS 390 command and prefixing it with a **P**. Parameter ordering generally coincides with the PS 390 commands as well. Examples of some of the Application Routines are below.

EXAMPLE 1

For commands which build operation display structures, such as

```
Name:=operate parameter1,parameter2,..., then apply;
```

The routine call is:

```
Poper('name',parameter1,parameter2,...,'apply', Error_Handler);
```

where:

oper is an abbreviated form of the PS 390 command such as rotate in x — Protx

'name' is a character string containing the name to be associated with the operate

parameter1,parameter2,..., are the parameters to be used in computing the operation. These may be boolean values, integers, real numbers, vectors, or matrices.

'apply' is a character string containing the name of the object to which this operate applies.

Error_Handler is the user-defined error handler routine.

EXAMPLE 2

For commands to send to functions or display structures, such as

```
Send datum to <input>dest;
```

The routine call is:

```
PSNDtyp(datum,input,'dest', Error_Handler);
```

where:

'typ' is an abbreviated form of the PS 390 command such as PSndFiX, PSndM2D,...

datum is what is to be sent. It may be Boolean, integer, real, character string, vector, or matrix.

input is an integer which specifies which input of the destination is being sent to.

'dest' is a character string containing the name of the display structure or function.

Error_Handler is the user-defined error handler routine.

EXAMPLE 3

For commands which create functions and connections such as:

```
Name := f:genfcn;
Name := f:genfcn(n);
Conn name<output>:<input>dest;
DISCONN name<output>:<input>dest;
```

The routines are:

```
PFNINST     ( 'name', 'genfcn', Error_Handler );
PFNINSTN    ( 'name', 'genfcn', n, Error_Handler );
PCONNECT    ( 'name',output,input,'dest', Error_Handler );
PDISC       ( 'name',output,input,'dest', Error_Handler );
```

where:

> **'name'** is a character string containing the name associated with the function instance.
>
> **'genfcn'** is a character string containing the name of the system generic function.
>
> **n** is an integer specifying the number of input/outputs for this function instance.
>
> **output,input** are integers specifying the output and input numbers.
>
> **'dest'** is a character string containing the name of the display data structure
>
> **Error_Handler** is the user-defined error handler routine.

Note that the function names in the GSRs are specified without the "F:" prefix that is used in the standard PS 390 command language.

2.3 Exceptions

To be fully specified using GSRs, three PS 390 commands require three separate calls to routines. The commands are LABEL, VECTOR_LIST, and POLYGON.

For example, to create, specify and complete a label block, the user must call:

```
PLabBegn - To create and open a label block
PLabAdd  - May be called multiple times to add to a previously
           opened label block
PLabEnd  - To complete the creation of a label block.
```

Together these three routines implement the PS 390 command:

```
Name :=  LABELS  x, y, z, 'string'
                        .
                        .
                 x, y, z, 'string';
```

In the same way, the user must use PVecBegn to begin a vector list, PVecList to send a piece of a vector list, and PVecEnd to end a vector list.

An example of a routine that varies slightly from the PS 390 command is PBSPL. In the BSPLINE command some of the parameters are optional. In the routine they are all required. This is also the case for the PRBSPL, PPOLY, and PRPOLY routines.

The PS 390 syntax allows for instancing multiple display entities and for creating multiple variables. In the PS 390 command language the commands would be:

```
NAME:= INSTANCE a,b,c,d;
```

for instancing multiple display entities, and

```
VARIABLE s,y,z,w,t,q;
```

for multiple variables.

To perform the equivalent instancing of multiple display entities or for creating multiple variables, the following GSRs should be used.

For the multiple instance case:

```
PINST('NAME', 'A', Error_Handler);
PINCL('B', 'NAME', Error_Handler);
PINCL('C', 'NAME', Error_Handler);
PINCL('D', 'NAME', Error_Handler);
```

For the multiple variable case:

```
PVAR ('S', Error_Handler);
PVAR ('Y', Error_Handler);
PVAR ('Z', Error_Handler);
PVAR ('W', Error_Handler);
PVAR ('T', Error_Handler);
PVAR ('Q', Error_Handler);
```

2.4 Error Handling

An error handling scheme has been employed to catch errors detected by the GSRs. Examples of errors detected by the GSRs:

- Prefix not followed by an operate.
- Follow not followed by an operate.
- Invalid characters in a name.

Command Interpreter errors and warnings are not detected by the GSRs. Examples of these errors are:

- Destination does not yet exist.
- Message rejected by destination.
- Connection not made.

Error checking will be performed within the GSRs to ensure that only valid characters are sent within names, and that routines are called in the proper order, in cases where order is required. No attempt has been made to capture errors and/or warnings from the Command Interpreter.

Each routine call includes an argument that specifies the user-written error handler. This error handler is of the form:

```
PROCEDURE Error_Handler (Error : INTEGER);
```

where ERROR is an integer error code corresponding to one of the errors.

It is the responsibility of the user to provide an error handling scheme to decide what action should be taken when an error is detected. The GSRs do not attempt to terminate execution or log errors.

The name, description, and error code of each detectable error is given in tables in Section *RM4*. An example error handling routine appears in the example programs in Appendix C and Appendix D. It is a sophisticated error handler that may be incorporated by the user into an error handling scheme, or used as an example of what an error handler should look like.

2.5 Programming Suggestions

The file PROCONST.PAS contains definitions for constants used by the GSRs. It is often convenient to think of these constants by name rather than by remembering numbers. Specifically, in the usual PS 390 command syntax, inputs to display structures are often referred to by name such as <append> and <clear> for vector_lists and <position> and <step> for character strings. There are also <delete>, <last>, and others. Other useful constants such as values for conditional tests for level of detail, and vector list class are obtainable from PROCONST.PAS.

PROCONST.PAS also contains a complete set of error/warning code definitions. These values may be referenced by name by the user routine if PROCONST.PAS is INCLUDED in the routine. The Error Tables in Section RM4 provide a list of the mnemonics and error codes. Using the mnemonics provides an easy way of checking for the correct error code value.

There are two other files that must be INCLUDED by the user. These additional files and their descriptions are:

PROTYPES.PAS – contains the GSR Pascal TYPE definitions
PROEXTRN.PAS – contains the VAX GSR EXTERNAL Routine Definitions

The following is an abbreviated list derived from PROCONST.PAS of the constants which should be most useful to the user.

GSR private constant declarations:

Name	Value	Meaning
P_Append	= 0;	<Append> input number.
P_Delete	= -1;	<Delete> input number.
P_Clear	= -2;	<Clear> input number.
P_Step	= -3;	<Step> input number.
P_Position	= -4;	<Position> input number.
P_Last	= -5;	<Last> input number.
P_Substitute	= -6;	<Substitute> input number.
P_LES	= 0;	"Less" level of detail comparison operator.
P_EQL	= 1;	"Equal" level of detail comparison operator.
P_LEQL	= 2;	"Less-equal" level of detail comparison operator.
P_GTR	= 3;	"Greater" level of detail comparison operator.
P_NEQL	= 4;	"Not-equal" level of detail comparison operator.
P_GEQL	= 5;	"Greater-equal" level of detail comparison operator.
P_Conn	= 0;	Vector list "Connected" class type.
P_Dots	= 1	Vector List "Dots" class type.
P_Item	= 2;	Vector List "Itemized" class type.
P_Sepa	= 3;	Vector List "Separate" class type.
P_Tab	= 4;	Vector List "Tabulated" class type.

3. UNIX/C Graphics Support Routines

The GSR Library provides **C** functions for each command accepted by the Command Interpreter. It is assumed that the E&S software distribution has been loaded into subdirectories of a directory named **/usr/ps300/dist/es.**

The object code for the PS 390 GSRs exists as the library archive file, **libgsr.a** in the **lib** subdirectory. Application programs using the GSRs should be linked with this library and must include **gsrext.h** from the include subdirectory. There are other header files in the include subdirectory, which may be used by programs which need access to the lower levels of the GSR library. They are **pidefs.h** and **netdefs.h.** Programs which call the piqiow and psnetio routines directly should include **pidefs.h** and **netdefs.h** respectively. These files define constants and data types, and declare external routines used in the corresponding modules. The **pidefs.h** file defines the data types and symbolic errors and other constants used by the parallel interface device driver. The **netdefs.h** file defines constants and macros used by the **psnetio** function.

3.1 The Lint library

The GSR library passes through the lint analyst without errors. Application programs should make use of the objects defined in the header files and the analysis of lint. The lint library **llib-lgsr.ln** is provided so that lint may analyze the usage of the GSRs in a user application program. This file exists in the **lib** subdirectory. The GSR lint library will be automatically updated whenever the GSR library is modified.

3.2 UNIX/PS 390 Communication Channels

The VAX/UNIX/C GSRs support communication with a PS 390 over an asynchronous serial line, the PS 390/UNIBUS parallel interface and the PS 390/Ethernet Interface. The synchronous serial interface device (DMR–11) is not supported. Output to disk files instead of the PS 390, is also supported. Application programs specify the communication channel using the PAttach routine in the GSR library, as shown below:

```
PAttach( devname );
```

where devname is a character string specifying a device special file. A hyphen ("-") is a special case and is assumed to specify the **stdout** file.

3.2.1 Automatic device typing

Regardless of the device specified by the application program for I/O to the PS 390, the type of the device being used (parallel, RS–232 line, or Ethernet node) is automatically determined by the GSR package at the time of initialization. For example, if **stdout** was specified in the application program (in the PAttach routine) and was redirected to a PS 390 parallel interface special file (regardless of the name of that special file) it will be recognized as such and controlled appropriately.

3.2.2 Parallel Interface Device

The GSR user needs only to know the name of the UNIX special file which is used to access the device. The recommended name pattern is /dev/pi?0. The second to last character in the name is a character in the range "a" through "z", identifying one of possibly several picture processors.

3.2.3 PS 390/Ethernet Interface

To use the PS 390 on the customer's Ethernet, its Official Host Name or an alias (found in /etc/hosts) should be specified as the "devname" to the

PAttach. For example, the shell scripts in the test subdirectory of the distribution assume that the /etc/hosts file contains an entry for a node named "net_300_1" and pass "net_300_1" to the test programs, which is used in the PAttach call.

Other facts to remember about the PS 390 Ethernet interface are:

- Dual line operation of the PS 390 is precluded.
- The 4.2 BSD UNIX trailer protocols must be turned off for the PS 390 Ethernet interface to work.

3.2.4 RS-232 Asynchronous serial channels

The GSR, PAttach, attempts to set up the communication line properly for asynchronous communications. However, some special precautions should be taken when using the async lines with the GSRs to communicate with the PS 390. If the tty being used for the PS 390 is not the same as the process's controlling tty(i.e., the PS 390 is operating in dual line mode or a separate terminal is being used) then logins must not be enabled for the device (i.e., no getty running on it). You may disable logins permanently for such a line by editing its entry in /etc/ttys (change the first character on the corresponding line to "0") and sending a hangup signal to init (with a "kill-HUP 1") or rebooting the system. Alternately, if your site has implemented enable/disable commands to do this more gracefully as required, you can disable the line dynamically upon each use. Note that anything written to **stderr** (as the standard GSR error handler does) will be sent to your terminal (i.e., the PS 390, probably in the middle of some binary data packet) unless you take steps to redirect it elsewhere.

3.2.5 Output to Files

Any file may be specified to receive the output of the GSRs for examination during debugging or delayed transmission to the PS 390. In this case, the output is formatted with the same type of headers used for asynchronous serial lines so that the file may simply be copied (e.g., with cat) to a PS 390 at a later time. Care must be taken to ensure that the asynchronous communication line is set up properly since the PAttach/PDetach calls usually take care of this for you. When directing output to a file other than a real PS 390, it is important to note that any GSR operations which require reading from the PS 390 will return nothing and generate a call to the error handler indicating a PSF_PHYGETFAI condition. In general, this can cause problems in the program's operation and is not recommended.

3.3 UNIX/C GSR Conventions

3.3.1 Names of user callable GSR library routines

The actual names of the user callable routines in the GSR library consist of upper and lower case letters and digits. However, application programs which use the GSRs may call these routines using all lower case letters provided the symbol "LOWERCASE_GSR" is defined when the **C** preprocessor processes the mandatory **gsrext.h** header file.

3.3.2 Data types of GSR parameters

The number, ordering, and function of parameters to the GSR library is, in general, the same as in the VMS Pascal implementation. There are a few exceptions as noted in the sections below. The type and/or structure of some of the parameters have also been changed as described in the following.

3.3.3 Strings and Bytes

All character string and byte buffer parameters are of the type **string** (i.e., pointer to a char) in C, whereas in the Pascal version they are P_VaryingType and P_VaryBufType. In C, this is equivalent to an array of char or to a quoted literal text string. When these are strings of ASCII characters, the C convention of terminating the string with a null character is assumed, requiring no explicit indication of length. There are three routines in the C version which make an exception to this convention and require explicit specification of length since they treat a string as a general purpose byte buffer which may contain the value zero (the conventional string terminator) as valid data. These routines are:

```
PPutG(buffer, actual_length)
string buffer;
int length;

int PGet(buffer, max_length)
string buffer;
int max_length;

int PGetWait(buffer, max_length)
string buffer;
int max_length;
```

In all three cases, the buffer pointer is followed by an integer length value. The length parameter to the PPutG routine is the number of bytes of valid data in the buffer. However, in PGet and PGetWait, the length parameter indicates the size of the buffer in bytes (i.e., the maximum number of bytes which may be returned in the buffer). The PGet and PGetWait routines return the number of bytes read from the PS 390 as their function value.

3.3.4 Booleans

A Boolean type definition (with values of TRUE and FALSE) is used in place of Pascal's built-in boolean type.

3.3.5 Floating Point Numbers

The 4.2 BSD C compiler and most others perform all floating-point operations in double precision, treat all f.p. constants as double precision, and convert all f.p. expressions and actual parameters to routines to double precision, and implicitly declare all formal f.p. parameters as double precision. For this reason, the C GSRs use the type double in place of the Pascal type REAL.

3.3.6 Other Special Types and Constants

The other special types used in the Pascal version have been converted to **C** with essentially no change in structure. They are:

```
typedef enum (P_Conn=0,P_Dots,P_Item,P_Sepa,P_Tab) P_vector_class;
typedef long int P_PatternType[32];
typedef double P_KnotArrayType[];
typedef double P_MatrixType[4][4];

typedef struct {
boolean Draw;
double V4[4];
} P_VectorType;

typedef P_VectorType P_VectorListType[];

typedef struct {
unsigned short red;
unsigned short green;
unsigned short blue;
} P_ColorType;
```

```
typedef struct {
int count;
unsigned short red;
unsigned short green;
unsigned short blue;
} P_RunColorType;

typedef P_RunColorType P_RunClrArrayType[];
```

The set of **P_Max...** constants used in Pascal as limits for the variable length arrays in Pascal are unnecessary in C due to the flexibility in passing arrays of different lengths as parameters.

All of the defined constants used in Pascal for specifying symbolic values for certain integer parameters (e.g., P_Delete, P_LES, P_Dots etc.,) are also available in the C GSRs.

3.4 Error Handling

All errors which were detected in the VMS version are likewise detected in the UNIX/C version (except for the VMS-specific errors). A few UNIX-specific errors have been added using the same scheme for error code assignment. Error handling in the C implementation is more efficient and more flexible. A standard error handler is provided in the GSR library which is used by default if an alternate one is not specified. The application program may provide its own error handling routine(s) at various points in the program. Passing an error handler routine to each of the GSRs has been eliminated. The symbolic names of GSR errors are defined in the header file **gsrerror.h** in the **include** subdirectory of the distribution.

3.4.1 The Default Error Handler

The default error handling routine provided by the GSR library is:

```
gsr_std_err_handler(error)
P_gsr_error_type error;
```

This routine prints a message on the stderr file, giving the integer error code passed to it.

3.4.2 User provided Error Handlers

An application program using the GSRs may provide its own error handling routine, even though the gsr_std_err_handler routine is in effect by default.

An alternate error handler may be specified using the set_gsr_err_handler routine as shown below.

```
set_gsr_err_handler (new_handler)
int (* new_handler) ();
```

where new_handler is the alternate error handler routine which should take one parameter of P_gsr_error_type which will be passed the current error code when called. If the set_gsr_err_handler routine is called with a null pointer (i.e., zero) it will revert back to the use of the default handler.

3.4.3 Error Handler Parameter

The address of an error handler is passed as a parameter to every routine in the VMS Pascal version of the library. However, since the C version provides a standard error handler and a mechanism for specifying user-supplied error handlers, this parameter has been eliminated. Whatever error handler is in effect is invoked directly at the site of the error detection.

3.5 Special Notes

3.5.1 Specifying Transformation Matrices

The following GSR library routines require the input of a transformation matrix:

```
PMat2x2  PMat3x3  PMat4x3  PMat4x4  PSndM2d  PSndM3d  PSndM4d
```

The GSR library can be built for input of transformation matrices in ROW MAJOR or COLUMN MAJOR mode by specifying the rowmajor or the columnmajor option when invoking the make utility which generates it. The default is the rowmajor option. The library archive **libgsr.a** which is distributed on magtape is built using the rowmajor option.

3.5.2 Miscellaneous notes on using the GSRs

There is no easy way for an application program to know when the PS 390 has completely processed the GSR equivalent of a PS 390 command and is in a quiescent state. The absence of such a general facility could be the source of some unpredictable program behavior unless the appropriate precautions are taken. A typical situation for such behavior is illustrated by the following segment of an application program.

```
char dummy[256];

/* use the parallel interface*/

PAttach("/dev/pia0");
PInit();

/* Connect Function Key 1 to HOST_MESSAGE so that when it is hit,

PGetWait will wakeup*/
PConnect("FKEYS",1,1,"HOST_MESSAGE");
PFnInst("inst_of_print","PRINT");

/* Create a character data node*/

CharSca("object", 0.04, 0.04, "charnode");
PChars("charnode",0.2,0.83,0.0,1.0,0.0,"");

/* Connect inst_of_print to the substitute input of the character
data node*/

PConnect("inst_of_print",1,-6,"object");
PSndStr("Hello",1,"inst_of_print");
PDisplay("object");

/* Wait for PS 390 user to hit key F1 */

PGetWait( dummy,256 );

/* At this point, the string "Hello" would appear on the PS 390
display. Now let us try to disconnect inst_of_print from object and
send the string "ABC" to it and THEN connect it again to object.*/

PDisc("inst_of_print",1,"object");
PSndStr("ABC",1,"inst_of_print");
PConnect("inst_of_print",1,-6,"object");

/* Wait for PS 300 user to hit key F1 */

PGetWait( dummy,256 );

/* It seems reasonable to expect that "ABC" should not be displayed
because the functions were disconnected prior to sending "ABC".
However, because of the way some commands are interpreted by the CI,
it is very likely that the string "ABC" ended up on the input of
inst_of_print AFTER the connection was made the second time, thus
making"ABC" appear on the display!  */
```

One could invent a specific solution to each situation of this kind which would satisfactorily take care of the problem. In the case above, we could resort to a crude solution by which the application program could wait for a reasonably long time, say 1 second, after sending the string "ABC" but before doing the PConnect again as shown below:

```
PDisc("inst_of_print",1,"object");
PSndStr("ABC",1,"inst_of_print");

/* Give enough time for the string to end up on the input of
inst_of_print before making the connection again*/ !

sleep(1);

/* sleep for 1 second */

PConnect("inst_of_print",1,-6,"object");
PGetWait( dummy,256 );

/* Wait for F1 key */
```

3.5.3 Using the PGetWait routine in the GSR library

The PGetWait routine is called by host application programs when they want to wait for some input from the PS 390, typically sent back via the HOST_MESSAGE function. However, when the application program uses an asynchronous communication line to the PS 390, all keys typed on the PS 390 keyboard in Terminal Emulator mode will also be sent to the host (bypassing HOST_MESSAGE). This means that the application program may receive unexpected input and behave in an unexpected manner. The solution is to make sure that the PS 390 keyboard is not in Terminal Emulator mode when PGetWait is used to wait for input from the PS 390. This situation does not arise when the VAX/UNIBUS Parallel Interface or the Ethernet Interface is used to communicate with the PS 390 because the Terminal Emulator uses only the async line.

3.6 Ethernet Interface I/O Operations

The PS 390 Ethernet Interface allows a PS 390 to be connected to an Ethernet local area network. The E&S software for this option consists of (a) PS 390 firmware which implements full duplex byte stream communications over an Ethernet and (b) the GSR library which provides easy access to the networked PS 390. The DoD TCP/IP is used in the initial release; a simpler E&S protocol may also be supported in the near future.

Application programs which use the GSR library can access the networked PS 390 by specifying its node name or an alias to the PAttach function as shown:

```
PAttach("PS 390nodeName");
```

The name of the PS 390 node and its aliases can be found in the /etc/hosts file.

Application programs may communicate with the networked PS 390 by calling the **psnetio** function in the **devlib.c** module of the GSR library. This function hides the protocols used and relieves the programmer of some housekeeping chores in handling the interface. More importantly, the physical I/O operations of the interface will be accessible only by calling **psnetio** directly.

3.6.1 The PSNETIO Function

The **psnetio** function allows the caller to issue logical, physical and diagnostic I/O requests to the PS 390/Ethernet Interface. The request parameter is an integer code passed to **psnetio** from the calling program. The need for and the meaning of the **argp1** and **argp2** parameters depends on the request. The request codes available and the corresponding parameter list are described below:

PSNET_ATT

This request is issued to create a communication socket and to establish a virtual circuit between this socket and a socket on the PS 390 node. The argp1 parameter is a pointer to the name of the PS 390 node or one of its aliases from the /etc/hosts file. There is no argp2 parameter for this request. If the connection is established successfully, a file descriptor value is returned; otherwise -1.

PSNET_DET

This request is issued to shutdown the virtual circuit established by the PSNET_ATT request. Further communication with the PS 390 node would be possible only after a new PSNET_ATT request. There are no other parameters for this request.

A 0 is returned if the call succeeds; -1 if it fails.

PSNET_LWRITE

This request is issued to send data to the function network in the PS 390 node. The argp1 parameter is a pointer to a user buffer which has the following format:

```
opcode    (8 bits)
flags     (8 bits)
bc        (16 bits)
lwdata    (bc bytes)
```

The fields of this buffer are:

opcode

Not written or read by psnetio's caller.

flags

The F_WRITESWAP bit should be set if the bytes in each 16-bit word of the lwdata field should be swapped by the PS 390 before passing them to the function network.

The F_READSWAP bit should be set if the bytes should be swapped in each 16-bit word sent to the host for a PSNET_LREAD request.(See below.)

bc

Number of bytes of data in lwdata. This does not include the opcode, flags and bc fields.

lwdata

Data to be sent to the function network.

The number of bytes of data sent, excluding the opcode, flags and bc fields, is returned as the function value. A -1 is returned in case of an error. There is no argp2 parameter for this request.

NOTE

A PSNET_LWRITE request throws away any data from the PS 390 which still remains unread.

PSNETLREAD

This request is issued to read data from the function network in the PS 390 node. The argp1 parameter is a pointer to a user buffer which has the following format:

```
opcode   (8 bits)
flags    (8 bits)
bc       (16 bits)
lrdata   (bc bytes)
```

The fields of this buffer are:

opcode

Not written or read by psnetio's caller.

flags

Bit mask returned by the PS 390. If the F_SWAPPED bit is set, the order of bytes in each 16-bit word of the lrdata field is the opposite of the way they are stored in PS 390 memory.

bc

Size of the lrdata field, in bytes. This field is modified by psnetio. On return, it contains the number of bytes of data received from the PS 390.

lrdata

All or part of the data received from the PS 390 function network.

The number of bytes of data, excluding the opcode, flags and bc fields is returned as the function value. A --1 is returned in case of an error. There is no argp2 parameter for this request.

NOTE

(1) If the order of data bytes from the PS 390 is to be reversed, it should have been specified by setting the F_READSWAP bit in the most recent PSNET_LWRITE request prior to the read.

(2) Psnetio may be called repeatedly to recover all of the data received from the PS 390.

(3) Any data remaining unread after a PSNET_LREAD will be thrown away by requests other than a PSNET_LREAD.

PSNET_PREAD

This request is issued to read data directly from PS 390 mass memory. The argp1 and argp2 parameters are pointers to two user buffers; the first contains a set of descriptors for the physical read operation and the second receives the data read from mass memory and a copy of the descriptors from the first buffer. The argp1 buffer has the following format:

```
opcode              (8 bits)
flags               (8 bits)
bc                  (16 bits)
N                   (16 bits)
Block #1 mmaddr     (32 bits)
Block #1 wc         (16 bits)
Block #2 mmaddr     (32 bits)
Block #2 wc         (16 bits)
Block #N mmaddr     (32 bits)
Block #N wc         (16 bits)
```

The fields of this buffer are:

opcode

Not written or read by psnetio's caller.

flags

The F_READSWAP bit should be set if the bytes should be swapped in each 16-bit word read from PS 390 mass memory before being sent to the host.

bc

Number of bytes in the rest of the buffer.

N

Number of blocks of mass memory to read in this request. Each block is specified by a mass memory address and a word count.

mmaddr

Source mass memory address.

wc

Number of 16-bit words to read beginning at mmaddr.

The argp2 buffer receives the physical read data and a copy of the other fields from the argp1 buffer. It has the following format:

```
opcode              (8 bits)
flags               (8 bits)
bc                  (16 bits)
N                   (16 bits)
Block #1 mmaddr     (32 bits)
Block #1 wc         (16 bits)
Block #1 mmdata     (per wc)
Block #2 mmaddr     (32 bits)
Block #2 wc         (16 bits)
Block #2 mmdata     (per wc)
Block #N mmaddr     (32 bits)
Block #N wc         (16 bits)
Block #N mmdata     (per wc)
```

The fields of the argp2 buffer are:

opcode

Not written or read by psnetio's caller.

flags

Bit mask returned by the PS 390. If the F_SWAPPED bit is set, the order of bytes in each 16-bit word returned in mmdata is the opposite of their order in PS 390 memory.

bc

Size of the rest of the buffer, in bytes. This field is modified by psnetio.

mmaddr

Copied by the PS 390 from the argp1 buffer.

wc

Copied by the PS 390 from the argp1 buffer.

mmdata

Physical data read from mass memory.

The number of bytes of data returned in the argp2 buffer is returned as the function value. This count excludes the opcode, flags, bc, N, mmaddr and wc fields. A –1 is returned in case of an error.

NOTE

Any physical data remaining unread after a PSNET_PREAD request is thrown away by the first request other than a PSNET_PREAD.

PSNET_PWRITE

This request is issued to write data directly to PS 390 mass memory. The argp1 parameter is a pointer to a user buffer containing the target mass memory addresses, word counts and the data for the physical write operation. The buffer has the following format:

opcode	(8 bits)
flags	(8 bits)
bc	(16 bits)
N	(16 bits)
Block #1 mmaddr	(32 bits)
Block #1 wc	(16 bits)
Block #1 mmdata	(per wc)
Block #2 mmaddr	(32 bits)
Block #2 wc	(16 bits)
Block #2 mmdata	(per wc)
Block #N mmaddr	(32 bits)
Block #N wc	(16 bits)
Block #N mmdata	(per wc)

The fields of this buffer are:

opcode

Not written or read by psnetio's caller.

flags

The F_WRITESWAP bit should be set if the bytes should be swapped in each 16-bit word of mmdata before being written to PS 390 mass memory.

bc

Size of the rest of the buffer, in bytes.

N

The number of blocks of mass memory to write in this request. Each block is specified by a mass memory address, the number of 16-bit words of data to write and the data.

mmaddr

Destination mass memory address.

wc

Number of 16-bit words to write, beginning at the mass memory address specified in mmaddr.

mmdata

Data to be written to mass memory.

The number of bytes of data sent to the PS 390 is returned as the function value. This count excludes the opcode, flags, bc, N, mmaddr and wc fields. A -1 is returned in case of an error.

NOTE

> Any physical data remaining unread after a PSNET_PREAD request is thrown away by the PSNET_PWRITE request.

PSNETLOOKUP

This request is issued to get the named entity pointer (i.e., the mass memory address) of a PS 390 object. The argp1 parameter is a pointer to a character string containing the name of the object. It has the following format:

```
opcode      (8 bits)
flags       (8 bits)
bc          (16 bits)
name        (bc bytes)
```

The fields of the buffer are:

opcode

Not written or read by psnetio's caller.

flags

Not read or written by psnetio's caller.

bc

Number of characters in name.

name

Character string specifying the name of the PS 390 object whose address is to be looked up.

NOTE

> The order in which the characters in the name are sent is assumed to be the order in which the PS 390 expects them. In other words, the PS 390 does not swap the bytes before looking up the name. The pointer to the named entity is returned as the function value. A -1 is returned in case of an error.

Appendix A - FORTRAN-77 Example Program

This appendix contains a network creation example program that illustrates the use of the PS 390 DEC/VAX FORTRAN-77 Graphics Support Routines. The program contains an error handler routine example.

```
      PROGRAM BlkLevF

      INCLUDE ´PROCONST.FOR/NOLIST´

C
C       Main program:
C
      REAL*4    Deg_rad
      PARAMETER (Deg_rad    = 0.017453292)

      REAL*4    Theta, DTheta, Front (4, 100),
     &          Vecs (4, 100), Zero_vec (3),
     &          Y_Up (3), At (3), From (3), Up (3)
      INTEGER*4 i, k, l, Times
      CHARACTER Name*63, DeviceSpec*1, DeviceName*5,
     &          AttachParameter*80
      LOGICAL*1 PFront (100), PVecs (100)

      CHARACTER Uppercase*1
      EXTERNAL  Err, Uppercase


      DeviceSpec = ´ ´
      DO WHILE ((DeviceSpec .NE. ´A´) .AND.
     &          (DeviceSpec .NE. ´E´) .AND.
     &          (DeviceSpec .NE. ´P´))
        WRITE (6, 1) ´Device Interface type = ´
     &            // ´(Parallel, Ethernet, Asynchronous): _´
        READ  (5, 2)  DeviceSpec
        DeviceSpec = Uppercase (DeviceSpec)
        IF ((DeviceSpec .NE. ´A´) .AND.
     &      (DeviceSpec .NE. ´E´) .AND.
     &      (DeviceSpec .NE. ´P´)) THEN
          WRITE (6, *) ´Invalid device type specified.´
        END IF
      END DO
      DeviceName = ´ ´
      DO WHILE (DeviceName .EQ. ´ ´)
        WRITE (6, 1) ´Physical device name (i.e. TT, ´
 &            // 'TTA6, XMDO): _'
    READ  (5, 3)  DeviceName
1   FORMAT (' ', A, $)
2   FORMAT (1A)
3   FORMAT (5A)
  END DO
  IF ((DeviceName (2:2)) .EQ. ' ') THEN
    DeviceName (2:) = ':'
  ELSE
    IF ((DeviceName (3:3)) .EQ. ' ') THEN
      DeviceName (3:) = ':'
    ELSE
      IF ((DeviceName (4:4)) .EQ. ' ') THEN
        DeviceName (4:) = ':'
      ELSE
        DeviceName (5:) = ':'
      END IF
    END IF
  END IF
  IF ((Uppercase (DeviceSpec)) .EQ. 'P') THEN
    AttachParameter = 'Logdevnam=' // DeviceName
 &                 // '/Phydevtyp=PARALLEL'
  ELSE
    IF ((Uppercase (DeviceSpec)) .EQ. 'E') THEN
      AttachParameter = 'Logdevnam=' // DeviceName
 &                   // '/Phydevtyp=Ethernet'
    ELSE
      AttachParameter = 'Logdevnam=' // DeviceName
 &                   // '/Phydevtyp=Async'
    END IF
  END IF
  CALL PAttch (AttachParameter, Err)
  At (1) = 0.3
  At (2) = 0
  At (3) = 0
  From (1) = 0
  From (2) = 0
  From (3) = -1
  Up (1) = 0.3
  Up (2) = 1
  Up (3) = 0
  Y_up (1) = 0
  Y_up (2) = 1
  Y_up (3) = 0
  Zero_vec (1) = 0
  Zero_vec (2) = 0
  Zero_vec (3) = 0
  CALL Pinit ( Err )
 CALL Peyebk ( 'eye', 1.0, 0.0, 0.0, 2.0, 0.0,
&             1000.0, 'inten', Err )
 CALL Pseint ( 'inten', .TRUE., 0.5, 1.0,
&             'look', Err )
 CALL PLooka ( 'look', At, From, Up, 'pic', Err )
 CALL Pfn    ( 'atx', 'xvec', Err )
 CALL Pfn    ( 'aty', 'yvec', Err )
 CALL Pfn    ( 'atz', 'zvec', Err )
 CALL Pfn    ( 'fromx', 'xvec', Err )
 CALL Pfn    ( 'fromy', 'yvec', Err )
 CALL Pfn    ( 'fromz', 'zvec', Err )
 CALL Pfn    ( 'ac_at', 'accumulate', Err )
 CALL Pfn    ( 'ac_from', 'accumulate', Err )
 CALL Pfn    ( 'add_up', 'addc', Err )
 CALL PfnN   ( 'sync_up', 'sync', 3, Err )
 CALL Pfn    ( 'fix_sync', 'nop', Err )
 CALL Pconn  ( 'sync_up', 3, 1, 'fix_sync', Err )
 CALL Pconn  ( 'fix_sync', 1, 3, 'sync_up', Err )
 CALL Psnboo ( .TRUE., 3, 'sync_up', Err )
 CALL Pfn    ( 'look_at', 'lookat', Err )
 CALL Pconn  ( 'dials', 1, 1, 'atx', Err )
 CALL Pconn  ( 'dials', 2, 1, 'aty', Err )
 CALL Pconn  ( 'dials', 3, 1, 'atz', Err )
 CALL Pconn  ( 'dials', 5, 1, 'fromx', Err )
 CALL Pconn  ( 'dials', 6, 1, 'fromy', Err )
 CALL Pconn  ( 'dials', 7, 1, 'fromz', Err )
 CALL Pconn  ( 'atx', 1, 1, 'ac_at', Err )
 CALL Pconn  ( 'aty', 1, 1, 'ac_at', Err )
 CALL Pconn  ( 'atz', 1, 1, 'ac_at', Err )
 CALL Pconn  ( 'fromx', 1, 1, 'ac_from', Err )
 CALL Pconn  ( 'fromy', 1, 1, 'ac_from', Err )
 CALL Pconn  ( 'fromz', 1, 1, 'ac_from', Err )
 CALL Pconn  ( 'ac_at', 1, 1, 'sync_up', Err )
 CALL Pconn  ( 'ac_at', 1, 1, 'add_up',  Err )
 CALL Pconn  ( 'add_up',1, 2, 'sync_up', Err )
 CALL Pconn  ( 'sync_up', 1, 1, 'look_at', Err )
 CALL Pconn  ( 'sync_up', 2, 3, 'look_at', Err )
 CALL Pconn  ( 'ac_from', 1, 2, 'look_at', Err )
 CALL Psnv3d ( At, 2, 'ac_at', Err )
 CALL Psnv3d ( From, 2, 'ac_from', Err )
 CALL Psnv3d ( Y_up, 2, 'add_up', Err )
 CALL Pconn  ( 'look_at', 1, 1, 'look', Err )
 CALL Pfn    ( 'fix_at', 'const', Err )
 CALL Pconn  ( 'ac_from', 1, 1, 'fix_at', Err )
 CALL Pconn  ( 'fix_at', 1, 1, 'ac_at', Err )
 CALL Psnv3d ( Zero_vec, 2, 'fix_at', Err )
 CALL Psnv3d ( Zero_vec, 1, 'ac_from', Err )
 CALL Pinst  ( 'pic','"', Err )
 Dtheta = 10.0 * Deg_rad
 Theta  = -Dtheta
 DO i = 1, 36
   Theta = Theta + Dtheta
   CALL Computewave (Theta, Vecs, PVecs)
   DO k=1, 50
     DO l=1, 4
       Front (l, k) = Vecs (l, (k-1)*2+1)
       PFront (k) = PVecs ((k-1) * 2 + 1)
     END DO
   END DO
   CALL Computename ( i, Name )
   CALL Pbegs  ( Name, Err )
   CALL Pser   ( ´"´, 1, 35, .FALSE., i, ´"´, Err )
   CALL Pifpha ( ´"´, .TRUE., ´"´, Err )
   CALL Pvcbeg ( ´"´, 100, .FALSE., .FALSE., 3,
&                PVsepa, Err )
   CALL Pvclis ( 100, Vecs, PVecs, Err )
   CALL Pvcend ( Err )
   CALL Pvcbeg ( ´"´, 50, .FALSE., .FALSE., 3,
&                PVconn, Err )
   CALL Pvclis ( 50, Front, PFront, Err )
   CALL Pvcend ( Err )
   CALL Pends ( Err )
   CALL Pincl ( Name, ´pic´, Err )
 END DO
 CALL Pdisp ( ´eye´, Err )
 CALL PSnSt ( ´X´, 1, ´Dlabel1´, Err )
 CALL PSnSt ( ´Y´, 1, ´Dlabel2´, Err )
 CALL PSnSt ( ´Z´, 1, ´Dlabel3´, Err )
 CALL PSnSt ( ´Look At´, 1, ´Dlabel4´, Err )
 CALL PSnSt ( ´X´, 1, ´Dlabel5´, Err )
 CALL PSnSt ( ´Y´, 1, ´Dlabel6´, Err )
 CALL PSnSt ( ´Z´, 1, ´Dlabel7´, Err )
 CALL PSnSt ( ´From´, 1, ´Dlabel8´, Err )
 CALL Pdtach ( Err )
 END



 SUBROUTINE Computename (Nameid, Name)
 INTEGER*4  NameId
 CHARACTER  Name*(*)

 INTEGER*4  j, L_name

 Name = ´List000"´
 L_name = Nameid
```

```
      j = 7
      DO WHILE (L_name .GT. 0)
        Name (j:j) = CHAR (MOD (L_name, 10) + ICHAR ('0'))
        L_name = L_name/10
        j = j - 1
      END DO
      RETURN
      END
      SUBROUTINE ComputeWave (Theta, VecList, PosLin)
      REAL*4     Theta, VecList (4, 100)
      LOGICAL*1  PosLin (*)

      REAL*4     Amp, Alpha, Beta
      PARAMETER  (Amp     = 0.8, Alpha  = -0.02,
     &            Beta    = 0.2513274123)

      INTEGER*4  i, IAddr

      Iaddr = -1
      DO i = 0, 49
        Iaddr = Iaddr + 2
        Veclist (1, Iaddr) = i / 50.0
        Veclist (2, Iaddr) = Amp * EXP (Alpha * i)
     &                     * cos (Theta - Beta * i)
        Veclist (3, Iaddr) = 0
        Veclist (4, Iaddr) = 1 - i/150.0
        PosLin  (   Iaddr) = .TRUE.
        Veclist (1, Iaddr+1) = Veclist (1, Iaddr)
        Veclist (2, Iaddr+1) = 0
        Veclist (3, Iaddr+1) = 0.5
        Veclist (4, Iaddr+1) = Veclist (4, Iaddr)
        PosLin  (   Iaddr+1) = .TRUE.
      END DO
      RETURN
      END


      CHARACTER*1 FUNCTION Uppercase (Chara)
      CHARACTER   Chara*(*)
      IF (((Chara (1:1)) .GE. 'a') .AND.
     &    ((Chara (1:1)) .LE. 'z')) THEN
        Uppercase = CHAR (ICHAR (Chara (1:1)) - 32)
      ELSE
        Uppercase = Chara
      END IF
      RETURN
      END
```

```
C
C
C
C   The following Error Handler demonstrates the general
C   overall recommended form that the user's own error
C   handler should follow.
C
C   This error handler upon being invoked writes ALL
C   messages to the data file:  'PROERROR.LOG'. Error
C   and warning explanation messages are written to
C   a data file for 2 reasons:
C
C
C       1. The error handler should not immediately
C           write information out on the PS 390 screen
C           since the explanatory text defining the error
C           or warning condition may be taken as data by
C           the PS 390 and therefore wind up not being
C           displayed on the PS 390 screen (as in the
C           case of a catastrophic data transmission
C           error).
C
C       2. The logging of errors and warnings to a
C           logfile allows any errors and/or warnings
C           to be reviewed at a later time.
C
C
C



      SUBROUTINE ERR (ERRCOD)

C
C       Procedural Interface (GSR) error handler:
C

      INCLUDE   'PROCONST.FOR/NOLIST'
      INTEGER*4  ERRCOD
      INTEGER*4  PsVMSerr
      LOGICAL    FILOPN
      DATA       FILOPN /.FALSE./
      EXTERNAL   PsVMSerr, DETERH, PIDCOD

      IF (FILOPN) GOTO 1
C
C       Open error file for logging of errors:
```

```
C
        OPEN (UNIT=10, FILE='PROERROR.LOG', STATUS='NEW',
     &        DISP='KEEP', ORGANIZATION='SEQUENTIAL',
     &        ACCESS='SEQUENTIAL', CARRIAGECONTROL='LIST')
        FILOPN = .TRUE.
C     END IF
    1 CALL PIDCOD (ERRCOD)
      IF (ERRCOD .LT. 512) GOTO 3
        WRITE (10, *) 'PS-I-ATDCOMLNK:  Attempting to '
     &             // 'detach PS 390/Host communications '
     &             // 'link.'
C
C       When we attempt to perform the Detach, use a
C       different error handler so as not to get caught
C       in a recursive loop if we consistently get an
C       error when attempting to detach.
C
        CALL PDTACH (DETERH)
        CLOSE (UNIT=10)
        IF ((ERRCOD .LT. PSFPAF) .OR.
     &      (ERRCOD .GT. PSFPPF)) GOTO 2
C
C         Identify VMS error if there was one
C
          CALL LIB$STOP (%VAL (PsVMSerr ()))
          GOTO 3
C       ELSE
    2     STOP
C       END IF
C     END IF
    3 RETURN
      END


      SUBROUTINE DETERH (ERRCOD)

C
C       Main Error handler Detach error handler:
C

      INTEGER*4   ERRCOD
      EXTERNAL    PIDCOD

      WRITE (10, *) 'PS-I-ERRWARDET:  Error/warning '
     &           // 'trying to Detach '
     &           // 'the communications'
      WRITE (10, *) 'link between the PS 390 and the host.'
      CALL PIDCOD (ERRCOD)
      RETURN
      END


      SUBROUTINE PIDCOD (ERRCOD)

C
C       PIDCOD: Identify Procedural Interface (GSR) Completion
C               code.
C

      INCLUDE    'PROCONST.FOR/NOLIST'
      INTEGER*4  ERRCOD
      CHARACTER  VMSDEF*133, PIDEF*133
      INTEGER*4  PsVMSerr
      CHARACTER  MSSG1*55, MSSG2*67
      PARAMETER  (MSSG1 = 'PS-W-UNRCOMCOD:  Procedural '
     &                 // 'Interface '
     &                 // '(GSR) completion ')
      EXTERNAL   PsVMSerr

      WRITE (10, *) 'PS-I-PROERRWAR:  Procedural '
     &           // 'Interface warning/'
     &           // 'error completion code was '
      WRITE (10, *) 'received.'
      IF (ERRCOD .NE. PSWBNC) GOTO 1
        WRITE (10, *) 'PS-W-BADNAMCHR:  Bad character '
     &             // 'in name was '
     &             // 'translated to:  "_".'
        GOTO 1000
C     ELSE
    1 IF (ERRCOD .NE. PSWNTL) GOTO 2
        WRITE (10, *) 'PS-W-NAMTOOLON:  Name too '
     &             // 'long. Name was '
     &             // 'truncated to '
        WRITE (10, *) '256 characters.'
        GOTO 1000
C     ELSE
    2 IF (ERRCOD .NE. PSWSTL) GOTO 7
        WRITE (10, *) 'PS-W-STRTOOLON:  String too '
     &             // 'long. String '
     &             // 'was truncated '
        WRITE (10, *) 'to 240 characters.'
        GOTO 1000
C     ELSE
    7 IF (ERRCOD .NE. PSWAAD) GOTO 8
        WRITE (10, *) 'PS-W-ATTALRDON:  Attach '
     &             // 'already done. '
     &             // 'Multiple call to PAttch without'
        WRITE (10, *) 'intervening PDtach call ignored.'
        GOTO 1000
C     ELSE
    8 IF (ERRCOD .NE. PSWAKS) GOTO 9
        WRITE (10, *) 'PS-W-ATNKEYSEE:  Attention key '
     &             // 'seen (depressed).'
        CALL PIBMSP
        GOTO 1000
C     ELSE
    9 IF (ERRCOD .NE. PSWBGC) GOTO 10
        WRITE (10, *) 'PS-W-BADGENCHR:  Bad generic '
     &             // 'channel character. Bad '
        WRITE (10, *) 'character in string sent via:  '
     &             // 'PPutGX  was translated to '
        WRITE (10, *) 'a blank.'
        CALL PIBMSP
        GOTO 1000
C     ELSE
   10 IF (ERRCOD .NE. PSWBSC) GOTO 11
        WRITE (10, *) 'PS-W-BADSTRCHR:  Bad '
     &             // 'character in string was '
     &             // 'translated to a blank.'
        CALL PIBMSP
        GOTO 1000
C     ELSE
   11 IF (ERRCOD .NE. PSWBPC) GOTO 12
        WRITE (10, *) 'PS-W-BADPARCHR:  Bad parser '
     &             // 'channel character. Bad '
     &             // 'character in string sent to'
        WRITE (10, *) 'PS 390 parser via:  PPutP  '
     &             // 'was translated to a blank.'
        CALL PIBMSP
        GOTO 1000
C     ELSE
   12 IF (ERRCOD .NE. PSEIMC) GOTO 13
        WRITE (10, *) 'PS-E-INVMUXCHA:  Invalid '
     &             // 'multiplexing channel '
     &             // 'specified in call to:'
        WRITE (10, *) 'PMuxCI, PMuxP, or PMuxG.'
        GOTO 1000
C     ELSE
   13 IF (ERRCOD .NE. PSEIVC) GOTO 14
        WRITE (10, *) 'PS-E-INVVECCLA:  Invalid '
     &             // 'vector list class '
     &             // 'specified'
        WRITE (10, *) 'in call to:  PVcBeg.'
        GOTO 1000
C     ELSE
   14 IF (ERRCOD .NE. PSEIVD) GOTO 15
        WRITE (10, *) 'PS-E-INVVECDIM:  Invalid '
     &             // 'vector list dimension '
     &             // 'specified in call to'
        WRITE (10, *) 'PVcBeg.'
        GOTO 1000
C     ELSE
   15 IF (ERRCOD .NE. PSEPOE) GOTO 16
        WRITE (10, *) 'PS-E-PREOPEEXP:  Prefix '
     &             // 'operator call was '
     &             // 'expected.'
        GOTO 1000
C     ELSE
   16 IF (ERRCOD .NE. PSEFOE) GOTO 17
        WRITE (10, *) 'PS-E-FOLOPEEXP:  Follow '
     &             // 'operator call was '
     &             // 'expected.'
        GOTO 1000
C     ELSE
   17 IF (ERRCOD .NE. PSELBE) GOTO 18
        WRITE (10, *) 'PS-E-LABBLKEXP:  Call to '
     &             // 'PLaAdd or PLaEnd was '
     &             // 'expected.'
        GOTO 1000
C     ELSE
   18 IF (ERRCOD .NE. PSEVLE) GOTO 19
        WRITE (10, *) 'PS-E-VECLISEXP:  Call to '
     &             // 'PVcLis or PVcEnd '
     &             // 'was expected.'
        GOTO 1000
C     ELSE
   19 IF (ERRCOD .NE. PSEAMV) GOTO 20
        WRITE (10, *) 'PS-E-ATTMULVEC:  Attempted '
     &             // 'multiple call '
     &             // 'sequence to PVcLis is NOT'
        WRITE (10, *) 'permitted for BLOCK '
     &             // 'normalized vectors.'
        GOTO 1000
C     ELSE
   20 IF (ERRCOD .NE. PSEMLB) GOTO 21
        WRITE (10, *) 'PS-E-MISLABBEG:  Missing '
     &             // 'label block begin call. '
     &             // 'Call to PLaAdd or PLaEnd'
        WRITE (10, *) 'without call to:  PLaBeg.'
        GOTO 1000
C     ELSE
```

```
   21 IF (ERRCOD .NE. PSEMVB) GOTO 22
        WRITE (10, *) 'PS-E-MISVECBEG:  Missing '
     &             // 'vector list begin '
     &             // 'call. Call to PVcLis'
        WRITE (10, *) 'or PVcEnd without call '
     &             // 'to:  PVcBeg.'
        GOTO 1000
C     ELSE
   22 IF (ERRCOD .NE. PSENUN) GOTO 23
        WRITE (10, *) 'PS-E-NULNAM:  Null name '
     &             // 'parameter is not allowed.'
        GOTO 1000
C     ELSE
   23 IF (ERRCOD .NE. PSEBCT) GOTO 24
        WRITE (10, *) 'PS-E-BADCOMTYP:  Bad '
     &             // 'comparison type operator '
     &             // 'specified in '
        WRITE (10, *) 'call to:  PIfLev.'
        GOTO 1000
C     ELSE
   24 IF (ERRCOD .NE. PSEIFN) GOTO 25
        WRITE (10, *) 'PS-E-INVFUNNAM:  Invalid '
     &             // 'function name. '
     &             // 'Attempted PS 390'
        WRITE (10, *) 'function instance failed '
     &             // 'because the named '
     &             // 'function cannot possibly'
        WRITE (10, *) 'exist. The function name '
     &             // 'identifying the '
     &             // 'function type to instance'
        WRITE (10, *) 'was longer than 256 characters.'
        GOTO 1000
C     ELSE
   25 IF (ERRCOD .NE. PSENNR) GOTO 26
        WRITE (10, *) 'PS-E-NULNAMREQ:  Null name '
     &             // 'parameter is '
     &             // 'required in operate node'
        WRITE (10, *) 'call following a PPref or '
     &             // 'PFoll procedure call.'
        GOTO 1000
C     ELSE
   26 IF (ERRCOD .NE. PSETME) GOTO 27
        WRITE (10, *) 'PS-E-TOOMANEND:  Too '
     &             // 'many END_STRUCTURE calls '
     &             // 'invoked.'
        GOTO 1000
C     ELSE
   27 IF (ERRCOD .NE. PSENOA) GOTO 28
```

```
        WRITE (10, *) 'PS-E-NOTATT:  The PS 390 '
     &             // 'communications link '
     &             // 'has not '
        WRITE (10, *) 'yet been established. '
     &             // 'PAttch has not been '
     &             // 'called or failed.'
        GOTO 1000
C     ELSE
   28 IF (ERRCOD .NE. PSEODR) GOTO 38
        WRITE (10, *) 'PS-E-OVEDURREA:  An '
     &             // 'overrun occurred during '
     &             // 'a read operation.'
        WRITE (10, *) 'The specified input buffer '
     &             // 'in call to:  PGET  '
     &             // 'or:  PGETW'
        WRITE (10, *) 'was too small and '
     &             // 'truncation has occurred.'
        GOTO 1000
C     ELSE
   38 IF (ERRCOD .NE. PSEPDT) GOTO 39
        WRITE (10, *) 'PS-E-PHYDEVTYP:  Missing '
     &             // 'or invalid physical '
     &             // 'device type'
        WRITE (10, *) 'specifier in call to PAttch.'
        CALL PVAXSP
        GOTO 1000
C     ELSE
   39 IF (ERRCOD .NE. PSELDN) GOTO 40
        WRITE (10, *) 'PS-E-LOGDEVNAM:  Missing '
     &             // 'or invalid logical '
     &             // 'device name'
        WRITE (10, *) 'specifier in call to PAttch.'
        CALL PVAXSP
        GOTO 1000
C     ELSE
   40 IF (ERRCOD .NE. PSEADE) GOTO 41
        WRITE (10, *) 'PS-E-ATTDELEXP:  Attach '
     &             // 'parameter string '
     &             // 'delimiter'
        WRITE (10, *) '"/" was expected.'
        CALL PVAXSP
        GOTO 1000
C     ELSE
   41 IF (ERRCOD .NE. PSFPAF) GOTO 42
        WRITE (10, *) 'PS-F-PHYATTFAI:  '
     &             // 'Physical attach operation '
     &             // 'failed.'
        GOTO 1000
```

```
C     ELSE
   42 IF (ERRCOD .NE. PSFPDF) GOTO 43
        WRITE (10, *) 'PS-F-PHYDETFAI:  Physical '
     &              // 'detach operation '
     &              // 'failed.'
        GOTO 1000
C     ELSE
   43 IF (ERRCOD .NE. PSFPGF) GOTO 44
        WRITE (10, *) 'PS-F-PHYGETFAI:  Physical '
     &              // 'GET operation failed.'
        GOTO 1000
C     ELSE
   44 IF (ERRCOD .NE. PSFPPF) GOTO 45
        WRITE (10, *) 'PS-F-PHYPUTFAI:  Physical '
     &              // 'PUT operation failed.'
        GOTO 1000
C     ELSE
   45 IF (ERRCOD .NE. PSFBTL) GOTO 46
        WRITE (10, *) 'PS-F-BUFTOOLAR:  Buffer '
     &              // 'too large error in '
     &              // 'call to:  PSPUT.'
        WRITE (10, *) 'This error should NEVER '
     &              // 'occur and indicates a '
     &              // 'Procedural Interface (GSR)'
        WRITE (10, *) 'internal validity check.'
        CALL PVAXSP
        GOTO 1000
C     ELSE
   46 IF (ERRCOD .NE. PSFWNA) GOTO 47
        WRITE (10, *) 'PS-F-WRONUMARG:  Wrong '
     &              // 'number of arguments '
     &              // 'in call to Procedural'
        WRITE (10, *) 'Interface (GSR) low-level '
     &              // 'I/O procedure '
     &              // '(source file:  PROIOLIB.MAR).'
        WRITE (10, *) 'This error should NEVER '
     &              // 'occur and indicates a '
     &              // 'Procedural Interface (GSR)'
        WRITE (10, *) 'internal validity check.'
        CALL PVAXSP
        GOTO 1000
C     ELSE
   47 IF (ERRCOD .NE. PSFPTL) GOTO 48
        WRITE (10, *) 'PS-F-PROTOOLAR:  Prompt '
     &              // 'buffer too large '
     &              // 'error in call to:  PSPRCV.'
        WRITE (10, *) 'This error should NEVER '
     &              // 'occur and indicates a '
     &             // 'Procedural Interface (GSR)'
        WRITE (10, *) 'internal validity check.'
        CALL PVAXSP
        GOTO 1000
C     ELSE
C
C     Unknown error message error message.
C
   48 IF (ERRCOD .GE. 512) GOTO 49
        MSSG2 = MSSG1 // 'warning'
        GOTO 51
C     ELSE
   49   IF (ERRCOD .GE. 1024) GOTO 50
          MSSG2 = MSSG1 // 'error '
          GOTO 51
C       ELSE
   50     MSSG2 = MSSG1 // 'fatal error '
C       END IF
C     END IF
   51 WRITE (10, *) MSSG2
      WRITE (10, *) 'code is unrecognized.'
      WRITE (10, *) 'Probable Procedural '
     &           // 'Interface (GSR) Internal '
     &           // 'validity check error.'
C     END IF
 1000 IF ((ERRCOD .LT. PSFPAF) .OR.
     &    (ERRCOD .GT. PSFPPF)) GOTO 2000
        CALL PSFVMSERR ( VMSdef, PIdef )
        WRITE (10, *) 'DEC VAX/VMS Error '
     &             // 'definition is:'
        WRITE (10, *)  VMSdef
        WRITE (10, *) 'Procedural Interface '
     &             // '(GSR) Interpretation of '
     &             // 'DEC VAX/VMS completion code:'
        WRITE (10, *)  PIdef
        WRITE (10, *) 'DEC VAX/VMS Error code '
     &             // 'value was: ', PsVMSerr ()
C     END IF
 2000 WRITE (10, *)
      RETURN
      END


      SUBROUTINE PIBMSP

C
C       PIBMSP:  Write the "IBM version specific"
C                message to the Error handler file.
```

```
C

       WRITE (10, *) 'This error/warning is '
      &           // 'applicable ONLY to the IBM '
      &           // 'version of the'
       WRITE (10, *) 'Procedural Interface (GSR).'
       RETURN
       END



       SUBROUTINE PVAXSP

C
C        PVAXSP:  Write the "DEC VAX/VMS Version
C                 specific" message to the Error
C                 handler file.
C
       WRITE (10, *) 'This error/warning is '
      &           // 'applicable ONLY to the DEC '
      &           // 'VAX/VMS version of'
       WRITE (10, *) 'the Procedural Interface (GSR).'
       RETURN
       END
```

Appendix B - VS FORTRAN Example Program

This appendix contains a network creation example program that illustrates the use of the PS 390/IBM VS FORTRAN Graphics Support Routines. The program contains an error handler routine example.

```
      PROGRAM BLKLEVF

      INCLUDE (PROCONSF)

C
C       MAIN PROGRAM:
C
      REAL*4    DEGRAD
      PARAMETER (DEGRAD    = 0.017453292)

      REAL*4    THETA, DTHETA, FRONT (4, 100),
     &          VECS (4, 100), ZERVEC (3),
     &          YUP (3), AT (3), FROM (3), UP (3)
      INTEGER*4 I, K, L, TIMES
      CHARACTER NAME*63
      LOGICAL*1 PFRONT (100), PVECS (100)

      EXTERNAL  ERR


      CALL PATTCH (´"´, ERR)
      AT (1) = 0.3
      AT (2) = 0
      AT (3) = 0
      FROM (1) = 0
      FROM (2) = 0
      FROM (3) = -1
      UP (1) = 0.3
      UP (2) = 1
      UP (3) = 0
      YUP (1) = 0
      YUP (2) = 1
      YUP (3) = 0
      ZERVEC (1) = 0
      ZERVEC (2) = 0
      ZERVEC (3) = 0
      CALL PINIT ( ERR )
      CALL PEYEBK ( ´EYE´, 1.0, 0.0, 0.0, 2.0, 0.0,
     &             1000.0, ´INTEN´, ERR )
```

```
      CALL PSEINT ( 'INTEN', .TRUE., 0.5, 1.0,
     &              'LOOK', ERR )
      CALL PLOOKA ( 'LOOK', AT, FROM, UP, 'PIC', ERR )
      CALL PFN    ( 'ATX', 'XVEC', ERR )
      CALL PFN    ( 'ATY', 'YVEC', ERR )
      CALL PFN    ( 'ATZ', 'ZVEC', ERR )
      CALL PFN    ( 'FROMX', 'XVEC', ERR )
      CALL PFN    ( 'FROMY', 'YVEC', ERR )
      CALL PFN    ( 'FROMZ', 'ZVEC', ERR )
      CALL PFN    ( 'AC_AT', 'ACCUMULATE', ERR )
      CALL PFN    ( 'AC_FROM', 'ACCUMULATE', ERR )
      CALL PFN    ( 'ADD_UP', 'ADDC', ERR )
      CALL PFNN   ( 'SYNC_UP', 'SYNC', 3, ERR )
      CALL PFN    ( 'FIX_SYNC', 'NOP', ERR )
      CALL PCONN  ( 'SYNC_UP', 3, 1, 'FIX_SYNC', ERR )
      CALL PCONN  ( 'FIX_SYNC', 1, 3, 'SYNC_UP', ERR )
      CALL PSNBOO ( .TRUE., 3, 'SYNC_UP', ERR )
      CALL PFN    ( 'LOOK_AT', 'LOOKAT', ERR )
      CALL PCONN  ( 'DIALS', 1, 1, 'ATX', ERR )
      CALL PCONN  ( 'DIALS', 2, 1, 'ATY', ERR )
      CALL PCONN  ( 'DIALS', 3, 1, 'ATZ', ERR )
      CALL PCONN  ( 'DIALS', 5, 1, 'FROMX', ERR )
      CALL PCONN  ( 'DIALS', 6, 1, 'FROMY', ERR )
      CALL PCONN  ( 'DIALS', 7, 1, 'FROMZ', ERR )
      CALL PCONN  ( 'ATX', 1, 1, 'AC_AT', ERR )
      CALL PCONN  ( 'ATY', 1, 1, 'AC_AT', ERR )
      CALL PCONN  ( 'ATZ', 1, 1, 'AC_AT', ERR )
      CALL PCONN  ( 'FROMX', 1, 1, 'AC_FROM', ERR )
      CALL PCONN  ( 'FROMY', 1, 1, 'AC_FROM', ERR )
      CALL PCONN  ( 'FROMZ', 1, 1, 'AC_FROM', ERR )
      CALL PCONN  ( 'AC_AT', 1, 1, 'SYNC_UP', ERR )
      CALL PCONN  ( 'AC_AT', 1, 1, 'ADD_UP',  ERR )
      CALL PCONN  ( 'ADD_UP',1, 2, 'SYNC_UP', ERR )
      CALL PCONN  ( 'SYNC_UP', 1, 1, 'LOOK_AT', ERR )
      CALL PCONN  ( 'SYNC_UP', 2, 3, 'LOOK_AT', ERR )
      CALL PCONN  ( 'AC_FROM', 1, 2, 'LOOK_AT', ERR )
      CALL PSNV3D ( AT, 2, 'AC_AT', ERR )
      CALL PSNV3D ( FROM, 2, 'AC_FROM', ERR )
      CALL PSNV3D ( YUP, 2, 'ADD_UP', ERR )
      CALL PCONN  ( 'LOOK_AT', 1, 1, 'LOOK', ERR )
      CALL PFN    ( 'FIX_AT', 'CONST', ERR )
      CALL PCONN  ( 'AC_FROM', 1, 1, 'FIX_AT', ERR )
      CALL PCONN  ( 'FIX_AT', 1, 1, 'AC_AT', ERR )
      CALL PSNV3D ( ZERVEC, 2, 'FIX_AT', ERR )
      CALL PSNV3D ( ZERVEC, 1, 'AC_FROM', ERR )
      CALL PINST  ( 'PIC','"', ERR )
      DTHETA = 10.0 * DEGRAD
```

```
 THETA  = -DTHETA
 DO I = 1, 36
   THETA = THETA + DTHETA
   CALL COMWAV (THETA, VECS, PVECS)
   DO K=1, 50
     DO L=1, 4
       FRONT (L, K) = VECS (L, (K-1)*2+1)
       PFRONT (K) = PVECS ((K-1) * 2 + 1)
     END DO
   END DO
   CALL COMNAM ( I, NAME )
   CALL PBEGS  ( NAME, ERR )
   CALL PSER   ( ´"´, 1, 35, .FALSE., I, ´"´, ERR )
   CALL PIFPHA ( ´"´, .TRUE., ´"´, ERR )
   CALL PVCBEG ( ´"´, 100, .FALSE., .FALSE., 3,
&                PVSEPA, ERR )
   CALL PVCLIS ( 100, VECS, PVECS, ERR )
   CALL PVCEND ( ERR )
   CALL PVCBEG ( ´"´, 50, .FALSE., .FALSE., 3,
&                PVCONN, ERR )
   CALL PVCLIS ( 50, FRONT, PFRONT, ERR )
   CALL PVCEND ( ERR )
   CALL PENDS ( ERR )
   CALL PINCL ( NAME, ´PIC´, ERR )
 END DO
 CALL PDISP ( ´EYE´, ERR )
 CALL PSNST ( ´X´, 1, ´DLABEL1´, ERR )
 CALL PSNST ( ´Y´, 1, ´DLABEL2´, ERR )
 CALL PSNST ( ´Z´, 1, ´DLABEL3´, ERR )
 CALL PSNST ( ´LOOK AT´, 1, ´DLABEL4´, ERR )
 CALL PSNST ( ´X´, 1, ´DLABEL5´, ERR )
 CALL PSNST ( ´Y´, 1, ´DLABEL6´, ERR )
 CALL PSNST ( ´Z´, 1, ´DLABEL7´, ERR )
 CALL PSNST ( ´FROM´, 1, ´DLABEL8´, ERR )
 CALL PDTACH ( ERR )
 END



 SUBROUTINE COMNAM (MANEID, NAME)

 INTEGER*4  NAMEID
 CHARACTER  NAME*(*)

 INTEGER*4  J, LNAME

 NAME = ´LIST000"´
```

```
      LNAME = NAMEID
      J = 7
      DO WHILE (LNAME .GT. 0)
        NAME (J:J) = CHAR (MOD (LNAME, 10) + ICHAR ('0'))
        LNAME = LNAME/10
        J = J - 1
      END DO
      RETURN
      END



      SUBROUTINE COMWAV (THETA, VECLIS, POSLIN)

      REAL*4     THETA, VECLIS (4, 100)
      LOGICAL*1  POSLIN (*)

      REAL*4     AMP, ALPHA, BETA
      PARAMETER  (AMP    = 0.8, ALPHA  = -0.02,
     &            BETA   = 0.2513274123)

      INTEGER*4  I, IADDR

      IADDR = -1
      DO I = 0, 49
        IADDR = IADDR + 2
        VECLIS (1, IADDR) = I / 50.0
        VECLIS (2, IADDR) = AMP * EXP (ALPHA * I)
     &                    * COS (THETA - BETA * I)
        VECLIS (3, IADDR) = 0
        VECLIS (4, IADDR) = 1 - I/150.0
        POSLIN (   IADDR) = .TRUE.
        VECLIS (1, IADDR+1) = VECLIS (1, IADDR)
        VECLIS (2, IADDR+1) = 0
        VECLIS (3, IADDR+1) = 0.5
        VECLIS (4, IADDR+1) = VECLIS (4, IADDR)
        POSLIN (   IADDR+1) = .TRUE.
      END DO
      RETURN
      END


C   THE FOLLOWING ERROR HANDLER DEMONSTRATES THE
C   GENERAL OVERALL RECOMMENDED FORM THAT THE USER'S
C   OWN ERROR HANDLER SHOULD FOLLOW.
C
C   THIS ERROR HANDLER UPON BEING INVOKED WRITES ALL
```

```
C   MESSAGES TO THE DATA FILE ASSOCIATED WITH THE
C   FORTRAN LOGICAL UNIT NUMBER OF 10. ERROR AND
C   WARNING EXPLANATION MESSAGES ARE WRITTEN TO
C   A DATA FILE FOR 2 REASONS:
C
C
C       1. THE ERROR HANDLER SHOULD NOT IMMEDIATELY
C           WRITE INFORMATION OUT ON THE PS 390
C           SCREEN SINCE THE EXPLANATORY TEXT
C           DEFINING THE ERROR OR WARNING CONDITION
C           MAY BE TAKEN AS DATA BY THE PS 390 AND
C           THEREFORE WIND UP NOT BEING DISPLAYED ON
C           THE PS 390 SCREEN (AS IN THE CASE OF A
C           CATASTROPHIC DATA TRANSMISSION ERROR).
C
C       2. THE LOGGING OF ERRORS AND WARNINGS TO A
C           LOGFILE ALLOWS ANY ERRORS AND/OR WARNINGS
C           TO BE REVIEWED AT A LATER TIME.
C
C
C



      SUBROUTINE ERR (ERRCOD)
C
C       PROCEDURAL INTERFACE ERROR HANDLER:
C

      INCLUDE    (PROCONSF)
      INTEGER*4  ERRCOD

    1 CALL PIDCOD (ERRCOD)
      IF (ERRCOD .LT. 512) GOTO 3
        WRITE (10, *) 'PS-I-ATDCOMLNK:  ATTEMPTING '
     &             // 'TO DETACH PS '
     &             // '300/HOST COMMUNICATIONS LINK.'
C
C       WHEN ATTEMPTING TO PERFORM THE DETACH, USE
C       A DIFFERENT ERROR HANDLER TO AVOID RECURSIVE
C       SUBROUTINE CALLS
C
        CALL PDTACH (DETERH)
        CLOSE (UNIT=10)
C
C         INVOKE TRACEBACK
C
```

```
        CALL ERRTRA
        STOP
C     END IF
    3 RETURN
      END



      SUBROUTINE DETERH (ERRCOD)
C
C       MAIN ERROR HANDLER DETACH ERROR HANDLER:
C

      INTEGER*4  ERRCOD

      WRITE (10, *) 'PS-I-ERRWARDET:  ERROR/WARNING '
     &           // 'TRYING TO DETACH '
     &           // 'THE COMMUNICATIONS'
      WRITE (10, *) 'LINK BETWEEN THE PS 300 AND '
     &           // 'THE HOST.'
      CALL PIDCOD (ERRCOD)
      RETURN
      END



      SUBROUTINE PIDCOD (ERRCOD)
C
C       PIDCOD: IDENTIFY PROCEDURAL INTERFACE
C               COMPLETION CODE.
C

      INCLUDE   (PROCONSF)
      INTEGER*4  ERRCOD
      CHARACTER  MSSG1*55, MSSG2*67
      PARAMETER  (MSSG1 = 'PS-W-UNRCOMCOD:  PROCEDURAL '
     &                 // 'INTERFACE (GSR) COMPLETION ')
      WRITE (10, *) 'PS-I-PROERRWAR:  PROCEDURAL '
     &           // 'INTERFACE WARNING/'
     &           // 'ERROR COMPLETION CODE WAS '
      WRITE (10, *) 'RECEIVED.'
      IF (ERRCOD .NE. PSWBNC) GOTO 1
        WRITE (10, *) 'PS-W-BADNAMCHR:  BAD CHARACTER '
     &             // 'IN NAME WAS '
     &             // 'TRANSLATED TO:  "_".'
        GOTO 1000
C     ELSE
```

```
    1 IF (ERRCOD .NE. PSWNTL) GOTO 2
        WRITE (10, *) ´PS-W-NAMTOOLON:  NAME TOO ´
     &             // ´LONG. NAME WAS ´
     &             // ´TRUNCATED TO ´
        WRITE (10, *) ´256 CHARACTERS.´
        GOTO 1000
C     ELSE
    2 IF (ERRCOD .NE. PSWSTL) GOTO 7
        WRITE (10, *) ´PS-W-STRTOOLON:  STRING TOO ´
     &             // ´LONG. STRING ´
     &             // ´WAS TRUNCATED ´
        WRITE (10, *) ´TO 240 CHARACTERS.´
        GOTO 1000
C     ELSE
    7 IF (ERRCOD .NE. PSWAAD) GOTO 8
        WRITE (10, *) ´PS-W-ATTALRDON:  ATTACH ´
     &             // ´ALREADY DONE. ´
     &             // ´MULTIPLE CALL TO PATTCH WITHOUT´
        WRITE (10, *) ´INTERVENING PDTACH CALL IGNORED.´
        GOTO 1000
C     ELSE
    8 IF (ERRCOD .NE. PSWAKS) GOTO 9
        WRITE (10, *) ´PS-W-ATNKEYSEE:  ATTENTION KEY ´
     &             // ´SEEN (DEPRESSED).´
        CALL PIBMSP
        GOTO 1000
C     ELSE
    9 IF (ERRCOD .NE. PSWBGC) GOTO 10
        WRITE (10, *) ´PS-W-BADGENCHR:  BAD GENERIC ´
     &             // ´CHANNEL CHARACTER. BAD ´
        WRITE (10, *) ´CHARACTER IN STRING SENT VIA:   ´
     &             // ´PPUTGX  WAS TRANSLATED TO ´
        WRITE (10, *) ´A BLANK.´
        CALL PIBMSP
        GOTO 1000
C     ELSE
   10 IF (ERRCOD .NE. PSWBSC) GOTO 11
        WRITE (10, *) ´PS-W-BADSTRCHR:  BAD ´
     &             // ´CHARACTER IN STRING WAS ´
     &             // ´TRANSLATED TO A BLANK.´
        CALL PIBMSP
        GOTO 1000
C     ELSE
   11 IF (ERRCOD .NE. PSWBPC) GOTO 12
        WRITE (10, *) ´PS-W-BADPARCHR:  BAD PARSER ´
     &             // ´CHANNEL CHARACTER. BAD ´
     &             // ´CHARACTER IN STRING SENT TO´
```

```
         WRITE (10, *) 'PS 300 PARSER VIA:  PPUTP  '
     &             // 'WAS TRANSLATED TO A BLANK.'
         CALL PIBMSP
         GOTO 1000
C       ELSE
   12 IF (ERRCOD .NE. PSEIMC) GOTO 13
         WRITE (10, *) 'PS-E-INVMUXCHA:  INVALID '
     &             // 'MULTIPLEXING CHANNEL '
     &             // 'SPECIFIED IN CALL TO:'
         WRITE (10, *) 'PMUXCI, PMUXP, OR PMUXG.'
         GOTO 1000
C       ELSE
   13 IF (ERRCOD .NE. PSEIVC) GOTO 14
         WRITE (10, *) 'PS-E-INVVECCLA:  INVALID '
     &             // 'VECTOR LIST CLASS '
     &             // 'SPECIFIED'
         WRITE (10, *) 'IN CALL TO:  PVCBEG.'
         GOTO 1000
C       ELSE
   14 IF (ERRCOD .NE. PSEIVD) GOTO 15
         WRITE (10, *) 'PS-E-INVVECDIM:  INVALID '
     &             // 'VECTOR LIST DIMENSION '
     &             // 'SPECIFIED IN CALL TO'
         WRITE (10, *) 'PVCBEG.'
         GOTO 1000
C       ELSE
   15 IF (ERRCOD .NE. PSEPOE) GOTO 16
         WRITE (10, *) 'PS-E-PREOPEEXP:  PREFIX '
     &             // 'OPERATOR CALL WAS '
     &             // 'EXPECTED.'
         GOTO 1000
C       ELSE
   16 IF (ERRCOD .NE. PSEFOE) GOTO 17
         WRITE (10, *) 'PS-E-FOLOPEEXP:  FOLLOW '
     &             // 'OPERATOR CALL WAS '
     &             // 'EXPECTED.'
         GOTO 1000
C       ELSE
   17 IF (ERRCOD .NE. PSELBE) GOTO 18
         WRITE (10, *) 'PS-E-LABBLKEXP:  CALL TO '
     &             // 'PLAADD OR PLAEND WAS '
     &             // 'EXPECTED.'
         GOTO 1000
C       ELSE
   18 IF (ERRCOD .NE. PSEVLE) GOTO 19
         WRITE (10, *) 'PS-E-VECLISEXP:  CALL TO '
     &             // 'PVCLIS OR PVCEND '
     &             // ´WAS EXPECTED.´
        GOTO 1000
C     ELSE
   19 IF (ERRCOD .NE. PSEAMV) GOTO 20
        WRITE (10, *) ´PS-E-ATTMULVEC:  ATTEMPTED ´
     &             // ´MULTIPLE CALL ´
     &             // ´SEQUENCE TO PVCLIS IS NOT´
        WRITE (10, *) ´PERMITTED FOR BLOCK ´
     &             // ´NORMALIZED VECTORS.´
        GOTO 1000
C     ELSE
   20 IF (ERRCOD .NE. PSEMLB) GOTO 21
        WRITE (10, *) ´PS-E-MISLABBEG:  MISSING ´
     &             // ´LABEL BLOCK BEGIN CALL. ´
     &             // ´CALL TO PLAADD OR PLAEND´
        WRITE (10, *) ´WITHOUT CALL TO:  PLABEG.´
        GOTO 1000
C     ELSE
   21 IF (ERRCOD .NE. PSEMVB) GOTO 22
        WRITE (10, *) ´PS-E-MISVECBEG:  MISSING ´
     &             // ´VECTOR LIST BEGIN ´
     &             // ´CALL. CALL TO PVCLIS´
        WRITE (10, *) ´OR PVCEND WITHOUT CALL ´
     &             // ´TO:  PVCBEG.´
        GOTO 1000
C     ELSE
   22 IF (ERRCOD .NE. PSENUN) GOTO 23
        WRITE (10, *) ´PS-E-NULNAM:  NULL NAME ´
     &             // ´PARAMETER IS NOT ALLOWED.´
        GOTO 1000
C     ELSE
   23 IF (ERRCOD .NE. PSEBCT) GOTO 24
        WRITE (10, *) ´PS-E-BADCOMTYP:  BAD ´
     &             // ´COMPARISON TYPE OPERATOR ´
     &             // ´SPECIFIED IN ´
        WRITE (10, *) ´CALL TO:  PIFLEV.´
        GOTO 1000
C     ELSE
   24 IF (ERRCOD .NE. PSEIFN) GOTO 25
        WRITE (10, *) ´PS-E-INVFUNNAM:  INVALID ´
     &             // ´FUNCTION NAME. ´
     &             // ´ATTEMPTED PS 300´
        WRITE (10, *) ´FUNCTION INSTANCE FAILED ´
     &             // ´BECAUSE THE NAMED ´
     &             // ´FUNCTION CANNOT POSSIBLY´
        WRITE (10, *) ´EXIST. THE FUNCTION NAME ´
     &             // ´IDENTIFYING THE ´
     &             // 'FUNCTION TYPE TO INSTANCE'
        WRITE (10, *) 'WAS LONGER THAN 256 CHARACTERS.'
        GOTO 1000
C     ELSE
   25 IF (ERRCOD .NE. PSENNR) GOTO 26
        WRITE (10, *) 'PS-E-NULNAMREQ:  NULL NAME '
     &             // 'PARAMETER IS '
     &             // 'REQUIRED IN OPERATE NODE'
        WRITE (10, *) 'CALL FOLLOWING A PPREF OR '
     &             // 'PFOLL PROCEDURE CALL.'
        GOTO 1000
C     ELSE
   26 IF (ERRCOD .NE. PSETME) GOTO 27
        WRITE (10, *) 'PS-E-TOOMANEND:  TOO '
     &             // 'MANY END_STRUCTURE CALLS '
     &             // 'INVOKED.'
        GOTO 1000
C     ELSE
   27 IF (ERRCOD .NE. PSENOA) GOTO 28
        WRITE (10, *) 'PS-E-NOTATT:  THE PS 300 '
     &             // 'COMMUNICATIONS LINK '
     &             // 'HAS NOT '
        WRITE (10, *) 'YET BEEN ESTABLISHED. '
     &             // 'PATTCH HAS NOT BEEN '
     &             // 'CALLED OR FAILED.'
        GOTO 1000
C     ELSE
   28 IF (ERRCOD .NE. PSEODR) GOTO 38
        WRITE (10, *) 'PS-E-OVEDURREA:  AN '
     &             // 'OVERRUN OCCURRED DURING '
     &             // 'A READ OPERATION.'
        WRITE (10, *) 'THE SPECIFIED INPUT BUFFER '
     &             // 'IN CALL TO:  PGET  '
     &             // 'OR:  PGETW'
        WRITE (10, *) 'WAS TOO SMALL AND '
     &             // 'TRUNCATION HAS OCCURRED.'
        GOTO 1000
C     ELSE
   38 IF (ERRCOD .NE. PSEPDT) GOTO 39
        WRITE (10, *) 'PS-E-PHYDEVTYP:  MISSING '
     &             // 'OR INVALID PHYSICAL '
     &             // 'DEVICE TYPE'
        WRITE (10, *) 'SPECIFIER IN CALL TO PATTCH.'
        CALL PVAXSP
        GOTO 1000
C     ELSE
   39 IF (ERRCOD .NE. PSELDN) GOTO 40
```

```
        WRITE (10, *) ´PS-E-LOGDEVNAM:  MISSING ´
     &             // ´OR INVALID LOGICAL ´
     &             // ´DEVICE NAME´
        WRITE (10, *) ´SPECIFIER IN CALL TO PATTCH.´
        CALL PVAXSP
        GOTO 1000
C     ELSE
   40 IF (ERRCOD .NE. PSEADE) GOTO 41
        WRITE (10, *) ´PS-E-ATTDELEXP:  ATTACH ´
     &             // ´PARAMETER STRING ´
     &             // ´DELIMITER´
        WRITE (10, *) ´"/" WAS EXPECTED.´
        CALL PVAXSP
        GOTO 1000
C     ELSE
   41 IF (ERRCOD .NE. PSFPAF) GOTO 42
        WRITE (10, *) ´PS-F-PHYATTFAI:  ´
     &             // ´PHYSICAL ATTACH OPERATION ´
     &             // ´FAILED.´
        GOTO 1000
C     ELSE
   42 IF (ERRCOD .NE. PSFPDF) GOTO 43
        WRITE (10, *) ´PS-F-PHYDETFAI:  PHYSICAL ´
     &             // ´DETACH OPERATION ´
     &             // ´FAILED.´
        GOTO 1000
C     ELSE
   43 IF (ERRCOD .NE. PSFPGF) GOTO 44
        WRITE (10, *) ´PS-F-PHYGETFAI:  PHYSICAL ´
     &             // ´GET OPERATION FAILED.´
        GOTO 1000
C     ELSE
   44 IF (ERRCOD .NE. PSFPPF) GOTO 45
        WRITE (10, *) ´PS-F-PHYPUTFAI:  PHYSICAL ´
     &             // ´PUT OPERATION FAILED.´
        GOTO 1000
C     ELSE
   45 IF (ERRCOD .NE. PSFBTL) GOTO 46
        WRITE (10, *) ´PS-F-BUFTOOLAR:  BUFFER ´
     &             // ´TOO LARGE ERROR IN ´
     &             // ´CALL TO:  PSPUT.´
        WRITE (10, *) ´THIS ERROR SHOULD NEVER ´
     &             // ´OCCUR AND INDICATES A ´
     &             // ´PROCEDURAL INTERFACE (GSR)´
        WRITE (10, *) ´INTERNAL VALIDITY CHECK.´
        CALL PVAXSP
        GOTO 1000
```

```
C     ELSE
   46 IF (ERRCOD .NE. PSFWNA) GOTO 47
        WRITE (10, *) 'PS-F-WRONUMARG:  WRONG '
     &             // 'NUMBER OF ARGUMENTS '
     &             // 'IN CALL TO PROCEDURAL'
        WRITE (10, *) 'INTERFACE (GSR) LOW-LEVEL '
     &             // 'I/O PROCEDURE '
     &             // '(SOURCE FILE:  PROIOLIB.MAR).'
        WRITE (10, *) 'THIS ERROR SHOULD NEVER '
     &             // 'OCCUR AND INDICATES A '
     &             // 'PROCEDURAL INTERFACE (GSR)'
        WRITE (10, *) 'INTERNAL VALIDITY CHECK.'
        CALL PVAXSP
        GOTO 1000
C     ELSE
   47 IF (ERRCOD .NE. PSFPTL) GOTO 48
        WRITE (10, *) 'PS-F-PROTOOLAR:  PROMPT '
     &             // 'BUFFER TOO LARGE '
     &             // 'ERROR IN CALL TO:  PSPRCV.'
        WRITE (10, *) 'THIS ERROR SHOULD NEVER '
     &             // 'OCCUR AND INDICATES A '
     &             // 'PROCEDURAL INTERFACE (GSR)'
        WRITE (10, *) 'INTERNAL VALIDITY CHECK.'
        CALL PVAXSP
        GOTO 1000
C     ELSE
C
C     UNKNOWN ERROR MESSAGE ERROR MESSAGE.
C
   48 IF (ERRCOD .GE. 512) GOTO 49
        MSSG2 = MSSG1 // 'WARNING'
        GOTO 51
C     ELSE
   49   IF (ERRCOD .GE. 1024) GOTO 50
          MSSG2 = MSSG1 // 'ERROR '
          GOTO 51
C       ELSE
   50     MSSG2 = MSSG1 // 'FATAL ERROR '
C       END IF
C     END IF
   51 WRITE (10, *) MSSG2
      WRITE (10, *) 'CODE IS UNRECOGNIZED.'
      WRITE (10, *) 'PROBABLE PROCEDURAL '
     &           // 'INTERFACE (GSR) INTERNAL '
     &           // 'VALIDITY CHECK ERROR.'
C     END IF
 1000 WRITE (10, *)
```

```
      RETURN
      END



      SUBROUTINE PIBMSP

C
C       PIBMSP:  WRITE THE "IBM VERSION SPECIFIC"
C                MESSAGE TO THE ERROR HANDLER FILE.
C

      WRITE (10, *) 'THIS ERROR/WARNING IS '
     &           // 'APPLICABLE ONLY TO THE IBM '
     &           // 'VERSION OF THE'
      WRITE (10, *) 'PROCEDURAL INTERFACE (GSR).'
      RETURN
      END



      SUBROUTINE PVAXSP

C
C       PVAXSP:  WRITE THE "DEC VAX/VMS VERSION
C                SPECIFIC" MESSAGE TO THE ERROR
C                HANDLER FILE.
C

      WRITE (10, *) 'THIS ERROR/WARNING IS '
     &           // 'APPLICABLE ONLY TO THE DEC '
     &           // 'VAX/VMS VERSION OF'
      WRITE (10, *) 'THE PROCEDURAL INTERFACE (GSR).'
      RETURN
      END
```

Appendix C - Pascal V2 Example Program

This appendix contains a network creation example program that that illustrates the use of the PS 390 DEC/VAX Pascal V2 Graphics Support Routines. The program contains an error handler routine example.

```
PROGRAM BlkLevp (INPUT, OUTPUT);

CONST
  Deg_rad = 0.017453292;
  %INCLUDE 'PROCONST.PAS'

TYPE
  %INCLUDE 'PROTYPES.PAS'

VAR
  Front          : P_VectorListType;
  Vecs           : P_VectorListType;
  Zero_vec       : P_VectorType;
  Y_Up           : P_VectorType;
  At             : P_VectorType;
  From           : P_VectorType;
  Up             : P_VectorType;
  Name           : P_VaryingType;
  Theta          : REAL;
  DTheta         : REAL;
  i              : INTEGER;
  k              : INTEGER;
  l              : INTEGER;
  Times          : INTEGER;

  %INCLUDE 'PROEXTRN.PAS'

  { The following Error Handler demonstrates the      }
  { general overall recommended form that the user's  }
  { own error handler should follow.                  }
  {                                                   }
  { This error handler upon being invoked writes ALL  }
  { messages to the data file:  'PROERROR.LOG' for 2  }
  { reasons:                                          }
  {                                                   }
  {     1. The error handler should NOT immediately   }
  {          write information out on the PS 390 screen }
  {          since the explanatory text defining the  }
  {          error or warning condition may be taken as }
{         data by the PS 390 and therefore wind up   }
{         not being displayed on the PS 390 screen   }
{         (as in the case of a catastrophic data     }
{         transmission error.                        }
{                                                    }
{     2. The logging of errors and warnings to a     }
{         logfile allows any errors and/or warnings  }
{         to be reviewed at a later time.            }


PROCEDURE Err ( Error_code: Integer );

VAR
  VMSdef, PIdef : P_VaryingType;
  Error_Log     : [STATIC] TEXT;
  ErrorFileOpen : [STATIC] BOOLEAN := FALSE;


  [EXTERNAL] PROCEDURE LIB$STOP
             (%IMMED CompletionCode : INTEGER); EXTERN;


  PROCEDURE IBM_Specific;
  BEGIN
    WRITE   (Error_Log, 'This error/warning is ');
    WRITE   (Error_Log, 'applicable ONLY to the IBM ');
    WRITELN (Error_Log, 'version of the');
    WRITELN (Error_Log, 'Procedural Interface (GSR).');
  END;


 PROCEDURE VAX_Specific;
  BEGIN
    WRITE   (Error_Log, 'This error/warning is ');
    WRITE   (Error_Log, 'applicable ONLY to the DEC ');
    WRITELN (Error_Log, 'VAX/VMS version of');
    WRITE   (Error_Log, 'the Procedural Interface ');
    WRITELN (Error_Log, '(GSR).');
  END;


  PROCEDURE UnknownError;
  BEGIN
    WRITE   (Error_Log, 'PS-W-UNRCOMCOD:  ');
    WRITE   (Error_Log, 'Procedural Interface ');
    WRITE   (Error_Log, '(GSR) completion ');
    IF Error_code < 512
    THEN WRITE (Error_Log, ´warning ´)
    ELSE IF Error_code < 1024
           THEN WRITE (Error_Log, ´error ´)
           ELSE WRITE (Error_Log, ´fatal error ´);
  WRITELN (Error_Log, ´code is unrecognized.´);
  WRITE   (Error_Log, ´Probable Procedural ´);
  WRITE   (Error_Log, ´Interface (GSR) Internal ´);
  WRITELN (Error_Log, ´validity check error.´);
END;


PROCEDURE IdentifyCompletionCode
          (Error_code : INTEGER);
BEGIN
  WRITE   (Error_Log, ´PS-I-PROERRWAR:  Procedural ´);
  WRITE   (Error_Log, ´Interface (GSR) warning/´);
  WRITE   (Error_Log, ´error completion code was ´);
  WRITELN (Error_Log, ´received.´);

  { Identify warning codes }

  IF Error_Code < 512 THEN CASE Error_Code OF
    PSW_BadNamChr:
    BEGIN
      WRITE   (Error_Log, ´PS-W-BADNAMCHR:  Bad ´);
      WRITE   (Error_Log, ´character in name was ´);
      WRITELN (Error_Log, ´translated to:  "_".´);
    END;
    PSW_NamTooLon:
    BEGIN
      WRITE   (Error_Log, ´PS-W-NAMTOOLON:  Name too ´);
      WRITE   (Error_Log, ´long. Name was truncated to ´);
      WRITELN (Error_Log, ´256 characters.´);
    END;
    PSW_StrTooLon:
    BEGIN
      WRITE   (Error_Log, ´PS-W-STRTOOLON:  String too ´);
      WRITE   (Error_Log, ´long. String was truncated ´);
      WRITELN (Error_Log, ´to 240 characters.´);
    END;
    PSW_AttAlrDon:
    BEGIN
      WRITE   (Error_Log, ´PS-W-ATTALRDON:  Attach ´);
      WRITE   (Error_Log, ´already done. Multiple call ´);
      WRITELN (Error_Log, ´to PAttach without´);
      WRITE   (Error_Log, ´intervening PDetach call ´);
      WRITELN (Error_Log, ´ignored.´);
  END;
  PSW_AtnKeySee:
  BEGIN
    WRITE   (Error_Log, ´PS-W-ATNKEYSEE:  Attention ´);
    WRITELN (Error_Log, ´key seen (depressed).´);
    IBM_Specific;
  END;
  PSW_BadGenChr:
  BEGIN
    WRITE   (Error_Log, ´PS-W-BADGENCHR:  Bad generic ´);
    WRITE   (Error_Log, ´channel character. Bad ´);
    WRITELN (Error_Log, ´character in string sent via:´);
    WRITE   (Error_Log, ´  PPutGX  was translated to ´);
    WRITELN (Error_Log, ´a blank.´);
    IBM_Specific;
  END;
  PSW_BadStrChr:
  BEGIN
    WRITE   (Error_Log, ´PS-W-BADSTRCHR:  Bad ´);
    WRITE   (Error_Log, ´character in string was ´);
    WRITELN (Error_Log, ´translated to a blank.´);
    IBM_Specific;
  END;
  PSW_BadParChr:
  BEGIN
    WRITE   (Error_Log, ´PS-W-BADPARCHR:  Bad parser ´);
    WRITE   (Error_Log, ´channel character. Bad ´);
    WRITELN (Error_Log, ´character in string sent to´);
    WRITE   (Error_Log, ´PS 300 parser via:  PPutP  ´);
    WRITELN (Error_Log, ´was translated to a blank.´);
    IBM_Specific;
  END;
  OTHERWISE UnknownError;
END

{ Identify errors }

ELSE IF Error_code < 1024 THEN CASE Error_Code OF
  PSE_InvMuxCha:
  BEGIN
    WRITE   (Error_Log, ´PS-E-INVMUXCHA:  Invalid ´);
    WRITE   (Error_Log, ´multiplexing channel ´);
    WRITELN (Error_Log, ´specified in call to:´);
    WRITELN (Error_Log, ´PMuxCI, PMuxP, or PMuxG.´);
  END;
  PSE_InvVecCla:
  BEGIN
  WRITE   (Error_Log, 'PS-E-INVVECCLA:  Invalid ');
  WRITE   (Error_Log, 'vector list class specified ');
  WRITELN (Error_Log, 'in call to:  PVecBegn.');
END;
PSE_InvVecDim:
BEGIN
  WRITE   (Error_Log, 'PS-E-INVVECDIM:  Invalid ');
  WRITE   (Error_Log, 'vector list dimension ');
  WRITELN (Error_Log, 'specified in call to');
  WRITELN (Error_Log, 'PVecBegn.');
END;
PSE_PreOpeExp:
BEGIN
  WRITE   (Error_Log, 'PS-E-PREOPEEXP:  Prefix ');
  WRITELN (Error_Log, 'operator call was expected.');
END;
PSE_FolOpeExp:
BEGIN
  WRITE   (Error_Log, 'PS-E-FOLOPEEXP:  Follow ');
  WRITELN (Error_Log, 'operator call was expected.');
END;
PSE_LabBlkExp:
BEGIN
  WRITE   (Error_Log, 'PS-E-LABBLKEXP:  Call to ');
  WRITE   (Error_Log, 'PLabAdd or PLabEnd was ');
  WRITELN (Error_Log, 'expected.');
END;
PSE_VecLisExp:
BEGIN
  WRITE   (Error_Log, 'PS-E-VECLISEXP:  Call to ');
  WRITE   (Error_Log, 'PVecList or PVecEnd was ');
  WRITELN (Error_Log, 'expected.');
END;
PSE_AttMulVec:
BEGIN
  WRITE   (Error_Log, 'PS-E-ATTMULVEC:  Attempted ');
  WRITE   (Error_Log, 'multiple call sequence to ');
  WRITELN (Error_Log, 'PVecList is NOT permitted');
  WRITELN (Error_Log, 'for BLOCK normalized vectors.');
END;
PSE_MisLabBeg:
BEGIN
  WRITE   (Error_Log, 'PS-E-MISLABBEG:  Missing ');
  WRITE   (Error_Log, 'label block begin call. ');
  WRITELN (Error_Log, 'Call to PLabAdd or PLabEnd');
  WRITELN (Error_Log, 'without call to:  PLabBegn.');
END;
```

```
PSE_MisVecBeg:
BEGIN
  WRITE   (Error_Log, ´PS-E-MISVECBEG:  Missing ´);
  WRITE   (Error_Log, ´vector list begin call. ´);
  WRITELN (Error_Log, ´Call to PVecList or PVecEnd´);
  WRITELN (Error_log, ´without call to:  PVecBegn.´);
END;
PSE_NulNam:
BEGIN
  WRITE   (Error_Log, ´PS-E-NULNAM:  Null name ´);
  WRITELN (Error_Log, ´parameter is not allowed.´);
END;
PSE_BadComTyp:
BEGIN
  WRITE   (Error_Log, ´PS-E-BADCOMTYP:  Bad ´);
  WRITE   (Error_Log, ´comparison type operator ´);
  WRITELN (Error_Log, ´specified in call to:´);
  WRITELN (Error_Log, ´PIfLevel.´);
END;
PSE_InvFunNam:
BEGIN
  WRITE   (Error_Log, ´PS-E-INVFUNNAM:  Invalid ´);
  WRITE   (Error_Log, ´function name. Attempted PS ´);
  WRITELN (Error_Log, ´300 function instance failed´);
  WRITE   (Error_Log, ´because the named function ´);
  WRITE   (Error_Log, ´cannot possibly exist. The ´);
  WRITELN (Error_Log, ´function name identifying the´);
  WRITE   (Error_Log, ´function type to instance ´);
  WRITE   (Error_Log, ´was longer than 256 ´);
  WRITELN (Error_Log, ´characters.´);
END;
PSE_NulNamReq:
BEGIN
  WRITE   (Error_Log, ´PS-E-NULNAMREQ:  Null name ´);
  WRITE   (Error_Log, ´parameter is required in ´);
  WRITELN (Error_Log, ´operate node call following´);
  WRITE   (Error_Log, ´a PPref or PFoll procedure ´);
  WRITELN (Error_Log, ´call.´);
END;
PSE_TooManEnd:
BEGIN
  WRITE   (Error_Log, ´PS-E-TOOMANEND:  Too many ´);
  WRITELN (Error_Log, ´END_STRUCTURE calls invoked.´);
END;
PSE_NotAtt:
BEGIN
  WRITE   (Error_Log, ´PS-E-NOTATT:  The PS 300 ´);
```

```
    WRITE   (Error_Log, 'communications link has not ');
    WRITELN (Error_Log, 'yet been established.');
    WRITE   (Error_Log, 'PAttach has not been called ');
    WRITELN (Error_Log, 'or failed.');
  END;
  PSE_OveDurRea:
  BEGIN
    WRITE   (Error_Log, 'PS-E-OVEDURREA:  An overrun ');
    WRITE   (Error_Log, 'occurred during a read ');
    WRITELN (Error_Log, 'operation.');
    WRITE   (Error_Log, 'The specified input buffer ');
    WRITE   (Error_Log, 'in call to:  PGET  or:  PGETW');
    WRITELN (Error_Log, ' was too small and truncation');
    WRITELN (Error_Log, 'has occurred.');
  END;
  PSE_PhyDevTyp:
  BEGIN
    WRITE   (Error_Log, 'PS-E-PHYDEVTYP:  Missing or ');
    WRITE   (Error_Log, 'invalid physical device type ');
    WRITELN (Error_Log, 'specifier in call to PAttach.');
    VAX_Specific;
  END;
  PSE_LogDevNam:
  BEGIN
    WRITE   (Error_Log, 'PS-E-LOGDEVNAM:  Missing or ');
    WRITE   (Error_Log, 'invalid logical device name ');
    WRITELN (Error_Log, 'specifier in call to PAttach.');
    VAX_Specific;
  END;
  PSE_AttDelExp:
  BEGIN
    WRITE   (Error_Log, 'PS-E-ATTDELEXP:  Attach ');
    WRITE   (Error_Log, 'parameter string delimiter ');
    WRITELN (Error_Log, '"/" was expected.');
    VAX_Specific;
  END;
  OTHERWISE UnknownError;
END

{ Identify fatal errors }

ELSE Case Error_Code OF
  PSF_PhyAttFai:
  BEGIN
    WRITE   (Error_Log, 'PS-F-PHYATTFAI:  Physical ');
    WRITELN (Error_Log, 'attach operation failed.');
  END;
```

```
PSF_PhyDetFai:
BEGIN
  WRITE   (Error_Log, 'PS-F-PHYDETFAI:  Physical ');
  WRITELN (Error_Log, 'detach operation failed.');
END;
PSF_PhyGetFai:
BEGIN
  WRITE   (Error_Log, 'PS-F-PHYGETFAI:  Physical ');
  WRITELN (Error_Log, 'get operation failed.');
END;
PSF_PhyPutFai:
BEGIN
  WRITE   (Error_Log, 'PS-F-PHYPUTFAI:  Physical ');
  WRITELN (Error_Log, 'put operation failed.');
END;
PSF_BufTooLar:
BEGIN
  WRITE   (Error_Log, 'PS-F-BUFTOOLAR:  Buffer too ');
  WRITE   (Error_Log, 'large error in call to:  ');
  WRITELN (Error_Log, 'PSPUT.');
  WRITE   (Error_Log, 'This error should NEVER ');
  WRITE   (Error_Log, 'occur and indicates a ');
  WRITELN (Error_Log, 'Procedural Interface (GSR)');
  WRITELN (Error_Log, 'validity check.');
  VAX_Specific;
END;
PSF_WroNumArg:
BEGIN
  WRITE   (Error_Log, 'PS-F-WRONUMARG:  Wrong ');
  WRITE   (Error_Log, 'number of arguments in call ');
  WRITELN (Error_Log, 'to Procedural Interface (GSR)');
  WRITE   (Error_Log, 'low-level I/O procedure ');
  WRITELN (Error_Log, '(source file:  PROIOLIB.MAR).');
  WRITE   (Error_Log, 'This error should NEVER ');
  WRITE   (Error_Log, 'occur and indicates a ');
  WRITELN (Error_Log, 'Procedural Interface (GSR) ');
  WRITELN (Error_Log, 'validity check.');
  VAX_Specific;
END;
PSF_ProTooLar:
BEGIN
  WRITE   (Error_Log, 'PS-F-PROTOOLAR:  Prompt ');
  WRITE   (Error_Log, 'buffer too large error in ');
  WRITELN (Error_Log, 'call to:  PSPRCV.');
  WRITE   (Error_Log, 'This error should NEVER ');
  WRITE   (Error_Log, 'occur and indicates a ');
  WRITELN (Error_Log, 'Procedural Interface (GSR) ');
```

```
        WRITELN (Error_Log, 'validity check.');
        VAX_Specific;
      END;
      OTHERWISE UnknownError;
    END;

    IF (Error_code >= PSF_PhyAttFai) AND
       (Error_code <= PSF_PhyPutFai) THEN BEGIN
      Psvmserr ( VMSdef, PIdef );
      WRITELN (Error_Log, 'DEC VAX/VMS Error definition is:');
      WRITELN (Error_Log,  VMSdef );
      WRITE   (Error_Log, 'Procedural Interface (GSR) ');
      WRITE   (Error_Log, 'Interpretation of ');
      WRITELN (Error_Log, 'DEC VAX/VMS completion code:');
      WRITELN (Error_Log,  PIdef );
      WRITE   (Error_Log, 'DEC VAX/VMS Error code value ');
      WRITELN (Error_Log, 'was: ', Psvmserr );
    END;
    WRITELN (Error_Log);
  END;


  PROCEDURE DetachErrorHan (Detach_Error : INTEGER);
  BEGIN
    WRITE   (Error_Log, 'PS-I-ERRWARDET:  Error/warning ');
    WRITE   (Error_Log, 'trying to Detach ');
    WRITELN (Error_Log, 'the communications link between ');
    WRITELN (Error_Log, 'the PS 300 and the host.');
    IdentifyCompletionCode (Detach_Error);
  END;


BEGIN
  IF NOT ErrorFileOpen THEN BEGIN

    { Open error file for the logging of errors }

    OPEN (Error_Log, 'Proerror.log', History := NEW);
    REWRITE (Error_Log);
    ErrorFileOpen := TRUE;
  END;
  IdentifyCompletionCode (Error_Code);
  IF Error_code >= 512 THEN BEGIN
    WRITE   (Error_Log, 'PS-I-ATDCOMLNK:  Attempting ');
    WRITE   (Error_Log, 'to detach PS 300');
    WRITELN (Error_Log, '/Host communications link.');
    { Use different error handler so as              }
    { not to get caught in a recursive               }
    { loop if we consistently get an                 }
    { error when attempting to detach                }

    PDetach (DetachErrorHan);
    CLOSE (Error_Log);
    IF (Error_code >= PSF_PhyAttFai) AND
       (Error_code <= PSF_PhyPutFai)
      { identify VMS error if there was one }

      THEN LIB$STOP (PsVMSerr)
      ELSE HALT;
  END;
END;



FUNCTION Uppercase (Chara : CHAR) : CHAR;
BEGIN
  IF (Chara >= 'a') AND (Chara <= 'z')
    THEN Uppercase := CHR (ORD (Chara) - 32)
    ELSE Uppercase := Chara;
END;



PROCEDURE Attach;

VAR
  DeviceSpec : CHAR;
  DeviceName : VARYING [5] OF CHAR;
  AttachParm : P_VaryingType;

BEGIN
  DeviceSpec := ' ';
  REPEAT
    IF DeviceSpec <> ' ' THEN
      WRITELN (OUTPUT, 'Invalid device type specified.');
    WRITE  (OUTPUT, 'Device Interface type = (PARALLEL, ');
    WRITE  (OUTPUT, 'Ethernet, Asynchronous): _');
    IF EOLN (INPUT)
      THEN DeviceSpec := ' '
      ELSE DeviceSpec := Uppercase (INPUT^);
    READLN (INPUT);
  UNTIL (DeviceSpec = 'P') OR (DeviceSpec = 'E') OR
        (DeviceSpec = 'A');
  REPEAT
    WRITE  (OUTPUT, 'Physical device name (i.e. ');
    WRITE  (OUTPUT, 'TT, TTA6, PS390): _');
    READLN (INPUT,   DeviceName);
  UNTIL LENGTH (DeviceName) > 0;
  AttachParm := 'Logdevnam=' + DeviceName + ':/Phydevtyp=';
  IF Uppercase (DeviceSpec) = 'P'
    THEN AttachParm := AttachParm + 'PARALLEL'
    ELSE IF Uppercase (DeviceSpec) = 'E'
           THEN AttachParm := AttachParm + 'Ethernet'
           ELSE AttachParm := AttachParm + 'Async';
  Pattach (AttachParm, ERR);
END;


PROCEDURE Computename (    NameId : INTEGER;
                       VAR Name   : P_VaryingType);
VAR
  j      : INTEGER;

BEGIN
  Name := 'List000';
  j := 7;
  WHILE (NameId > 0) DO BEGIN
    Name [j] := CHR (NameId MOD 10 + ORD ('0'));
    NameId := NameId DIV 10;
    j := PRED (j);
  END;
END;


PROCEDURE ComputeWave (    Theta   : REAL;
                       VAR VecList : P_VectorListType);
CONST
  Amp          =  0.8;
  Alpha        = -0.02;
  Beta         =  0.2513274123;

VAR
  i            : INTEGER;
  Addr         : INTEGER;
  Iaddr        : INTEGER;

BEGIN
  Iaddr := 0;
  FOR i := 0 TO 49 DO BEGIN
    Iaddr := SUCC (Iaddr);
```

```
      VecList [Iaddr].V4 [1] := i / 50.0;
      VecList [Iaddr].V4 [2] := Amp * EXP (Alpha * i)
                                    * COS (Theta - Beta * i);
      VecList [Iaddr].V4 [3] := 0;
      VecList [Iaddr].V4 [4] := 1 - i/150.0;
      VecList [Iaddr].Draw := TRUE;
      Iaddr := SUCC (Iaddr);
      VecList [Iaddr].V4 [1] := VecList [PRED (Iaddr)].V4 [1];
      VecList [Iaddr].V4 [2] := 0;
      VecList [Iaddr].V4 [3] := 0.5;
      VecList [Iaddr].V4 [4] := VecList [PRED (Iaddr)].V4 [4];
      VecList [Iaddr].Draw := TRUE;
    END;
  END;
BEGIN
  Attach;                       { Do the Attach }
  At.V4 [1] := 0.3;
  At.V4 [2] := 0;
  At.V4 [3] := 0;
  From.V4 [1] := 0;
  From.V4 [2] := 0;
  From.V4 [3] := -1;
  Up.V4 [1] := 0.3;
  Up.V4 [2] := 1;
  Up.V4 [3] := 0;
  Y_Up.V4 [1] := 0;
  Y_Up.V4 [2] := 1;
  Y_Up.V4 [3] := 0;
  Zero_vec.V4 [1] := 0;
  Zero_vec.V4 [2] := 0;
  Zero_vec.V4 [3] := 0;
  PInit    ( Err );
  PEyeBack ( ´eye´, 1.0, 0.0, 0.0, 2.0, 0.0,
               1000.0, ´inten´, Err );
  PSetInt  ( ´inten´, TRUE, 0.5, 1.0, ´look´, Err );
  PLookat  ( ´look´, At, From, Up, ´pic´, Err );
  PFnInst  ( ´atx´, ´xvec´, Err );
  PFnInst  ( ´aty´, ´yvec´, Err );
  PFnInst  ( ´atz´, ´zvec´, Err );
  PFnInst  ( ´fromx´, ´xvec´, Err );
  PFnInst  ( ´fromy´, ´yvec´, Err );
  PFnInst  ( ´fromz´, ´zvec´, Err );
  PFnInst  ( ´ac_at´, ´accumulate´, Err );
  PFnInst  ( ´ac_from´, ´accumulate´, Err );
  PFnInst  ( ´add_up´, ´addc´, Err );
  PFnInstN ( ´sync_up´, ´sync´, 3, Err );
  PFnInst  ( ´fix_sync´, ´nop´, Err );
```

```
PConnect ( ´sync_up´, 3, 1, ´fix_sync´, Err );
PConnect ( ´fix_sync´, 1, 3, ´sync_up´, Err );
PSndBool ( TRUE, 3, ´sync_up´, Err );
PFnInst  ( ´look_at´, ´lookat´, Err );
PConnect ( ´dials´, 1, 1, ´atx´, Err );
PConnect ( ´dials´, 2, 1, ´aty´, Err );
PConnect ( ´dials´, 3, 1, ´atz´, Err );
PConnect ( ´dials´, 5, 1, ´fromx´, Err );
PConnect ( ´dials´, 6, 1, ´fromy´, Err );
PConnect ( ´dials´, 7, 1, ´fromz´, Err );
PConnect ( ´atx´, 1, 1, ´ac_at´, Err );
PConnect ( ´aty´, 1, 1, ´ac_at´, Err );
PConnect ( ´atz´, 1, 1, ´ac_at´, Err );
PConnect ( ´fromx´, 1, 1, ´ac_from´, Err );
PConnect ( ´fromy´, 1, 1, ´ac_from´, Err );
PConnect ( ´fromz´, 1, 1, ´ac_from´, Err );
PConnect ( ´ac_at´, 1, 1, ´sync_up´, Err );
PConnect ( ´ac_at´, 1, 1, ´add_up´,  Err );
PConnect ( ´add_up´,1, 2, ´sync_up´, Err );
PConnect ( ´sync_up´, 1, 1, ´look_at´, Err );
PConnect ( ´sync_up´, 2, 3, ´look_at´, Err );
PConnect ( ´ac_from´, 1, 2, ´look_at´, Err );
PSndV3D  ( At, 2, ´ac_at´, Err );
PSndV3D  ( From, 2, ´ac_from´, Err );
PSndV3D  ( Y_up, 2, ´add_up´, Err );
PConnect ( ´look_at´, 1, 1, ´look´, Err );
PFnInst  ( ´fix_at´, ´const´, Err );
PConnect ( ´ac_from´, 1, 1, ´fix_at´, Err );
PConnect ( ´fix_at´, 1, 1, ´ac_at´, Err );
PSndV3D  ( Zero_vec, 2, ´fix_at´, Err );
PSndV3D  ( Zero_vec, 1, ´ac_from´, Err );
PInst    ( ´pic´, ´´, Err );
Dtheta := 10.0 * Deg_rad;
Theta  := -Dtheta;
FOR i := 1 TO 36 DO BEGIN
  Theta := Theta + Dtheta;
  Computewave (Theta, Vecs);
  FOR k := 1 TO 50 DO BEGIN
    FOR l := 1 TO 4 DO Front [k].V4 [l]
      := Vecs [SUCC (PRED (k) * 2)].V4 [l];
    Front [k].Draw := Vecs [SUCC (PRED (k) * 2)].Draw;
  END;
  Computename ( i, Name );
  PBegins  ( Name, Err );
  PSetR    ( ´´, 1, 35, FALSE, i, ´´, Err );
  PIfPhase ( ´´, TRUE, ´´, Err );
  PVecBegn ( ´´, 100, FALSE, FALSE, 3, P_Sepa, Err );
```

```
    PVecList ( 100, Vecs, Err );
    PVecEnd  ( Err );
    PVecBegn ( ´´, 50, FALSE, FALSE, 3, P_Conn, Err );
    PVecList ( 50, Front, Err );
    PVecEnd  ( Err );
    PEnds    ( Err );
    PIncl    ( Name, ´pic´, Err );
  END;
  PDisplay ( ´eye´, Err );
  PSndStr  ( ´X´, 1, ´Dlabel1´, Err );
  PSndStr  ( ´Y´, 1, ´Dlabel2´, Err );
  PSndStr  ( ´Z´, 1, ´Dlabel3´, Err );
  PSndStr  ( ´Look At´, 1, ´Dlabel4´, Err );
  PSndStr  ( ´X´, 1, ´Dlabel5´, Err );
  PSndStr  ( ´Y´, 1, ´Dlabel6´, Err );
  PSndStr  ( ´Z´, 1, ´Dlabel7´, Err );
  PSndStr  ( ´From´, 1, ´Dlabel8´, Err );
  Pdetach  ( Err );
END.
```

Appendix D – Pascal/VS Example Program

This appendix contains a network creation example program that illustrates the use of the PS 390/IBM PASCAL/VS Graphics Support Routines. The program contains an error handler routine example.

```
PROGRAM BlkLevp (INPUT, OUTPUT);

CONST
  Deg_rad = 0.017453292;
  %INCLUDE PROCONST

TYPE
  %INCLUDE PROTYPES

VAR
  Front          : P_VectorListType;
  Vecs           : P_VectorListType;
  Zero_vec       : P_VectorType;
  Y_Up           : P_VectorType;
  At             : P_VectorType;
  From           : P_VectorType;
  Up             : P_VectorType;
  Name           : STRING (10);
  Theta          : SHORTREAL;
  DTheta         : SHORTREAL;
  i              : INTEGER;
  k              : INTEGER;
  l              : INTEGER;
  Times          : INTEGER;


  %INCLUDE PROEXTRN


  { The following Error Handler demonstrates      }
  { the general overall recommended form that     }
  { the user´s own error handler should follow.   }
  {                                               }
  { This error handler upon being invoked         }
  { writes ALL messages to the data file          }
  { associated with the PASCAL/VS identifier of:  }
  { ´ErrorLog´. The messages are written to a     }
  { data file for two reasons:                    }
  {                                               }
{      1. The error handler should NOT          }
{          immediately write information out    }
{          on the PS 390 screen since the       }
{          explanatory text defining the error  }
{          or warning condition may be taken    }
{          as data by the PS 390 and therefore  }
{          wind up not being displayed on the   }
{          PS 390 screen (as in the case of a   }
{          catastrophic data transmission       }
{          error).                              }
{                                               }
{      2. The logging of errors and warnings    }
{          to a logfile allows any errors       }
{          and/or warnings to be reviewed at a  }
{          later time.                          }


PROCEDURE Err ( Error_code: Integer );

STATIC
  ErrorFileOpen : BOOLEAN;
  ErrorLog      : TEXT;

VALUE
  ErrorFileOpen := FALSE;



  PROCEDURE IBM_Specific;
  BEGIN
    WRITE   (ErrorLog, 'This error/warning is ');
    WRITE   (ErrorLog, 'applicable ONLY to the IBM ');
    WRITELN (ErrorLog, 'version of the');
    WRITELN (ErrorLog, 'Procedural Interface (GSR).');
  END;


  PROCEDURE VAX_Specific;
  BEGIN
    WRITE   (ErrorLog, 'This error/warning is ');
    WRITE   (ErrorLog, 'applicable ONLY to the DEC ');
    WRITELN (ErrorLog, 'VAX/VMS version of');
    WRITE   (ErrorLog, 'the Procedural Interface ');
    WRITELN (ErrorLog, '(GSR).');
  END;
```

```
PROCEDURE UnknownError;
BEGIN
  WRITE   (ErrorLog, 'PS-W-UNRCOMCOD:  ');
  WRITE   (ErrorLog, 'Procedural Interface ');
  WRITE   (ErrorLog, '(GSR) completion ');
  IF Error_code < 512
    THEN WRITE (ErrorLog, 'warning ')
    ELSE IF Error_code < 1024
           THEN WRITE (ErrorLog, 'error ')
           ELSE WRITE (ErrorLog, 'fatal error ');
  WRITELN (ErrorLog, 'code is unrecognized.');
  WRITE   (ErrorLog, 'Probable Procedural ');
  WRITE   (ErrorLog, 'Interface (GSR) Internal ');
  WRITELN (ErrorLog, 'validity check error.');
END;


PROCEDURE IdentifyCompletionCode
          (Error_code : INTEGER);
BEGIN
  WRITE   (ErrorLog, 'PS-I-PROERRWAR:  Procedural ');
  WRITE   (ErrorLog, 'Interface (GSR) warning/');
  WRITE   (ErrorLog, 'error completion code was ');
  WRITELN (ErrorLog, 'received.');

  { Identify warning codes }

  IF Error_Code < 512 THEN CASE Error_Code OF
    PSW_BadNamChr:
    BEGIN
      WRITE   (ErrorLog, 'PS-W-BADNAMCHR:  Bad ');
      WRITE   (ErrorLog, 'character in name was ');
      WRITELN (ErrorLog, 'translated to:  "_".');
    END;
    PSW_NamTooLon:
    BEGIN
      WRITE   (ErrorLog, 'PS-W-NAMTOOLON:  Name too ');
      WRITE   (ErrorLog, 'long. Name was truncated to ');
      WRITELN (ErrorLog, '256 characters.');
    END;
    PSW_StrTooLon:
    BEGIN
      WRITE   (ErrorLog, 'PS-W-STRTOOLON:  String too ');
      WRITE   (ErrorLog, 'long. String was truncated ');
      WRITELN (ErrorLog, 'to 240 characters.');
    END;
    PSW_AttAlrDon:
```

```
  BEGIN
    WRITE   (ErrorLog, ´PS-W-ATTALRDON:  Attach ´);
    WRITE   (ErrorLog, ´already done. Multiple call ´);
    WRITELN (ErrorLog, ´to PAttach without´);
    WRITE   (ErrorLog, ´intervening PDetach call ´);
    WRITELN (ErrorLog, ´ignored.´);
  END;
  PSW_AtnKeySee:
  BEGIN
    WRITE   (ErrorLog, ´PS-W-ATNKEYSEE:  Attention ´);
    WRITELN (ErrorLog, ´key seen (depressed).´);
    IBM_Specific;
  END;
  PSW_BadGenChr:
  BEGIN
    WRITE   (ErrorLog, ´PS-W-BADGENCHR:  Bad generic ´);
    WRITE   (ErrorLog, ´channel character. Bad ´);
    WRITELN (ErrorLog, ´character in string sent via:´);
    WRITE   (ErrorLog, ´  PPutGX  was translated to ´);
    WRITELN (ErrorLog, ´a blank.´);
    IBM_Specific;
  END;
  PSW_BadStrChr:
  BEGIN
    WRITE   (ErrorLog, ´PS-W-BADSTRCHR:  Bad ´);
    WRITE   (ErrorLog, ´character in string was ´);
    WRITELN (ErrorLog, ´translated to a blank.´);
    IBM_Specific;
  END;
  PSW_BadParChr:
  BEGIN
    WRITE   (ErrorLog, ´PS-W-BADPARCHR:  Bad parser ´);
    WRITE   (ErrorLog, ´channel character. Bad ´);
    WRITELN (ErrorLog, ´character in string sent to´);
    WRITE   (ErrorLog, ´PS 300 parser via:  PPutP  ´);
    WRITELN (ErrorLog, ´was translated to a blank.´);
    IBM_Specific;
  END;
  OTHERWISE UnknownError;
END

{ Identify errors }

ELSE IF Error_code < 1024 THEN CASE Error_Code OF
  PSE_InvMuxCha:
  BEGIN
    WRITE   (ErrorLog, ´PS-E-INVMUXCHA:  Invalid ´);
  WRITE   (ErrorLog, 'multiplexing channel ');
  WRITELN (ErrorLog, 'specified in call to:');
  WRITELN (ErrorLog, 'PMuxCI, PMuxP, or PMuxG.');
END;
PSE_InvVecCla:
BEGIN
  WRITE   (ErrorLog, 'PS-E-INVVECCLA:  Invalid ');
  WRITE   (ErrorLog, 'vector list class specified ');
  WRITELN (ErrorLog, 'in call to:  PVecBegn.');
END;
PSE_InvVecDim:
BEGIN
  WRITE   (ErrorLog, 'PS-E-INVVECDIM:  Invalid ');
  WRITE   (ErrorLog, 'vector list dimension ');
  WRITELN (ErrorLog, 'specified in call to');
  WRITELN (ErrorLog, 'PVecBegn.');
END;
PSE_PreOpeExp:
BEGIN
  WRITE   (ErrorLog, 'PS-E-PREOPEEXP:  Prefix ');
  WRITELN (ErrorLog, 'operator call was expected.');
END;
PSE_FolOpeExp:
BEGIN
  WRITE   (ErrorLog, 'PS-E-FOLOPEEXP:  Follow ');
  WRITELN (ErrorLog, 'operator call was expected.');
END;
PSE_LabBlkExp:
BEGIN
  WRITE   (ErrorLog, 'PS-E-LABBLKEXP:  Call to ');
  WRITE   (ErrorLog, 'PLabAdd or PLabEnd was ');
  WRITELN (ErrorLog, 'expected.');
END;
PSE_VecLisExp:
BEGIN
  WRITE   (ErrorLog, 'PS-E-VECLISEXP:  Call to ');
  WRITE   (ErrorLog, 'PVecList or PVecEnd was ');
  WRITELN (ErrorLog, 'expected.');
END;
PSE_AttMulVec:
BEGIN
  WRITE   (ErrorLog, 'PS-E-ATTMULVEC:  Attempted ');
  WRITE   (ErrorLog, 'multiple call sequence to ');
  WRITELN (ErrorLog, 'PVecList is NOT permitted');
  WRITELN (ErrorLog, 'for BLOCK normalized vectors.');
END;
PSE_MisLabBeg:
```

```
BEGIN
  WRITE   (ErrorLog, ´PS-E-MISLABBEG:  Missing ´);
  WRITE   (ErrorLog, ´label block begin call. ´);
  WRITELN (ErrorLog, ´Call to PLabAdd or PLabEnd´);
  WRITELN (ErrorLog, ´without call to:  PLabBegn.´);
END;
PSE_MisVecBeg:
BEGIN
  WRITE   (ErrorLog, ´PS-E-MISVECBEG:  Missing ´);
  WRITE   (ErrorLog, ´vector list begin call. ´);
  WRITELN (ErrorLog, ´Call to PVecList or PVecEnd´);
  WRITELN (ErrorLog, ´without call to:  PVecBegn.´);
END;
PSE_NulNam:
BEGIN
  WRITE   (ErrorLog, ´PS-E-NULNAM:  Null name ´);
  WRITELN (ErrorLog, ´parameter is not allowed.´);
END;
PSE_BadComTyp:
BEGIN
  WRITE   (ErrorLog, ´PS-E-BADCOMTYP:  Bad ´);
  WRITE   (ErrorLog, ´comparison type operator ´);
  WRITELN (ErrorLog, ´specified in call to:´);
  WRITELN (ErrorLog, ´PIfLevel.´);
END;
PSE_InvFunNam:
BEGIN
  WRITE   (ErrorLog, ´PS-E-INVFUNNAM:  Invalid ´);
  WRITE   (ErrorLog, ´function name. Attempted PS ´);
  WRITELN (ErrorLog, ´300 function instance failed´);
  WRITE   (ErrorLog, ´because the named function ´);
  WRITE   (ErrorLog, ´cannot possibly exist. The ´);
  WRITELN (ErrorLog, ´function name identifying the´);
  WRITE   (ErrorLog, ´function type to instance ´);
  WRITE   (ErrorLog, ´was longer than 256 ´);
  WRITELN (ErrorLog, ´characters.´);
END;
PSE_NulNamReq:
BEGIN
  WRITE   (ErrorLog, ´PS-E-NULNAMREQ:  Null name ´);
  WRITE   (ErrorLog, ´parameter is required in ´);
  WRITELN (ErrorLog, ´operate node call following´);
  WRITE   (ErrorLog, ´a PPref or PFoll procedure ´);
  WRITELN (ErrorLog, ´call.´);
END;
PSE_TooManEnd:
BEGIN
```

```
   WRITE   (ErrorLog, 'PS-E-TOOMANEND:  Too many ');
   WRITELN (ErrorLog, 'END_STRUCTURE calls invoked.');
 END;
 PSE_NotAtt:
 BEGIN
   WRITE   (ErrorLog, 'PS-E-NOTATT:  The PS 300 ');
   WRITE   (ErrorLog, 'communications link has not ');
   WRITELN (ErrorLog, 'yet been established.');
   WRITE   (ErrorLog, 'PAttach has not been called ');
   WRITELN (ErrorLog, 'or failed.');
 END;
 PSE_OveDurRea:
 BEGIN
   WRITE   (ErrorLog, 'PS-E-OVEDURREA:  An overrun ');
   WRITE   (ErrorLog, 'occurred during a read ');
   WRITELN (ErrorLog, 'operation.');
   WRITE   (ErrorLog, 'The specified input buffer ');
   WRITE   (ErrorLog, 'in call to:  PGET  or:  PGETW');
   WRITELN (ErrorLog, ' was too small and truncation');
   WRITELN (ErrorLog, 'has occurred.');
 END;
 PSE_PhyDevTyp:
 BEGIN
   WRITE   (ErrorLog, 'PS-E-PHYDEVTYP:  Missing or ');
   WRITE   (ErrorLog, 'invalid physical device type ');
   WRITELN (ErrorLog, 'specifier in call to PAttach.');
   VAX_Specific;
 END;
 PSE_LogDevNam:
 BEGIN
   WRITE   (ErrorLog, 'PS-E-LOGDEVNAM:  Missing or ');
   WRITE   (ErrorLog, 'invalid logical device name ');
   WRITELN (ErrorLog, 'specifier in call to PAttach.');
   VAX_Specific;
 END;
 PSE_AttDelExp:
 BEGIN
   WRITE   (ErrorLog, 'PS-E-ATTDELEXP:  Attach ');
   WRITE   (ErrorLog, 'parameter string delimiter ');
   WRITELN (ErrorLog, '"/" was expected.');
   VAX_Specific;
 END;
 OTHERWISE UnknownError;
END

   { Identify fatal errors }
```

```
ELSE Case Error_Code OF
  PSF_PhyAttFai:
  BEGIN
    WRITE   (ErrorLog, ´PS-F-PHYATTFAI:  Physical ´);
    WRITELN (ErrorLog, ´attach operation failed.´);
  END;
  PSF_PhyDetFai:
  BEGIN
    WRITE   (ErrorLog, ´PS-F-PHYDETFAI:  Physical ´);
    WRITELN (ErrorLog, ´detach operation failed.´);
  END;
  PSF_PhyGetFai:
  BEGIN
    WRITE   (ErrorLog, ´PS-F-PHYGETFAI:  Physical ´);
    WRITELN (ErrorLog, ´get operation failed.´);
  END;
  PSF_PhyPutFai:
  BEGIN
    WRITE   (ErrorLog, ´PS-F-PHYPUTFAI:  Physical ´);
    WRITELN (ErrorLog, ´put operation failed.´);
  END;
  PSF_BufTooLar:
  BEGIN
    WRITE   (ErrorLog, ´PS-F-BUFTOOLAR:  Buffer too ´);
    WRITE   (ErrorLog, ´large error in call to:  ´);
    WRITELN (ErrorLog, ´PSPUT.´);
    WRITE   (ErrorLog, ´This error should NEVER ´);
    WRITE   (ErrorLog, ´occur and indicates a ´);
    WRITELN (ErrorLog, ´Procedural Interface (GSR)´);
    WRITELN (ErrorLog, ´validity check.´);
    VAX_Specific;
  END;
  PSF_WroNumArg:
  BEGIN
    WRITE   (ErrorLog, ´PS-F-WRONUMARG:  Wrong ´);
    WRITE   (ErrorLog, ´number of arguments in call ´);
    WRITELN (ErrorLog, ´to Procedural Interface (GSR)´);
    WRITE   (ErrorLog, ´low-level I/O procedure ´);
    WRITELN (ErrorLog, ´(source file:  PROIOLIB.MAR).´);
    WRITE   (ErrorLog, ´This error should NEVER ´);
    WRITE   (ErrorLog, ´occur and indicates a ´);
    WRITELN (ErrorLog, ´Procedural Interface (GSR) ´);
    WRITELN (ErrorLog, ´validity check.´);
    VAX_Specific;
  END;
```

```
      PSF_ProTooLar:
      BEGIN
        WRITE   (ErrorLog, 'PS-F-PROTOOLAR:  Prompt ');
        WRITE   (ErrorLog, 'buffer too large error in ');
        WRITELN (ErrorLog, 'call to:  PSPRCV.');
        WRITE   (ErrorLog, 'This error should NEVER ');
        WRITE   (ErrorLog, 'occur and indicates a ');
        WRITELN (ErrorLog, 'Procedural Interface (GSR) ');
        WRITELN (ErrorLog, 'validity check.');
        VAX_Specific;
      END;
      OTHERWISE UnknownError;
    END;
    WRITELN (ErrorLog);
  END;
  PROCEDURE DetachErrorHan (Detach_Error : INTEGER);
  BEGIN
    WRITE   (ErrorLog, 'PS-I-ERRWARDET:  Error/warning ');
    WRITE   (ErrorLog, 'trying to Detach ');
    WRITELN (ErrorLog, 'the communications link between ');
    WRITELN (ErrorLog, 'the PS 300 and the host.');
    IdentifyCompletionCode (Detach_Error);
  END;

BEGIN
  IF NOT ErrorFileOpen THEN BEGIN

    { Open error file for the logging of errors }

    REWRITE (ErrorLog);
    ErrorFileOpen := TRUE;
  END;
  IdentifyCompletionCode (Error_Code);
  WRITE   (ErrorLog, 'PS-I-PASTRABAC:  PASCAL/VS ');
  WRITELN (ErrorLog, 'Traceback follows:');

  { Display PASCAL/VS traceback }

  TRACE (ErrorLog);
  WRITELN (ErrorLog);
  IF Error_code >= 512 THEN BEGIN
    WRITE   (ErrorLog, 'PS-I-ATDCOMLNK:  ');
    WRITE   (ErrorLog, 'Attempting to detach PS 300');
    WRITELN (ErrorLog, '/Host communications link.');

    { Use different error handler so as        }
    { not to get caught in a recursive         }
    { loop if we consistently get an          }
    { error when attempting to detach         }

    PDetach (DetachErrorHan);
    CLOSE (ErrorLog);
    HALT;                                 {stop}
  END;
END;


PROCEDURE Computename (    NameId : INTEGER;
                       VAR Name   : STRING);
VAR
  j      : INTEGER;

BEGIN
  Name := 'List000';
  j := 7;
  WHILE (NameId > 0) DO BEGIN
    Name (.j.) := CHR (NameId MOD 10 + ORD ('0'));
    NameId := NameId DIV 10;
    j := PRED (j);
  END;
END;


PROCEDURE ComputeWave (    Theta   : SHORTREAL;
                       VAR VecList : P_VectorListType);
CONST
  Amp         =  0.8;
  Alpha       = -0.02;
  Beta        =  0.2513274123;

VAR
  i           : INTEGER;
  Addr        : INTEGER;
  Iaddr       : INTEGER;

BEGIN
  Iaddr := 0;
  FOR i := 0 TO 49 DO BEGIN
    Iaddr := SUCC (Iaddr);
    VecList (.Iaddr.).V4 (.1.) := i / 50.0;
    VecList (.Iaddr.).V4 (.2.) := Amp * EXP (Alpha * i)
                                  * COS (Theta - Beta * i);
    VecList (.Iaddr.).V4 (.3.) := 0;
    VecList (.Iaddr.).V4 (.4.) := 1 - i/150.0;
```

```
      VecList (.Iaddr.).Draw := TRUE;
      Iaddr := SUCC (Iaddr);
      VecList (.Iaddr.).V4 (.1.) := VecList (.PRED (Iaddr).).V4 (.1.);
      VecList (.Iaddr.).V4 (.2.) := 0;
      VecList (.Iaddr.).V4 (.3.) := 0.5;
      VecList (.Iaddr.).V4 (.4.) := VecList (.PRED (Iaddr).).V4 (.4.);
      VecList (.Iaddr.).Draw := TRUE;
    END;
  END;


BEGIN
  PAttach ('', Err );           { Do the Attach }
  At.V4 (.1.) := 0.3;
  At.V4 (.2.) := 0;
  At.V4 (.3.) := 0;
  From.V4 (.1.) := 0;
  From.V4 (.2.) := 0;
  From.V4 (.3.) := -1;
  Up.V4 (.1.) := 0.3;
  Up.V4 (.2.) := 1;
  Up.V4 (.3.) := 0;
  Y_Up.V4 (.1.) := 0;
  Y_Up.V4 (.2.) := 1;
  Y_Up.V4 (.3.) := 0;
  Zero_vec.V4 (.1.) := 0;
  Zero_vec.V4 (.2.) := 0;
  Zero_vec.V4 (.3.) := 0;
  PInit    ( Err );
  PEyeBack ( 'eye', 1.0, 0.0, 0.0, 2.0, 0.0,
             1000.0, 'inten', Err );
  PSetInt  ( 'inten', TRUE, 0.5, 1.0, 'look', Err );
  PLookat  ( 'look', At, From, Up, 'pic', Err );
  PFnInst  ( 'atx', 'xvec', Err );
  PFnInst  ( 'aty', 'yvec', Err );
  PFnInst  ( 'atz', 'zvec', Err );
  PFnInst  ( 'fromx', 'xvec', Err );
  PFnInst  ( 'fromy', 'yvec', Err );
  PFnInst  ( 'fromz', 'zvec', Err );
  PFnInst  ( 'ac_at', 'accumulate', Err );
  PFnInst  ( 'ac_from', 'accumulate', Err );
  PFnInst  ( 'add_up', 'addc', Err );
  PFnInstN ( 'sync_up', 'sync', 3, Err );
  PFnInst  ( 'fix_sync', 'nop', Err );
  PConnect ( 'sync_up', 3, 1, 'fix_sync', Err );
  PConnect ( 'fix_sync', 1, 3, 'sync_up', Err );
  PSndBool ( TRUE, 3, 'sync_up', Err );
```

```
PFnInst  ( ´look_at´, ´lookat´, Err );
PConnect ( ´dials´, 1, 1, ´atx´, Err );
PConnect ( ´dials´, 2, 1, ´aty´, Err );
PConnect ( ´dials´, 3, 1, ´atz´, Err );
PConnect ( ´dials´, 5, 1, ´fromx´, Err );
PConnect ( ´dials´, 6, 1, ´fromy´, Err );
PConnect ( ´dials´, 7, 1, ´fromz´, Err );
PConnect ( ´atx´, 1, 1, ´ac_at´, Err );
PConnect ( ´aty´, 1, 1, ´ac_at´, Err );
PConnect ( ´atz´, 1, 1, ´ac_at´, Err );
PConnect ( ´fromx´, 1, 1, ´ac_from´, Err );
PConnect ( ´fromy´, 1, 1, ´ac_from´, Err );
PConnect ( ´fromz´, 1, 1, ´ac_from´, Err );
PConnect ( ´ac_at´, 1, 1, ´sync_up´, Err );
PConnect ( ´ac_at´, 1, 1, ´add_up´,  Err );
PConnect ( ´add_up´,1, 2, ´sync_up´, Err );
PConnect ( ´sync_up´, 1, 1, ´look_at´, Err );
PConnect ( ´sync_up´, 2, 3, ´look_at´, Err );
PConnect ( ´ac_from´, 1, 2, ´look_at´, Err );
PSndV3D  ( At, 2, ´ac_at´, Err );
PSndV3D  ( From, 2, ´ac_from´, Err );
PSndV3D  ( Y_up, 2, ´add_up´, Err );
PConnect ( ´look_at´, 1, 1, ´look´, Err );
PFnInst  ( ´fix_at´, ´const´, Err );
PConnect ( ´ac_from´, 1, 1, ´fix_at´, Err );
PConnect ( ´fix_at´, 1, 1, ´ac_at´, Err );
PSndV3D  ( Zero_vec, 2, ´fix_at´, Err );
PSndV3D  ( Zero_vec, 1, ´ac_from´, Err );
PInst    ( ´pic´, ´´, Err );
Dtheta := 10.0 * Deg_rad;
Theta  := -Dtheta;
FOR i := 1 TO 36 DO BEGIN
  Theta := Theta + Dtheta;
  Computewave (Theta, Vecs);
  FOR k := 1 TO 50 DO BEGIN
    FOR l := 1 TO 4 DO Front (.k.).V4 (.l.)
      := Vecs (.SUCC (PRED (k) * 2).).V4 (.l.);
    Front (.k.).Draw := Vecs (.SUCC (PRED (k) * 2).).Draw;
  END;
  Computename ( i, Name );
  PBegins  ( Name, Err );
  PSetR    ( ´´, 1, 35, FALSE, i, ´´, Err );
  PIfPhase ( ´´, TRUE, ´´, Err );
  PVecBegn ( ´´, 100, FALSE, FALSE, 3, P_Sepa, Err );
  PVecList ( 100, Vecs, Err );
  PVecEnd  ( Err );
  PVecBegn ( ´´, 50, FALSE, FALSE, 3, P_Conn, Err );
```

```
    PVecList ( 50, Front, Err );
    PVecEnd  ( Err );
    PEnds    ( Err );
    PIncl    ( Name, 'pic', Err );
  END;
  PDisplay ( 'eye', Err );
  PSndStr  ( 'X', 1, 'Dlabel1', Err );
  PSndStr  ( 'Y', 1, 'Dlabel2', Err );
  PSndStr  ( 'Z', 1, 'Dlabel3', Err );
  PSndStr  ( 'Look At', 1, 'Dlabel4', Err );
  PSndStr  ( 'X', 1, 'Dlabel5', Err );
  PSndStr  ( 'Y', 1, 'Dlabel6', Err );
  PSndStr  ( 'Z', 1, 'Dlabel7', Err );
  PSndStr  ( 'From', 1, 'Dlabel8', Err );
  Pdetach  ( Err );
END.
```

TT4. FUNCTION NETWORK EDITOR

TT4. FUNCTION NETWORK EDITOR

NETEDIT

CONTENTS

ILLUSTRATIONS

Section TT4

Function Network Editor

NETEDIT

This software package is distributed by Evans & Sutherland as a convenience to customers and as an aid to understanding the capabilities of the PS 390 graphics systems. Evans & Sutherland Customer Engineering supports the package to the extent of answering questions concerning installation and operation of the programs, as well as receiving reports on any bugs encountered while the programs are running. However, Evans & Sutherland makes no commitment to correct any errors which may be found.

The NETEDIT Function Network Editor is a program to aid in the creation of PS 390 function networks. Networks are created as diagrams using a drawing program with menu selections. Symbols representing functions are placed in the diagram and their inputs and outputs are connected much as in a wiring diagram. Constants and variables can be specified. Items can be named and annotations can be added freely. When the diagram is complete, the Editor allows you to generate a file of the corresponding PS 390 ASCII commands and comments or of FORTRAN or Pascal Graphics Support Routines (GSRs). Hardcopies of diagrams can be obtained with the PS 390 system Writeback feature (refer to Section *TT9 Transformed Data and Writeback*).

1. Introduction to the Function Network Editor

The Network Editor currently runs under VAX VMS 3.3 and higher, uses Pascal V2.2 and higher source code, and uses version A2 and higher of the PS 390 firmware. Files are distributed on magnetic tape and are installed as explained in *Appendix A*.

Command files display menus which let you start the Editor and restart if a crash occurs. A log file is kept each time the Editor is started and this is used in recovery. A parameter file can be created to specify user-definable options, such as directory names and file extensions.

1.1 Editing a File

NETEDIT stores the diagram of the networks as a hierarchical data structure in a sequential file. It allows single files to represent extended function networks with external contact points to other function networks or nodes in a display structure. It also allows you to use macros (references to libraries of other networks) and user-written functions.

You can edit a file by making menu selections with the data tablet or in some cases with the function keys. Selections let you place items in the display area to create the network drawing, or change the drawing as needed. Other selections display HELP information, access other files, and generate ASCII or GSR files from the network diagram.

1.1.1 Network Diagram Primitives

Intrinsic functions, initial function instances, user-written functions, and macros are represented as boxes with numbered inputs and outputs. Functions are selected and placed in the display area and named using the Labels selection. This results in **name:=F:Function_name;** statements or equivalent GSR calls in the code file that is generated.

Connections corresponding to **CONNECT name<i>:<j>name;** commands and equivalent GSRs are made by routing arcs from one connection point to another. Connecting arcs are shown as lines much like wires in a wiring diagram. A connector is an arc endpoint. It may be an input queue to a function and so part of the function box, or one of several free-floating types of endpoints.

Constant connectors can be placed in the diagram and connected to function inputs. The value associated with the constant is entered also. This results in **SEND value TO <i>:name;** statements or equivalent GSR calls in the code file.

Variables are created as connectors also. These correspond to instances of the **VARIABLE name1;** command or equivalent GSR calls.

1.1.2 Constructing the Diagram

Since the display area is limited and networks are often quite extensive, most diagrams will be broken up into pages. The Editor allows you to construct a diagram hierarchically by creating a "frame" for each page and by letting you create "detail frames," which represent lower pages in the hierarchy.

Detail frames are shown as pseudo-3D boxes with inputs and outputs. They represent different functional blocks of a network. For example, the parts of a network which handle input from the dials can be shown as a detail frame within a page that shows a general network of peripherals and display manipulation. When you move into that detail frame the actual functions which comprise the detail will be shown. Details can be nested to any level.

The hierarchical nature of the network diagram means you can create a network top down or bottom-up. Detail frames can be created first and their contents specified later, or parts of the diagram can be moved into or deleted from detail frames. The diagram can be constructed and restructured however you want. You navigate between frames using function keys.

1.1.3 Generating the PS 390 Command File

When the diagram is completed, selections from the menus allow you to generate a file of PS 390 ASCII commands or of FORTRAN or Pascal GSRs which instance the functions, connect inputs and outputs, declare variables, and send data as shown in the diagram.

A sample ASCII file generated by the Editor is included in *Appendix B*.

2. Getting Started

The Network Editor is started and entered through menu selections displayed by a command file. After the Network Editor and associated files have been installed and the command files NETUSER.COM and NETBUILD.COM have been customized (refer to *Appendix A*), enter the following command:

```
$ @[HomeDir]NETUSER
```

For [HomeDir], substitute the name of the directory in which NETEDIT resides. This command file brings up the following Initial Menu.

```
          Evans & Sutherland PS 300 Utilities V1.08
                        Initial Menu

    0) Exit
    1) Initialize the PS 300
    2) Send a file to the PS 300
    3) Run NetProbe - Function Network Debugger (Menu)
    4) Run NetEdit - Function Network Editor (Menu)
    5) Character Font Utilities (Menu)
```

Select option 4 to bring up the following NETEDIT Menu of options specific to running the Network Editor.

```
          Evans & Sutherland Function Network Editor
              Maintenance Command File V1.08

          NetEdit: PS 300 Function Network Editor Menu

    0) Exit
    1) Start NetEdit from scratch, download support net
    2) Start NetEdit without full init, but download support net
    3) Restart NetEdit without downloading support net
    4) Read the current release notes
    5) Start NetEdit without full init from floppy disk
    6) Init from floppy
```

Selection 1 initializes the PS 390 and loads NETEDIT. This is the selection most often made when the Editor is run. Selection 2 loads NETEDIT without initializing the PS 390. Selection 3 restarts the Editor after a crash or an aborted session. Selection 4 lets you review the current release notes. Selection 5 allows you to load NETEDIT from a floppy diskette. Selection 6 initializes the PS 390 from a floppy diskette. This must be done if NETEDIT was loaded from a floppy diskette.

To start NETEDIT for the first time, use selection 1 or 2. When the Editor display appears, pick SELECT NETWORK and you will be prompted for the name of the file you want to edit.

A sample editing session is included in *Appendix B*. You may wish to glance through this before reading the following sections.

2.1 Restarting

Should the program crash while you are editing or should you deliberately abort the current session using CTRL C, the network editor may be restarted without reloading the support network and display structures by using the menu selections or typing the following command.

```
$ @[HomeDir]NETUSER  4  3
```

For [HomeDir], substitute the name of the directory in which NETEDIT resides. The parameters 4 3 make the menu selections for you. Note that all selections from the command file menus can be given as parameters to bypass the menu displays.

2.2 Parameter File

A parameter file permits each user to customize the Editor by describing a working set of directories and selecting some options.

Create a parameter file called NETPARMS.TXT. In this file, list the directories (up to 30) that you want to have in your working set. List them in order of preference, since the directories will be searched in this order.

The parameter file can also contain other operating parameters. Currently, these consist of the following.

@EXTENSION   .300

This sets the ASCII output file extension to .300, and may be changed to any other extension. GSR output files have the extension .FOR for FORTRAN and .PAS for Pascal. These extensions cannot be changed in the parameter file.

@SYSTEMPRIVILEGE

The Editor is set by default to use the user/primitive function data base but may be changed to use system privileged functions.

@PRIMITIVEPROMPT ON/OFF

Enables/disables prompting for function names immediately as they are instanced and placed.

@ATTACHTO

This specifies a parameter which is passed to the GSRs when the Editor starts up. The parameter contains the logical device name and physical device type (asynchronous, parallel). The format of the parameter is exactly what the PATTACH GSR expects:

```
LOGDEVNAM=name/PHYDEVTYP=type
```

For more information consult the PATTACH description in the PS 390 DEC VAX/VMS Pascal GSR summary.

The parameter file is expected to reside in the directory NETUSERDIR. You must make this logical assignment either manually or by inserting the following line into your login or similar file.

```
@ASSIGN    [UserHomeDir]    NETUSERDIR:
```

[UserHomeDir] should be replaced with the directory in which you keep your parameter file. This ensures that the Editor can find your parameter file from wherever it may be run.

3. General Characteristics

3.1 Display Organization

The Editor display is divided into three sections.

The main section is the diagram DISPLAY AREA in which you assemble and edit the network diagram. This is made up of two parts: a header bar, which describes the frame, and a work area.

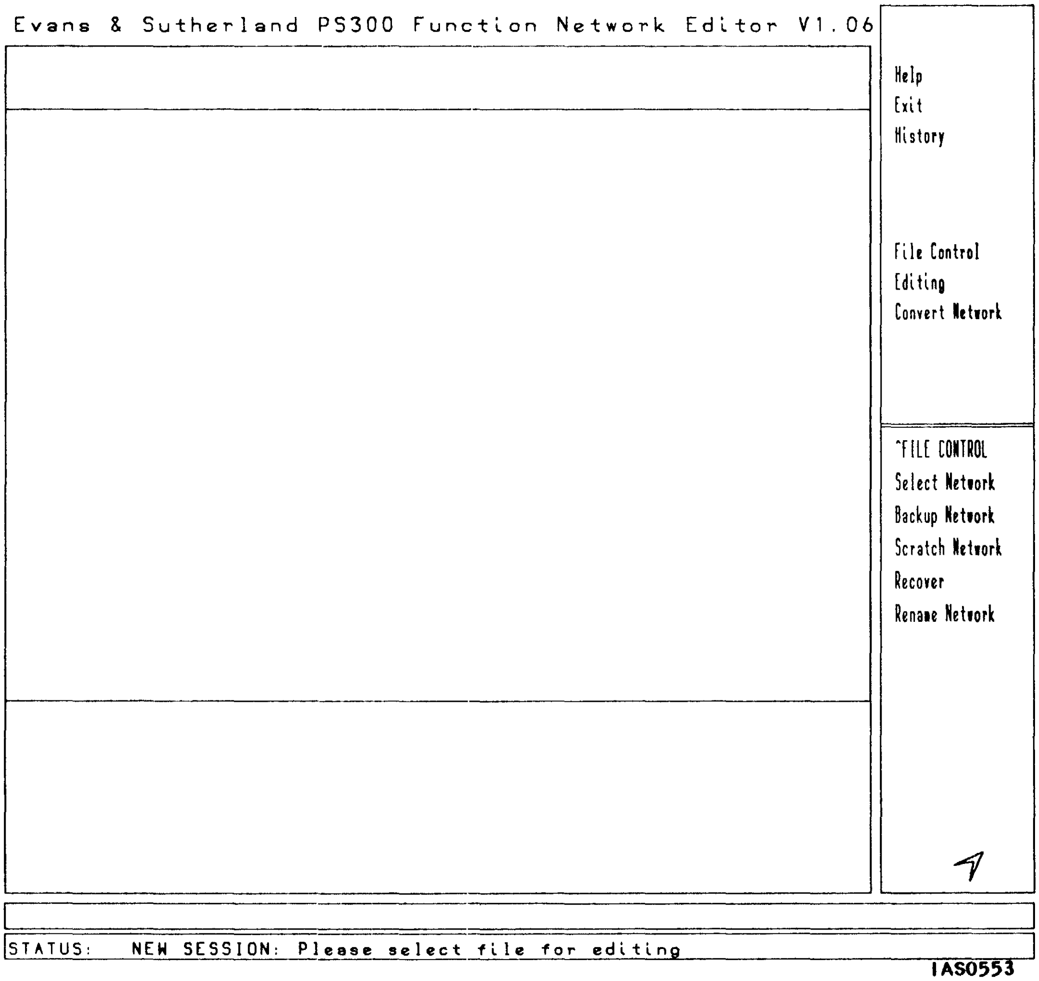
The header bar includes the name and prefix of the current frame, the file name, a page number, the total number of pages, and the date the file was last modified. The name and prefix in the header can be modified by picking the item from the bar and entering a new value.

The work area is an oblong of a size which allows hardcopy to fit neatly on 8 1/2 by 11 inch paper. Panning and zooming using the Control Dials can be performed in the display area. The header bar is unaffected by panning and zooming.

On the right edge of the display is the MENU AREA in which the different Editor menus are displayed. Up to three menus may be present at a time, depending on where you are in the hierarchy of menu options.

At the bottom of the screen is the MESSAGE AREA, two lines in which messages are displayed. The top line serves as a PROMPT and text entry line, and the second line displays warning and STATUS messages. The abbreviations I - information, W - warning, E - error are used to indicate the relative severity of the message.

Figure 4–1 shows the initial Network Editor display.

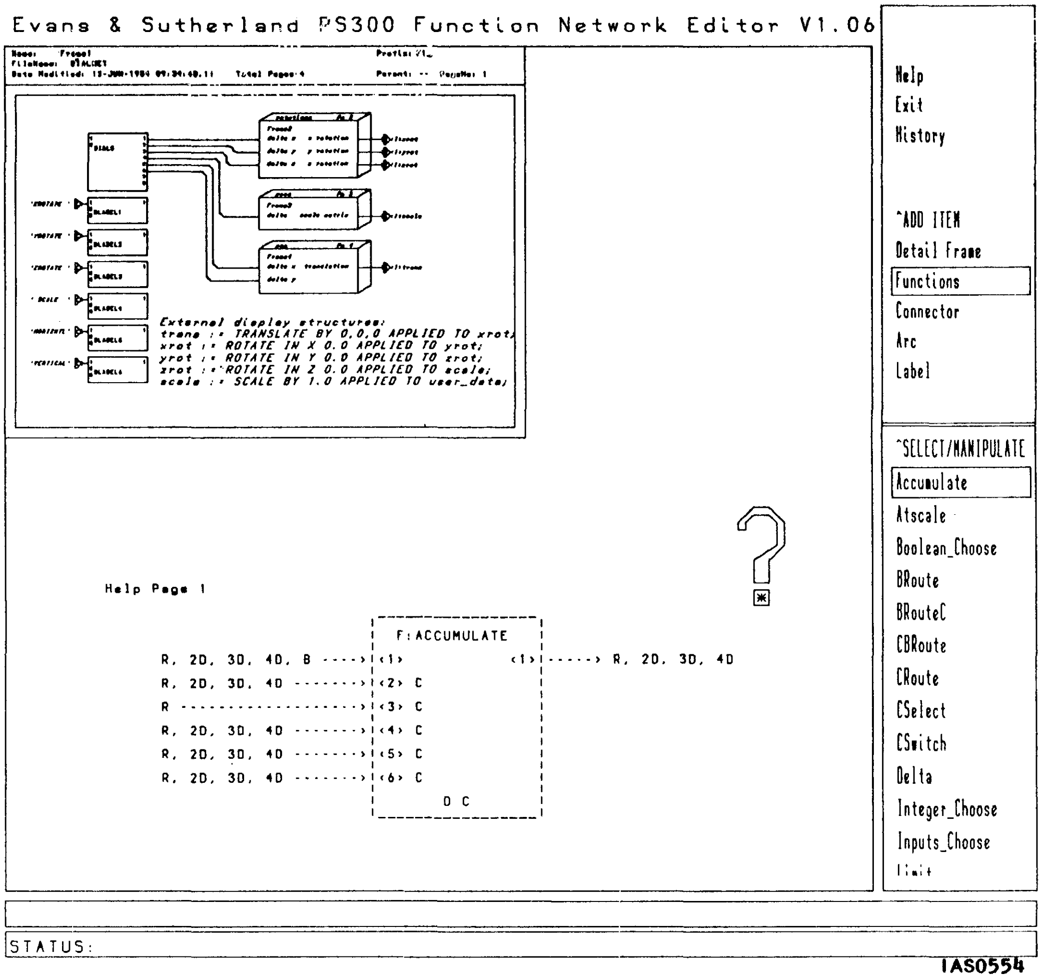


STATUS: NEW SESSION: Please select file for editing

Figure 4-1. Function Network Editor Display

Function key F1, the VIEWMENU function key, is used to alternate between the diagram/menu display for editing and the diagram only display for hardcopy and closer inspection.

The HELP and HISTORY selections also change the display. When these functions are chosen, the display is as shown in Figure 4-2



Evans & Sutherland PS300 Function Network Editor V1.06

```
Name:    Frame1                                          Prefix: V1_
FileName:  DIALSET
Date Modified: 13-JUN-1984 09:34:40.11    Total Pages:4  Parent: --  PageNo: 1
```

```
External display structures:
trans := TRANSLATE BY 0,0,0 APPLIED TO xrot;
xrot := ROTATE IN X 0.0 APPLIED TO yrot;
yrot := ROTATE IN Y 0.0 APPLIED TO zrot;
zrot := ROTATE IN Z 0.0 APPLIED TO scale;
scale := SCALE BY 1.0 APPLIED TO user_data;
```

Help Page 1

```
                         ---------------------
                        |                     |
                        |   F:ACCUMULATE      |
R, 2D, 3D, 4D, B ---->| <1>               <1> |-----> R, 2D, 3D, 4D
R, 2D, 3D, 4D ------->| <2>  C                |
R ------------------->| <3>  C                |
R, 2D, 3D, 4D ------->| <4>  C                |
R, 2D, 3D, 4D ------->| <5>  C                |
R, 2D, 3D, 4D ------->| <6>  C                |
                        |        D C          |
                         ---------------------
```

STATUS:

Figure 4-2. The HELP and HISTORY Display

3.2 Cursor Shapes

You interact with the display through a combination of tablet and keyboard actions. The cursor shows not only the current location at which you are pointing but also the current state of the program by changing the cursor shape for different actions. The most basic cursor shapes indicate when only a menu selection is permitted (a chevron), when no action is yet permitted (hourglass), when keyboard entry is permitted (downward pointing hand), and when an object may be placed, moved, or deleted (various shapes). A clock shape with sweeping arm appears for extended periods of waiting. This will help you judge the progress of the operation.

With most cursor shapes, an asterisk (*) indicates exactly at what point on the shape the stylus tip is, which is the point at which object or menu "picking" is performed. Where an asterisk is not present, as with the arrow cursor shapes, the tip of the arrow corresponds to the point of the stylus.

Cursor shapes are described in the course of the documentation as appropriate. An optional cross-hair may be displayed at the cursor position by toggling function key F2. This cross-hair is useful for aligning objects on the display.

3.3 Menu Selections

The MENU AREA is divided into three main menus: the PERMANENT MENU (HELP, EXIT, HISTORY); the MAIN MENU (editing selections and further options); and the SUBMENU (object categories, file options) which appear as needed. The permanent menu is always present and may always be selected from. When a permanent menu option has been invoked, that option is highlighted. All cursor shapes except the hourglass or clock may be used to select from the menu at any time. Any incomplete action is canceled by making another selection. This includes keyboard entry and object placement.

The main and submenus are arranged as a hierarchy which will sometimes display two different levels (MAIN and SUBMENU) and at other times just one level (MAIN). You move from menu to menu by picking selections with the data tablet and stylus or pressing certain function keys. The first item in all but the top level menu is in capital letters and preceded by a chevron (^) to signify that it is both the title of the menu and the entry point to move back up. Selecting it will reset the menu display accordingly.

Many submenus are particularly long. When a submenu is displayed it may be scrolled up or down by means of the first dial on the control dials unit. At the bottom of the submenu is a long string of dashes to indicate that you have moved off the bottom and that you should scroll upwards to find the submenu.

The menu hierarchy is as follows.

```
HELP
EXIT
HISTORY

FILE CONTROL
    SELECT NETWORK
    BACKUP NETWORK
    SCRATCH NETWORK
    RECOVER
    RENAME NETWORK

EDITING
    ADD ITEM
        Detail Frame
        Functions
        Connector
            Input Frame
            Output Frame
            Constants
            Variable
            In-External
            Out-External
        Arc
        Labels
    MOVE
    MOVE AREA
    DELETE
    DELETE AREA
    OPTIONS
        Change Scale
        Redraw Frame
        Replace Functions
        Update Macros
        Print Page
        Print Page Set

CONVERT NETWORK
    ASCII OUTPUT
    FORTRAN GSR
    PASCAL GSR
    USE FRAME PREFIX
    USE MACRO PREFIX
    COMPILE MACRO
    SUPPRESS COMMENTS
```

3.4 Permanent Menu Items

HELP

The HELP selection provides information on individual functions, menu selections, and a variety of other topics. To get information on menu selections, select HELP and then pick the menu item. To get help on a function, pick EDITING, then ADD ITEM, then Functions, and pick the name of the function you are interested in. The scrolling dial, Dial 3, can be used to scan forwards and backwards through long descriptions.

EXIT

EXIT saves any existing network that has been edited, closes all open files, returns the keyboard to terminal emulator mode, and exits from the program. If the file name is incorrect, EXIT will not allow you to leave the program. When this occurs, you must either scratch or rename the network and select EXIT again. Note that EXIT must be picked twice before it is selected.

HISTORY

This selection allows you to view the last ten pages of status messages. This can be useful when a code conversion produces errors and the messages have moved past faster than they could be read. The scrolling dial, Dial 3, can be used to scan forwards and backwards through the pages.

3.5 Function Keys

Currently, 11 of the 12 function keys are programmed to perform specific operations. Most keys perform only one function, but keys F8, F9, and F10 have double functions. The keys are programmed as follows.

Key F1 - VIEWMENU

Changes the display for closer inspection and for hardcopy of diagrams. Removes the MENU area and MESSAGE area and displays just the diagram at a size that produces 8 1/2 by 11 inch hardcopies.

Key F2 - CROSS

Displays a cross-hair to help place objects in the diagram.

Key F3 - GO UP

Moves you up one level in the diagram hierarchy from the current frame (context) to its parent frame. The frame you were just in appears as a detail frame in the new display. If you are in the top frame, you are notified that you cannot move higher.

Key F4 - OUTLINE

Displays a page which shows the structure of the diagram file. All frames in the file are listed, and indentation shows the hierarchical dependencies. The frame currently being edited is highlighted. The outline can be scrolled using Dial 2. You may also pick a frame in the outline and proceed directly to that frame without going through the intermediate frames.

Key F5 - GO DOWN

Moves you down to a detail frame in the context frame you are currently editing. If more than one detail frame is present, a large down-pointing arrow is displayed to allow you to select the detail frame you want to enter.

Key F6 - FULL VIEW

Resets the display after zooming and panning has taken place with the control dials.

Key F7 - BY NAME

Allows you to select an intrinsic function, initial function instance, macro, or user-written function by name. Press this key and then enter the name at the PROMPT line at the bottom of the screen. For primitive functions, you may place multiple copies before selecting another primitive. For macros and user-written functions, you are prompted after each placement.

Key F8 - MOVE (double function)

Allows you to select MOVE or MOVE AREA without picking from the menu. One press selects MOVE, two presses select MOVE AREA.

Key F9 - DELETE (double function)

Allows you to select DELETE or DELETE AREA without picking from the menu. One press selects DELETE, two presses select DELETE AREA.

Key F10 - ARC/TEXT (double function)

Allows you to place an arc or edit labels without picking from the menu. One press selects ARC, two presses select LABELS.

Key F11

This key is currently unused.

Key F12

This key is currently unused.

3.6 Control Dials

The Editor uses 6 of the 8 Control Dials to help in building and viewing network diagrams. The dials are programmed as follows.

Dial 1 - SUBMENU

Scrolls a submenu up and down.

Dial 2 - OUTLINE

Scrolls the diagram outline page forwards and backwards.

Dial 3 - FLIPPAGE

Scans forwards and backwards through HELP or HISTORY pages.

Dial 4

This dial is currently unused.

Dial 5 - ZOOM

Zooms in and out of the diagram.

Dial 6 - HORIZNTL

Pans left and right in the diagram after zooming. When panning, you cannot move out of the diagram work area.

Dial 7 - VERTICAL

Pans up and down in the diagram after zooming. Again, you cannot move out of the diagram work area.

Dial 8

This dial is currently unused.

3.7 Text

There are two types of text used in a network diagram: permanent text and notations. Any textual information which is in italics on the diagram is considered as notations and may be altered interactively by using the ADD ITEM/Labels selection. Any text shown in the standard font is permanent. When you are prompted for text entry (the downward pointing hand), either type in the string you want followed by a carriage return, or select another menu item to change your mind. Any text entered but not followed by a return will have no effect on the display or current status.

3.8 Macros

Macros are a means of incorporating code into a network file which is described in another file. They may be referenced repeatedly in the same file and may be nested to any level. When a macro is instanced, it appears in the diagram exactly as a function would, except that the name is preceded by M: instead of F:. Any existing network file that has been created by the Editor can be referenced as a macro. The macro description is derived from the top level frame of the network file, using the list of directories set up in your parameter file.

Since the Editor allows you to generate output files in ASCII, FORTRAN GSR, and Pascal GSR form, you must ensure that the code for all macros referenced in a network has the same form (i.e. ASCII, FORTRAN, Pascal). If you attempt to reference incompatible macros, (for example, an ASCII macro when you are generating Pascal code), the Editor gives a warning.

3.8.1 Instancing Macros

To instance a macro, use the BY NAME function key (F7) just as you do for selecting a primitive function, but enter the file name of the source network file instead of a function name. If the name does not conflict with an existing primitive function, the editor will try to find the file.

3.8.2 Compiling and Prefixing

Macros must be compiled using the Compile Macro option of the menu selection CONVERT NETWORK. Macros may be prefixed with the Use Macro Prefix option to distinguish multiple uses of the same macro. Compiling a macro produces a .MAC file, which may be incorporated into the code for another file with proper instancing and connections.

3.8.3 Prefixing Constants, Variables and External References

Since final names of constants, variables, and external references may not be known until the final level code conversion, you can flag them to indicate where prefixes should be placed. By adding \M\ at the beginning of the name within the string, the macro prefix will be added as needed in place of it, but no frame prefix will be included. By adding \F\, both macro and frame prefixes will be added.

3.8.4 Date Checking

Each macro instance is flagged with the date that the source file was last modified. This allows the Update Macro option to check against the original source file for changes. Macro code which is compiled is flagged with the last date the source file was edited and the date the code was compiled. A warning is given during code compilation if the .MAC file was generated from a different version of the source file than it was instanced from. The Updating Macros selection brings the instance into agreement with the source file, and recompiling brings the macro code into agreement with the source file.

3.9 User-Written Functions

User-written functions are referenced the same as macros. The name of a user-written function is indicated on the diagram as **U:name**, even though it is instanced in the code as **F:name**. If a network file contains no arcs, primitives, or detail frames, then it is automatically assumed to represent a user-written function. This allows you to create a description of the user-written function with named inputs, outputs, and internal comments which can later be used as a help item on that function. No macro code need be compiled for user-written functions, since they generate instances exactly as primitive functions do.

4. Editing

4.1 ADD ITEM

The ADD ITEM selection allows objects to be placed into the diagram space to construct a diagram. Generally the object appears at the cursor shape and is placed by pushing down the stylus when it is in the desired location. The asterisk shows where the stylus tip actually is. To "discard" the object that you are moving, merely select another menu item. Note that while objects are seen completely before they are placed, they will be clipped against the boundary of the display space once placed.

The ADD ITEM selection offers the following options: Detail Frame, Functions, Connector, Arc, and Labels.

4.1.1 Detail Frame

A frame is a portion of the hierarchical representation of the diagram, equivalent to a "page" of the complete network diagram. There are two types of frames: the one you are in (context frame) or a subsidiary frame within the context frame which refers to a lower level of the diagram hierarchy (detail frame). A context frame is a diagram page and the program may handle up to 100 frames within a file, though this may be an impractical size for memory and load/save speeds.

Context frames are bounded by a box outline corresponding to a higher level detail frame box. You can place Input Frame or Output Frame connectors on this outline in the context frame to create connection points between the context frame you are working in and the higher level detail frame which references it. Each context frame has a PREFIX (upper left-hand corner) which can be changed one level higher on the detail frame representation. The prefix is (optionally) used before the function names to maintain unique naming between frames. The prefix can be edited in the current frame by picking the prefix in the header bar while in Labels mode. Each context frame also has a NAME which is used to provide a more descriptive identifier while editing.

Frames are created in two ways. When a new file is created, the top level frame is created at the same time. From there on, the ADD ITEM/Detail Frame selection will add a symbol for the detail frame and also create the accompanying frame.

Detail frames are displayed as pseudo-3D boxes to indicate that they include more detail at a lower level. Initially all detail frames have 0 inputs and outputs and are created as a minimum size detail symbol. As connectors are added in the corresponding frame below, the detail symbol will be updated to reflect its new description. The detail frame includes a single line label which may be edited exactly as a function box label.

Input/Output Frame connectors may be attached to the left and right edges of the context frame (outside box), respectively. They may later be moved or deleted as necessary, at which time the detail and its attached arcs will be modified as needed.

Before a frame can be deleted, you are asked to verify the delete. Then the frame is deleted along with all contained detail frames and objects.

To move between frames, there are three function keys: UP, DOWN, and OUTLINE. UP will reset the current context frame to the parent of the one you were working in. DOWN will move into a detail frame in the current context. If there is more than one, a large down pointing arrow will appear to allow you to select the desired detail frame. By hitting the OUTLINE function key, an outline page will appear for selecting any frame in the current file; indentation indicates the tree structure of the file. The frame currently being edited will be highlighted. The outline is implemented as a page in the diagram and may be scrolled using Dial 2.

4.1.2 Functions

A function is an intrinsic function or initial function instance supported by the PS 390 Command Language. When the menu item Functions is selected, a submenu of function classes appears, organized by class. Since the list of classes is long, some are off the bottom of the display and may be seen by turning the dial marked SUBMENU to scroll up and down. When a class is selected, its list of functions will appear in place of the class submenu for selection. When a function is chosen, its box representation will appear.

A function may also be selected by name. This may be a faster method for many sessions. Press the BY NAME function key (F7) and you will be prompted for the name of a function. After the name is entered, the function box will appear and may be placed. The same box will appear at the cursor after one has been placed, and may be placed as often as needed.

You must enter the complete name of the function. For "n" type functions such as F:SYNC(n), you are prompted for the number of outputs.

The BY NAME key can also be used for instancing macros and user-written functions. Unlike intrinsic functions or initial function instances, only one instance of the macro or user-written function can be placed at a time. After one instance is placed, you are prompted for another name.

A function consists of a box; a set of up to 50 inputs, which appear on the left edge; a set of up to 50 outputs; the name of the function type (**F:function_name**) on the top half of the box, and a user label, initially assigned by the system as *P(n)*, written in italics (e.g. *P1*). For initial function instances, there is no user label and the function name appears as **TABLETIN**, or whatever. This user label may be altered interactively at any time (see Labels below). Long names are broken at an underscore if one is present in the name.

The function box should be placed within the context frame. At any point that arcs are being drawn, the function's inputs and outputs will be activated for picking as appropriate.

The **PRIMITIVEPROMPT ON/OFF** option in the parameter file **NETPARMS.TXT** can be set to enable or disable prompting for function names immediately as they are instanced and placed.

4.1.3 Connectors

There are various types of connectors, but all are basically similar in function to the primitive inputs and outputs. They serve as the source or destination of an arc, which establishes a data path between two points. Currently there are the following types of connectors: Input Frame, Output Frame, Constant, Variable, In-External, and Out-External.

The connector shapes are indicated by the direction of the arrow and a contained letter (C for Constant, V for Variable, E for External) and may be freely placed anywhere in the diagram. For connectors containing text, you will be prompted for an initial value and then a copy of the shape and the value will be fixed at that location.

4.1.4 Input Frame

Input Frame connectors are attached to the left-hand side of the surrounding box outline in the frame and represent input to the frame. When you select a frame connector, the system will assign a name to it. This can be changed using the ADD ITEM/Labels menu selection or function key F10 (TEXT). Names will be reflected in the detail frame above as soon as you have finished adding them and moved to another action. Up to 50 of these may be placed.

4.1.5 Output Frame

Output Frame connectors are attached to the right-hand side of the surrounding box outline and represent an output channel from the frame's contents. They are treated the same as Input Frame connectors.

4.1.6 Constants

Constant connectors allow a line of text to be SENT to another point in the network. You will immediately be prompted for the value that you wish to SEND. Enter this string exactly as it would appear in the normal PS 390 command syntax. Note that syntax checking is not currently performed by the Network Editor. You must then route an arc from the constant connector to the intended input.

4.1.7 Variable

Variable connectors create variables to hold values apart from primitives. The variable will be instanced using the optional prefix in the name if \M\ or \F\ are included in the name. Any connections going to these variables will be added when code is generated.

4.1.8 In-External

In-External connectors are a means of making connections to external networks or display structures freely. They are input points from outside sources of data. You should be careful in using them to make sure that when the code is downloaded, these connections already exist if they are data outputs. Also when prompted for the connection name, you should enter the complete reference including the port number (e.g. INNAME<1>).

4.1.9 Out-External

Out-External connectors are output points to external destinations. Make sure that when the code is downloaded these connections already exist if they are data outputs, though this is not as important as it is with in-external connectors. Also when prompted for the connection name, you should enter the complete reference including the port number (e.g. <append>Out-VecList).

4.1.10 Arc

An arc is a line indicating a pathway along which data tokens are expected to move during execution. They are much like wires between the inputs and outputs of integrated circuits. They correspond to the CONNECT or SEND statement in the PS 390 Command Language. Arcs must start at a data source (frame or external input, a constant, a primitive or detail output) and terminate at a data target (frame or external output, a variable, or primitive or detail input). An arc may follow a circuitous route, making as many turns as necessary. You start the arc as needed and then manually route the arc to the desired endpoint. The pathway is automatically grid locked and bent to horizontal or vertical lines. If the arc is not completed by making another menu selection before completion, it is canceled.

When Arc is selected, the cursor changes to an Arc Start Arrow: a single arrow which points to the left. Once the arc is started, the cursor changes to an Arc End Arrow—an arrow pointing to the right. A corner shape will appear at the last bend to indicate in what directions a turn can be made. Arcs can only be routed in horizontal and vertical segments. The point of bending is indicated by a four-way corner shape. Each time a new corner is added, this corner shape moves. Once terminated, the arc will flash once and then become a permanent part of the diagram, and all corners will be rounded off to more easily distinguish the arcs from the other squared off shapes and lines around them.

Arcs are homed into the starting or ending connector.

When an arc is placed, the editor checks the types of the output and input connectors and beeps and issues a warning if they are incompatible. An arc placed between incompatible connector types will be highlighted. These arcs will remain in the diagram and must be deleted explicitly. Note that connection type checking is only performed on connections between primitives. Connector symbols such as Constants and Variables which have editable strings are not checked currently.

Duplicate arcs are deleted when a second connection is made between the same pair of connectors. There is no need to explicitly delete the old arc.

Note that Arc can also be selected by pressing function key F10 once.

4.1.11 Labels

Labels are any text strings in the diagram which can be edited. Labels appear in italics to distinguish them from text which cannot be edited. The first shape that appears is an arrow which points to the upper right. This is used to pick either a point in space at which to place a free-floating label (comment) or to pick any object which has a label associated with it such as a function box, or a previously defined label. If a new point is picked, then a new comment label will be placed there. Otherwise the already existing label will be replaced by the new value.

Labels are limited to 80 characters. Any label larger than that is truncated to 80.

Once a selection is made, the text-entry hand shape appears and is frozen in position where you have pointed. A second, dimmer copy of the hand will move about, allowing you to cancel the action by making another menu selection. The hand indicates that keyboard entry of text is expected. As you type on the keyboard, the text will appear in place at either the position of the previous label or at the point at which you are pointing. A second copy of what you are editing appears at the prompt line.

To correct mistakes, use the DELETE key on the PS 390 keyboard, and deleted characters will be erased. Once the string is complete, press the RETURN key and the new value will be stored. You will remain in text-entry mode so that more strings can be entered until you enter a return. In this way, you can create text as a block. To change from the Labels selection, pick another item from the menu.

The text editor uses the following control characters for editing effects:

CONTROL-A	Moves the cursor to the beginning of the line.
CONTROL-B	Moves the cursor back (left) one character.
CONTROL-D	Deletes the character at the cursor position.
CONTROL-E	Moves the cursor to the end of the line.
CONTROL-F	Moves the cursor forward (right) one character.
CONTROL-K	Kills (deletes) to the end of the line.
CONTROL-R	Retype line
CONTROL-U	Deletes the entire line.
DELETE	Deletes the character to the left of the cursor.
RETURN	Signals completion and disconnects the keyboard.

Note that Labels can also be selected be pressing function key F10 twice.

Specially flagged labels can be used to insert random PS 390 commands in a network. Floating comments which start with \+\ or \-\ indicate commands to be inserted before or after the other code for the frame, respectively. These commands are always written to the output file during code conversion, regardless of the SUPPRESS COMMENTS setting.

The statements can be ordered by including a priority number in the flag. Statements prefixed with \-1\ are guaranteed to be output before statements prefixed with \-2\. This is useful for sending an ordered sequence of constants to the same input of a function.

Typically, commands that should be inserted before the other code for a frame are INITIALIZE commands or display structure definitions. Commands that should be specified to go at the end of the code for the frame are SETUP CNESS commands, and SEND statements. NETEDIT does not perform any syntax or validity checking on the commands.

Names of functions, variables, and display structures that are referenced in these commands may be prefixed with \F\ and/or \M\ to indicate that the appropriate frame and/or macro prefix should be substituted during code conversion.

4.2 MOVE

All of the diagram objects may be moved once they have been included in the diagram, except arcs which are only moved by moving what they are attached to. A four-directional arrow will appear to indicate that you may move objects. You may pick any of the above objects for moving at any point in their symbol. An identical "ghost" copy will then appear to help you accurately place the object again. If the ghost-symbol is not placed within the diagram, no movement will occur. The four-way arrow will also shrink to indicate that you have successfully picked an object up and are in the second half of moving an object. Any placement rules that apply to that object, such as placing a connector on the frame, still apply during movement.

To move an object or set of objects to another frame, pick the detail frame that you wish to move into, or pick the outer box outline to move up into the parent frame of the current context frame. The frame display will change to the selected context and you can repeat this process until you do NOT pick a frame or detail. At this point you can place the object or set of objects as if they were still in the original frame. Arcs which have had both of their endpoints moved are carried along while arcs for which only one endpoint has been affected will be stretched if the move is within the same frame, and destroyed if the move has jumped into another frame. A detail frame cannot be moved down into itself, even though it may originally have been picked up or included in the selected area. The frame outline will automatically be restructured to reflect the change made due to the move operation.

MOVE may also be selected by pressing function key F8 once.

4.3 MOVE AREA

By selecting MOVE AREA and indicating any two opposite corners of an area box, you can move the items contained within the area. A large lower-left angle (first point) and a large upper-right angle (second point) set the area. Select the lower-left and then the upper-right corners and then move the box to a third point. Objects within will be shifted to the new location of the area box.

Detail frames must be completely contained within the area box if they are to be moved. Connectors and primitives need only have their placement point (the center of the cursor shape you notice when moving the item) within the area. Arcs are moved if their connection points are moved; they will be bent if only one endpoint is moved, but moved completely if both endpoints are moved. MOVE AREA will not allow you to position items outside of the frame area.

MOVE AREA can also be selected by pressing function key F8 twice.

4.4 DELETE

Any object in the diagram may be deleted. When DELETE is selected, the cursor changes to a large X shape which can be used to pick any of the diagram objects. The object picked will be removed from the display together with any attached arcs.

Delete can also be selected be pressing function key F9 once.

4.5 DELETE AREA

As with MOVE AREA, you can delete all items in an area by indicating any two opposite corners of the area box. Objects within will be deleted from the diagram with the same inclusion rules as in MOVE AREA. Arcs are deleted if either of their connection points are deleted.

DELETE AREA can also be selected by pressing function key F9 twice.

4.6 OPTIONS

The OPTIONS area of the EDITING menu offers selections that are less often used. These are: Change Scale, Redraw Frame, Replace Functions, and Update Macros.

4.6.1 Change Scale

Change Scale is used to change the overall size of the working page from the current size (size 2) up to size 20, which gives 10 times the working space. The selected size is noted in the frame data record and is automatically reset when you enter the frame. This allows you to have different sized frames within the same file. Frame connectors are moved automatically to the outer edge of the frame box.

4.6.2 Redraw Frame

This option will clear and redraw the frame if for any reason the display contains errors or was partially lost in transmission to the PS 390.

4.6.3 Replace Function

This selection lets you pick an existing function in the diagram and replace it with another. You are prompted for the name of the replacement function. The replacement can be any valid type of function: Initial Function Instance, primitive function, macro, or User-Written Function. When functions are swapped, existing arcs are checked. They are highlighted if the connector types are incompatible with the new function. Arcs leading to inputs no longer available in the new function are deleted.

4.6.4 Update Macros

This option locates all macros used in the file and compares them to the original network file from which they were derived. If the file has since been edited, the macro is updated. First, the display is set to the page containing the macro to allow you to see the related changes. Then, as with changes to detail frames, the existing connections are moved or deleted if the corresponding frame connectors in the top level of the source file have been changed. When updating is complete, the display returns to the original page.

Note that updating is based on the internal ID of the original frame connectors. If you delete the connector, connections to it are lost even if you rename a new connector to the same name. This allows you to change names without losing the original connection, but if you delete the original connection, the editor will also delete all connections to it in the corresponding usage as a macro.

5. File Control

A network file is a structured ASCII file with an extension of .NET which is created and edited by NETEDIT. The FILE CONTROL selection offers the following options: SELECT NETWORK, BACKUP NETWORK, SCRATCH NETWORK, RECOVER, and RENAME NETWORK.

5.1 Select Network

This selection lets you enter the name of an existing file which you want to edit or lets you create a new file. You may use directory names or logical names preceding the filename. Do not give the file an extension (.NET is assumed by the Editor).

NOTE

File names are truncated to nine characters. Before a new file is created, the directory list in your parameter file is searched from beginning to end to see if the file name already exists.

5.2 Backup Network

During the course of editing, you can back up the file by selecting BACKUP NETWORK which will save the current network file you are working in. Backup also happens automatically when EXITing or SELECTing a new network file. Backup will not occur if no editing has taken place. Simply pressing the MOVE function key is sufficient to "touch" a file and consider it edited.

5.3 Scratch Network

If you wish to abandon the network you are currently working on without saving any of it, select SCRATCH NETWORK. As a precaution against stray menu picks, you must select this twice before the network is scratched.

5.4 Recover

A log file with an extension of .LOG is kept for every edit of a file. Log files are purged after a normal exit from the editor, but only in the current working directory. If a crash occurs during editing, the log file can be used to recover editing that was done between saves of the file.

Use the RECOVER selection to rerun the editing operations that were performed before the crash. DO NOT LOAD THE ORIGINAL NETWORK. The RECOVER selection loads the network automatically. Then, if a log file is found with the correct name, it is read in and executed as if the commands were coming from the PS 390. The diagram is reconstructed step by step. When the recovery is complete, a message is displayed. At this point, select BACKUP NETWORK to close the current log file and open another.

If the crash was caused by the Editor, or if you wish to undo the last few commands that you gave, edit the log file and remove the last few lines before you select RECOVER.

5.5 Rename Network

This selection lets you rename the file you are currently working on. Note that the editor does not check to see if the file name you enter already exists.

6. Convert Network

This option will produce an output file from the diagram structure currently in memory. An ASCII file has an extension of .300. A FORTRAN or Pascal file has an extension of .FOR or .PAS. In all cases, the name of the file is the same as the name of the network file. Other extensions for ASCII files can be set up in your parameter file NETPARMS.TXT. However, the parameter file cannot be used to change the extensions of GSR files.

Primitives result in **name := F:function name;** statements. The selection optionally adds the prefix in each frame to the primitives within it. Arcs between sources and targets produce **CONNECT name1<1>:<1>name2;** connection statements or equivalent GSR calls. Constants are sent to targets with **SEND value TO <1>name;** commands or equivalent GSR calls. VARIABLE connections cause the creation of the needed variables. External input and output connections are connected, expecting the external code to already be resident in the PS 390. Free-floating labels in the diagram are added as comments within the code. Labels flagged with \+\ or \-\ become literal PS 390 commands inserted before or after the other code for the frame in which they are included.

NOTE

> This selection will use the file currently in memory, which may be more recent than the accompanying diagram file unless you have just loaded or backed up the file.

The following options are available: ASCII Output, FORTRAN GSR, Pascal GSR, Use Frame Prefix, Use Macro Prefix, Compile Macro, and Suppress Comments. Options are selected by being picked once and canceled by being picked again. When an option has been selected, it is highlighted. Some options are present by default. Before you select the output file type (ASCII, FORTRAN, Pascal) be sure to toggle the other options to the selections you want.

6.1 ASCII Output

This selection generates an ASCII file from the network diagram. Choose this selection after selecting the other options as you wish. If an item is highlighted, it is selected; if not, it is disabled. The file generated will have the same root name as the source file and an extension of .300 or the user-selected extension in the parameter file.

6.2 FORTRAN GSR

This selection generates a FORTRAN subroutine from the network diagram. The FORTRAN code is compatible with VAX/VMS FORTRAN-77. Choose this selection after selecting the other options you want. If an item is highlighted, it is selected; if not, it is disabled. The subroutine file produced will have the extension .FOR.

To compile and link the generated code, the host program must perform calls to attach and detach the PS 390 (PATTCH/PDTACH). You must also supply an error-handling routine, as described in the DEC VAX/VMS FORTRAN GSR documentation, called ERR.

The output file generated by the Editor may be compiled independently or included in a file containing other FORTRAN subprograms. You must then link it with your main program, the error-handler, and the FORTRAN GSR library.

6.3 Pascal GSR

This selection generates a Pascal procedure from the network diagram. The code is compatible with VAX/VMS Pascal V2. Choose this selection after selecting the other options you want. If an item is highlighted, it is selected; if not, it is disabled. The procedure file produced will have an extension of .PAS.

To compile and link the generated code, the host program must perform calls to attach and detach the PS 390 (PATTACH)/PDETACH). You must also supply an error-handling procedure as described in the DEC VAX/VMS Pascal GSR documentation, called PI_Error_Handler.

To compile the procedure, it is recommended that you include the file in your main program using the "% include" directive. Your program must also include the declarations in PROCONST.PAS, PROTYPES.PAS, and PROETRN.PAS. After compiling the program, you must link it with the Pascal GSR library.

6.4 Use Frame Prefix

Each context frame has a prefix (upper right-hand corner) which can be changed by going one level higher on the detail frame representation or by picking the prefix from the header bar. The prefix is (optionally) used before the function names to maintain unique naming between frames. This selection lets you specify whether or not frame-prefixes are used with function names. If you do not select this option, prefixes will NOT be used.

6.5 Use Macro Prefix

This selection controls the inclusion of a special macro prefix in the ASCII code file in several ways. The prefix (M1$, M2$, M3$, etc.) is used to distinguish multiple uses of the same macro. If this option is selected when the original macro is compiled, then the prefix will always be used later. You should use this selection if you intend to make multiple uses of a macro. If the option is off when the macro is compiled, then use of the prefix is optional.

If the option is selected during final code generation, then all macros will include the Mn$ prefix. Otherwise, only those macros that were compiled to force inclusion of the prefix will use the prefix.

Prefixes are a way of making multiple copies of a macro with unique names. At the same time, by making these optional, the user has the flexibility of controlling the function names completely.

6.6 Compile Macro

A macro must be compiled before it can be included in the code of a network that instances it.

Macros are compiled into an intermediate form of ASCII code which is different from the code which the PS 390 normally expects. This form of output is selected by the Compile Macro option. When this is selected, the ASCII output is written to a macro file with an extension of .MAC which is specially generated to allow later inclusion as a macro. Codes are embedded (\n\) which can later be interpreted to provide unique prefixes and allow arbitrary nesting of macros.

These codes make the ASCII code unreadable to the PS 390. In addition, the connections from the top frame connectors are listed at the end of the file to allow a network that instances the macro to be hooked up to the right inputs and outputs within the file. When one macro is compiled, other macro files are merged in with the special codes updated to allow unique prefixing later.

6.7 Suppress Comments

This selection lets you decide whether or not comments and frame headers are included in the output file of ASCII commands or GSRs. If you choose this selection, no comments will be generated. If you do not choose Suppress Comments, comments and headers will be generated in the file. Comments are included in an arbitrary order, but they are placed with the code for that frame.

NOTE

Literal PS 390 commands which are flagged with /+/ or /–/ are written to the output file whether or not you choose the Suppress Comments selection.

Appendix A
Installation Instructions

NETEDIT is distributed on magtape along with NETPROBE, the function network debug program, and MAKEFONT, the character font editor program. The tape contains executables as well as source files. This simplifies the installation procedure for sites where no modifications to the source or data files are planned, or where no Pascal compiler is available. Two sets of installation instructions are supplied below. The first is simpler and assumes you have already used the VMS Backup Utility to copy the distribution tape to the host.

1. Installation without Rebuilding the Executables

The procedure for installing NETEDIT without rebuilding it entirely is as follows:

1. Set default to NETEDIT subdirectory in the A2.V02 subdirectory.
2. Edit the NETUSER.COM and change the definition of the NETROOT (marked !*INSTALL-DEPENDENT) to the name of the directory created. Make sure this file is readable and executable by all users. See comments in Netuser.com "Site Customization of Netuser.Com."

3 Copy the empty user log file, NETEDIT0.USR to NETEDIT.USR. Set the protection on this file so that it is writable by all users.

2. Installation with Rebuilding Required

The files are installed in three stages. First, the files are transferred onto the VAX system. Then two menu-driven command files, NETBUILD.COM and NETUSER.COM are edited to customize the home directory in which the files are to reside. Finally, NETBUILD.COM is run to compile and link all of the files.

2.1 Distribution Tape Format and Installation Procedure

PS 390 VAX/VMS sites receive the distribution tape (PS 390 host software) in VMS Backup format. To install the VAX PS 390 host software, first create a subdirectory for the PS 390 software and set your default to that directory. Using the VMS Backup Utility, enter the following commands:

```
$ Allocate   MTNN:
$ Mount/Foreign MTNN
$ Backup MTNN:PSDIST.BCK [...]*.*
$ Dismount MTNN:
$ Deallocate MTNN:
```

where MTNN: is the physical device name of the tape drive being used.

This will create the subdirectory A2V02.DIR which is the parent directory of the PS 390 host software.

UNIX sites will receive a 1600-bpi distribution tape in tar format. IBM sites will receive a 1600-bpi distribution tape with a block size of 6400 and a logical record length of 80.

2.2 Customizing the Command Files

A menu-driven command file called NETBUILD.COM is provided to help you install the files. Another command file, NETUSER.COM, displays a programming utility menu from which NETEDIT, NETPROBE, and MAKEFONT are accessed. Both command files must be edited to set up the home directory in which the utility program files will reside. With a text editor, enter NETBUILD.COM and NETUSER.COM and change the entries which are marked with !*INSTALL-DEPENDENT*. These are the name of the directory in which the files will reside, the UIC reference, and the directory where the Pascal GSR library resides.

2.3 Installing the Files

When the changes have been made to NETBUILD.COM, start the command file by typing the following command.

```
$ @[HomeDir]NETBUILD.COM
```

[HomeDir] is the name of the directory in which the files reside. The following menu is displayed.

```
Evans & Sutherland PS 300 Utilities Maintenance
      Command File V1.08 Main Menu

  0) Exit
  1) Initial installation - interactive
  2) Initial installation - submit as batch job
```

To install the network editor files, select 1 or 2 for interactive or batch compilation and linking of the entire system. Note that compilation will only occur if the object code is missing or if the source code or related files have been updated.

The other selections on the menu display further menus of options for updating programs individually (selection 3), updating the data base (selection 4), and miscellaneous support activities (selection 5).

3. Files that are Loaded

The following is a list of all the files that are loaded from the distribution tape. The files are ordered by logical groupings and in the same way they would appear if you were working in a multiple directory.

WORK:	
NetParms.TXT	A sample parameter file
Init.300	ASCII command file to initialize the PS 300
NetBuild.COM*	The NetEdit maintenance command file
NetUser.COM*	The shared user utility command file
NELinker.COM	A command file to link programs with NEUtil library
NEPascal.COM	A conditional Pascal compilation command file
NEFileLst.DAT	The list of files needed for NetEdit distribution
NEFileDbg.DAT	The list of files needed for NetProbe distribution
NetProbe.PAS	The NetProbe debugger source program file
NetProbe.COM	The NetProbe maintenance command file
NetProbe.300	The NetProbe debugger control network
NetProbeA.300	Command file to label function keys
NetEdit0.Usr	A dummy NetEdit usage log file-copied automatically
PROG:	
NEComm.MOD	PS 300 communications
NEControl.MOD	Intermediate level database management
NEConvert.MOD	Network->ASCII command file conversion
NEDraw.MOD	Object graphics
NEEdit.MOD	High level editing control

NEError.MOD	Error handling management (see also NEUtil)
NEGraph.MOD	Generic graphics support
NEInfo.MOD	Function and Help database interface
NEMain.MOD	Top control loop and file control
NEParse.MOD	Parsing routines
NERecord.MOD	Low-level database management and I/O
NEUtil.MOD	Shared library of string routines and file handling
NEUtilCon.DCL	NEUtil Constants
NEUtilTyp.DCL	NEUtil Types
NEUtilVar.DCL	NEUtil Variables
NEUtilExt.DCL	NEUtil External declarations
NEError.DCL	Error codes
NetEdit.DCL	Global declarations
NetEdit.PAS	Top-level program for NetEdit
NetEdit.EXT	Global external declarations
DBASE:	
Config.TXT	Configuration file function list
D*.*	Digit vectors for 1-9 input and output on functions
GrandCF.OLD	P5 function database
GrandCF.TXT	A1 function database
InitD.PAS	Merge digits into sets of 1-9
NetData.PAS	Generate main function database files (user, system)
NetFcn.PAS	Parse the function appendix file into a database
NetLoad.PAS	Bind output of NetData and NetFcn together
NetResolv.PAS	Merge function and help databases and cross reference
OldToNew.PAS	Compare P5 and A1 databases and produce change list
ParsUser.PAS	Parse the users manual and produce indexed file
PS300Man.DOC	PS 300 User′s Manual appendix on functions
FNEUser.Man	Function Network editor′s manual
DOC:	
Announce.DOC	Announcement of new release
Database.DOC	Function and Help Database notes
V108.DOC	V108 release notes
MENU:	
Menu.DOC	Menu construction information
NetMenu.DAT	Menu outline file
NetMenu.PAS	Menu construction program
NETW:	
Editor.300	ASCII version of network editor support
Editor.Net	Top network file-host communications and integration
EdMenuMgr.Net	Menu manager network-menu display and highlighting
EdPlace.Net	PointLine placement network-Drawing
EdSysMgr.Net	System management network-Hardcopy, Memory Alloc

EdText.Net	Text entry network
FetchPr.Net	Fetch and print network-used in EdSysMgr
HNet.Net	Help page control-dials
PickMgr.Net	Pick manager
Timer.Net	Clock display timer control
*.MAC	Macro code versions of net files
Editor.Doc	Description of network contents
Editor.DSP	Main display structure
NetInit.300	Front end for network editor network
NetEnd.300	Tail end for network editor network
DRAW:	
NetDraw.PAS	Simple drawing program used to draw cursors
NETran.PAS	Translate and scale drawings
NetDraw.300	NetDraw support network
NeCursors.300	Combined library of cursor shapes-must be broken into individual cursor files for editing.

4. Error Handling

Should the program crash, the current routine stack will be recorded automatically in NETEDIT.ERR. An error message will appear on the status line at the bottom of the screen, and the terminal will be reset to the normal terminal emulator mode. After a crash, you should save the error file, along with the log file that is kept during a session (Filename.LOG) and your data file (Filename.NET) as they are so that they are available for later examination during attempts to identify the problem.

5. User Log File

NETEDIT.USR is a log file that is kept to indicate who uses NETEDIT and when. This file may become long and should be cleared occasionally by the system manager. If you have no use for the log file, it can be disabled in the NETUSER.COM command file.

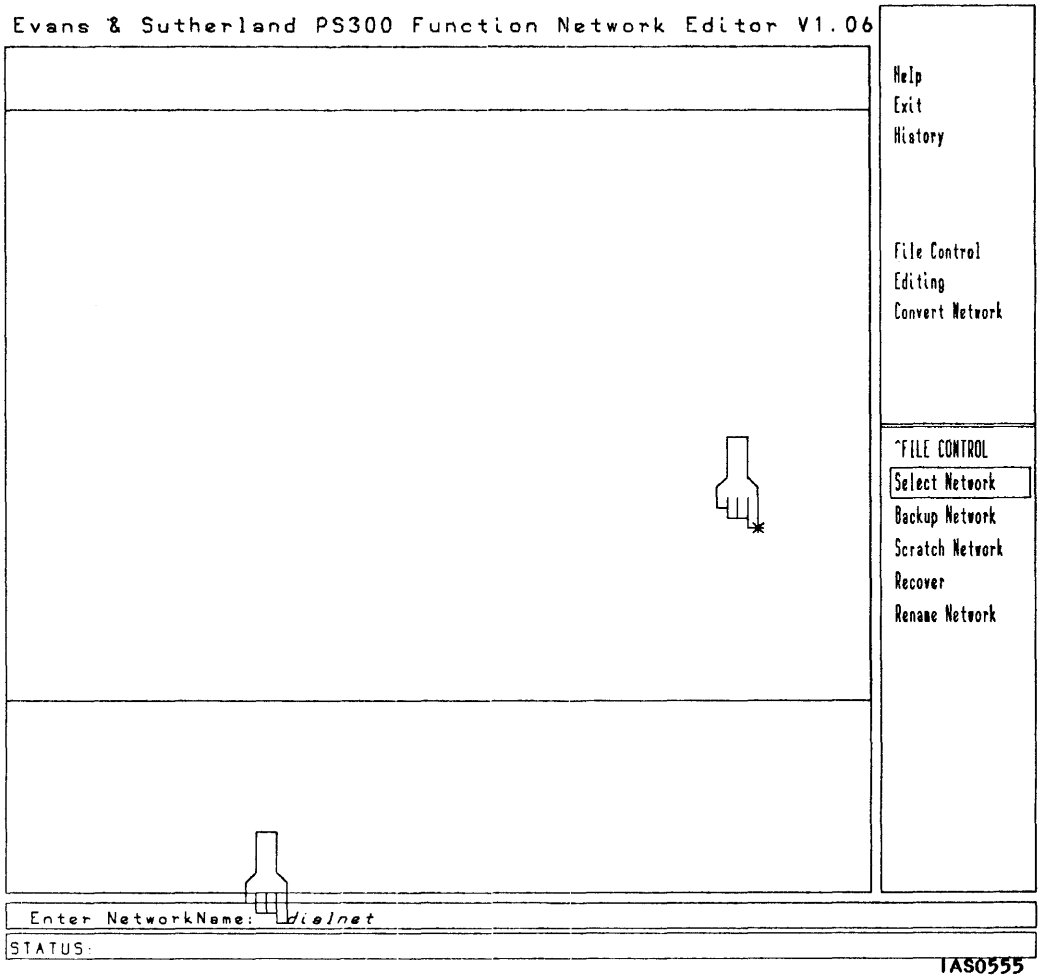
6. User-Written Functions

Source files for the user-written function used by NETEDIT are also provided, along with a command file to build the .300 files which may be downloaded to the PS 390. However, to rebuild the user-written functions, you must have the Motorola 68000 cross software, which is not supplied by Evans & Sutherland.

Appendix B
Sample Editing Session

In this sample session, NETEDIT is used to design a simple function network which allows the control dials to be used to rotate, translate, and scale displayed objects. The transcript illustrates the sequence of operations used in creating the network, and shows how to place functions, constants and arcs; create and manipulate detail frames; and make connections to external display structures.

When NETEDIT is started, it will ask you to select the network to be edited. In Figure 4-3, the SELECT NETWORK menu item has been selected and the name of the network (DIALNET) typed in. The network file will be called DIALNET.NET and the file containing the ASCII code will be named DIALNET.300, unless specified otherwise in the parameter file.

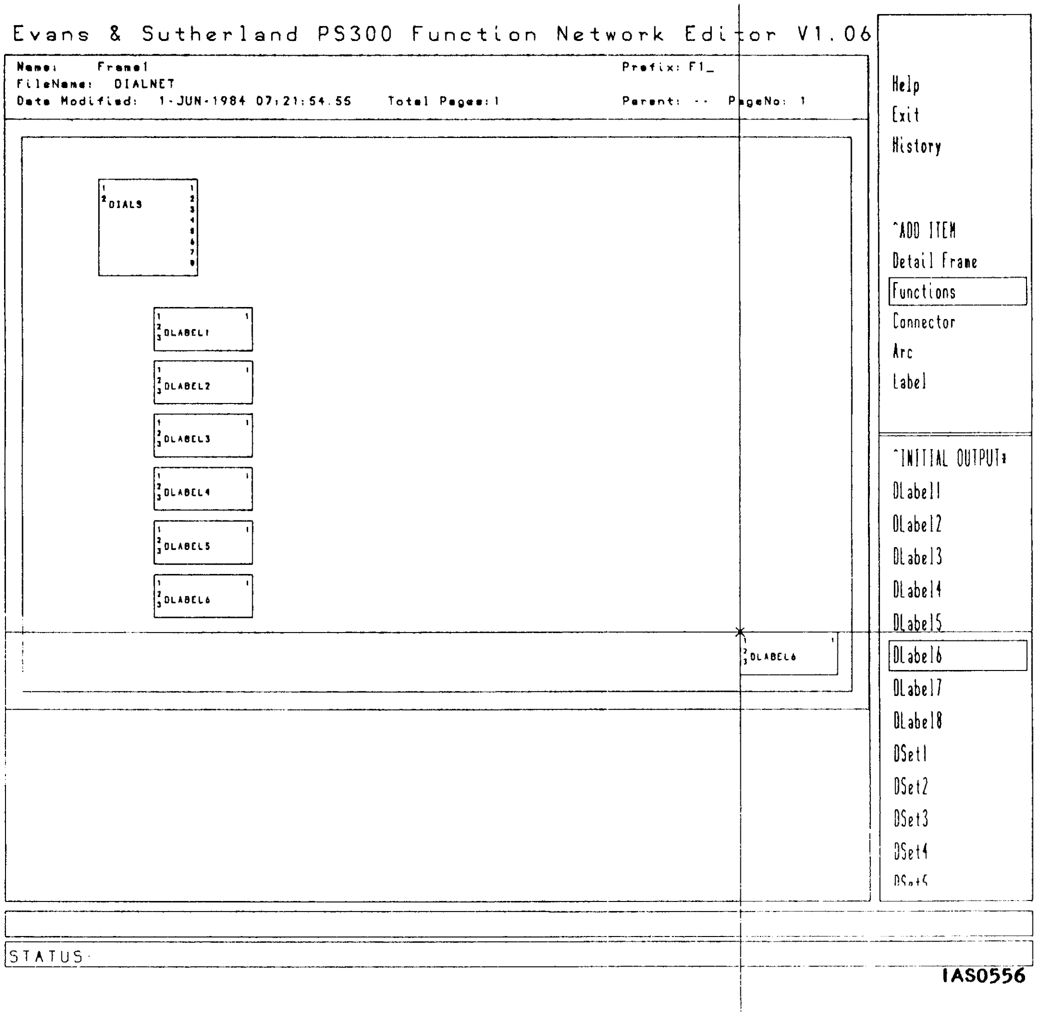


Enter NetworkName: dialnet

STATUS:

Figure 4-3. Selecting the Network File

In Figure 4-4, the menu items for EDITING, ADD ITEM, and FUNCTIONS have been selected to get the Functions menu. Here, functions are being placed in the top-level frame. All of the functions that have been added here are Initial Function Instances, so they have not been assigned user-defined names as primitive functions are. The cross-hairs cursor has been turned on to help align the function boxes.

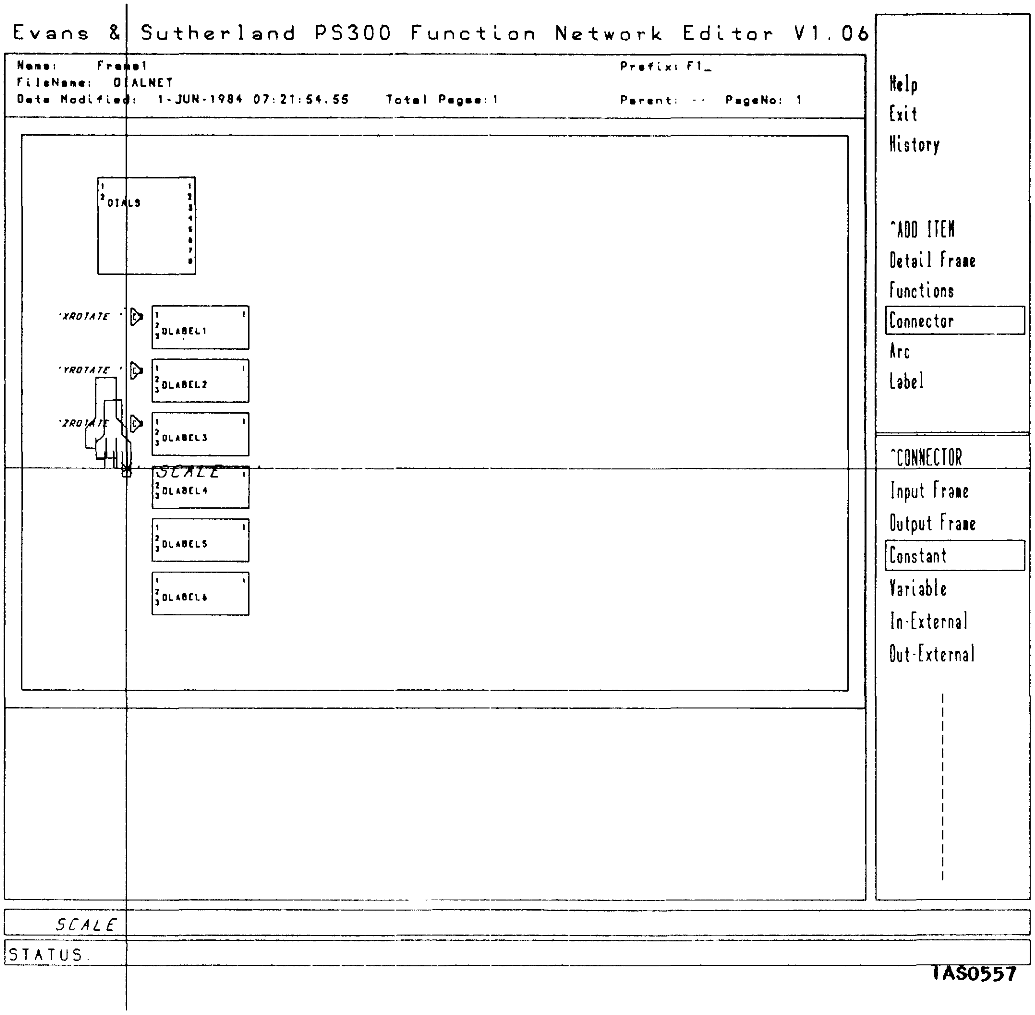


Evans & Sutherland PS300 Function Network Editor V1.06

Name: Frame1 Prefix: F1_
FileName: DIALNET
Date Modified: 1-JUN-1984 07:21:54.55 Total Pages: 1 Parent: -- PageNo: 1

Figure 4-4. Placing Functions With the Cross-Hairs Cursor

In Figure 4-5, the CONSTANT item has been selected from the EDITING menu to allow placement of constants, corresponding to PS 390 SEND commands. The user is prompted for the value to be sent as each constant is positioned. Note that the constants are not connected automatically to function inputs. Here, strings to label the dials according to their functions are being created.

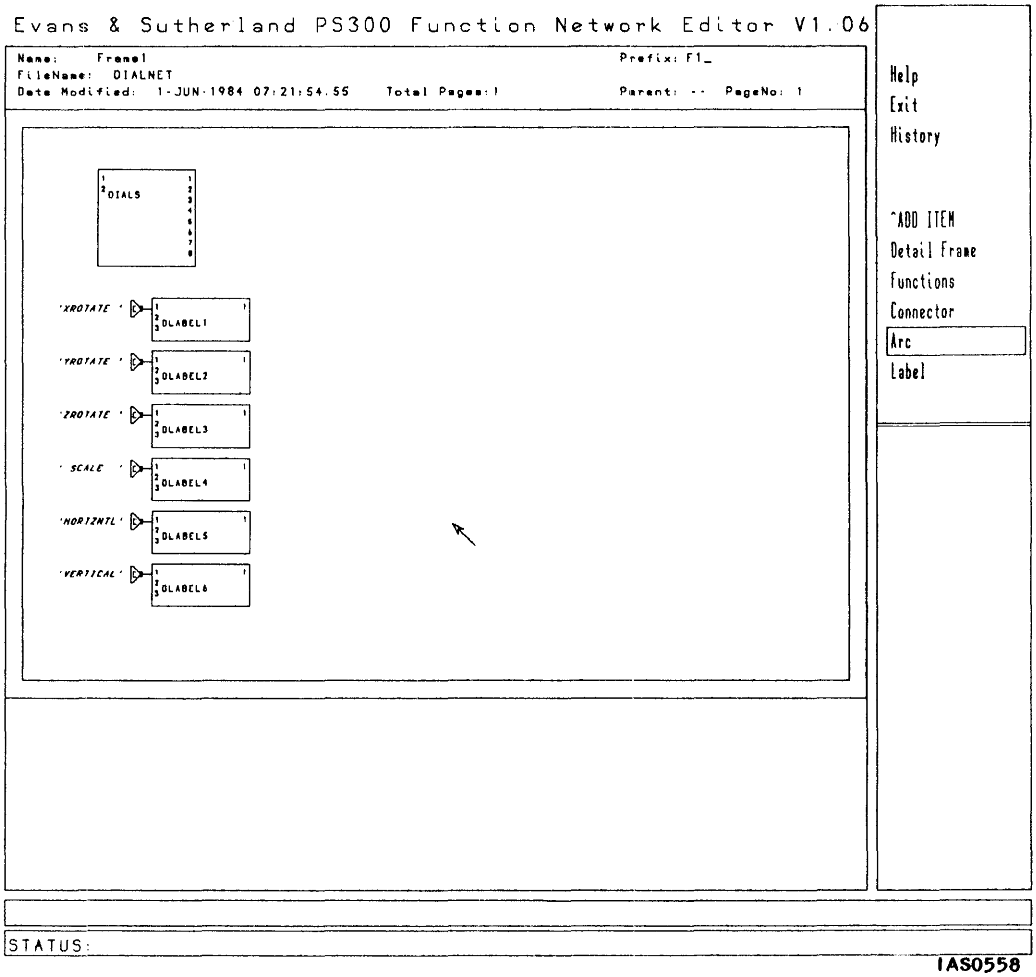


Evans & Sutherland PS300 Function Network Editor V1.06

Name: Frame1 Prefix: F1_
FileName: DIALNET
Date Modified: 1-JUN-1984 07:21:54.55 Total Pages: 1 Parent: -- PageNo: 1

Figure 4-5. Creating Strings to Label the Dials

Arcs have been added to connect the constants to the function inputs in Figure 4-6. Arcs may be inserted either by selecting the Arc menu item or from the ARC/TEXT function key.



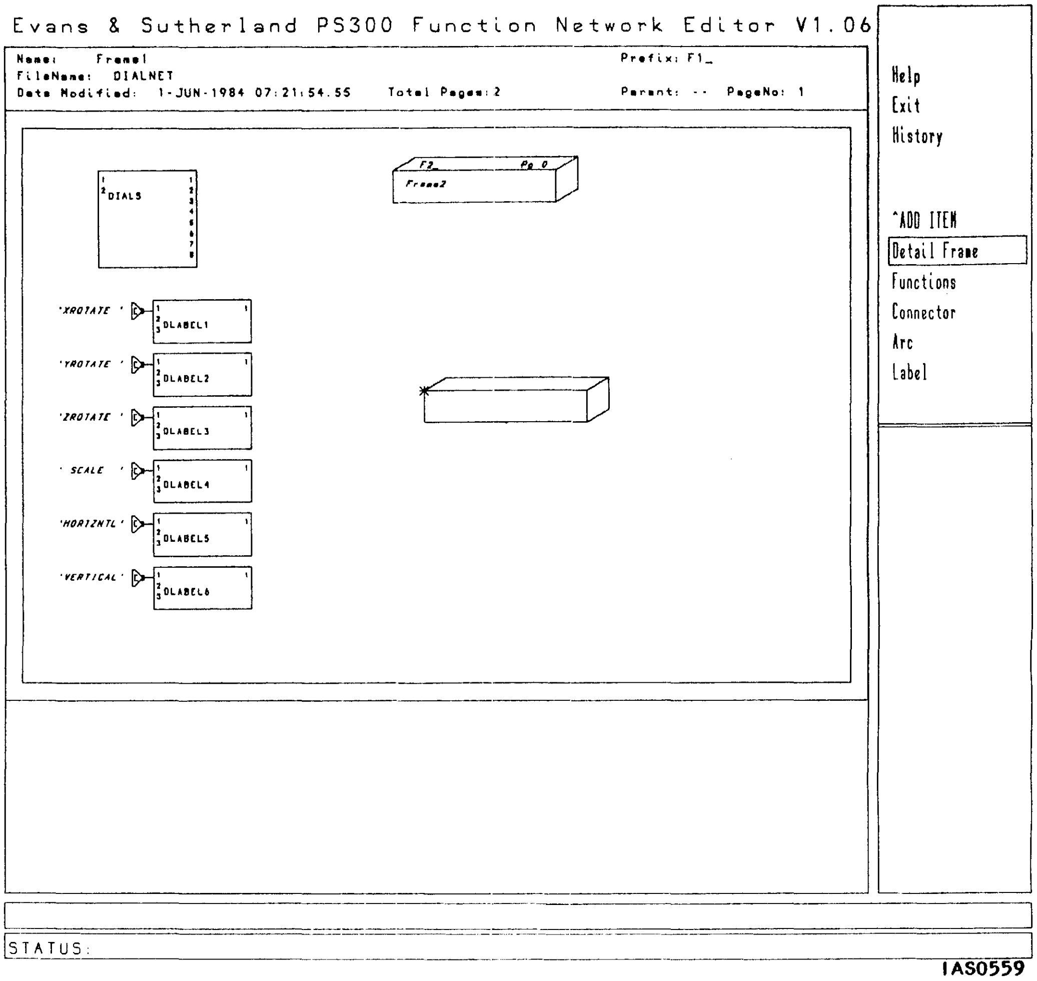
Evans & Sutherland PS300 Function Network Editor V1.06

Name: Frame1
FileName: DIALNET
Date Modified: 1-JUN-1984 07:21:54.55 Total Pages: 1
Prefix: F1_
Parent: -- PageNo: 1

STATUS:

Figure 4-6. Connecting Constants to Inputs With Arcs

Detail frames are being created in Figure 4-7. Instead of putting on one page all the functions to turn input from the dials into transformation matrices, using detail frames allows the details to be split up into logically independent blocks. Notice that all detail frames initially have no inputs or outputs; the names and prefixes are assigned default values automatically.



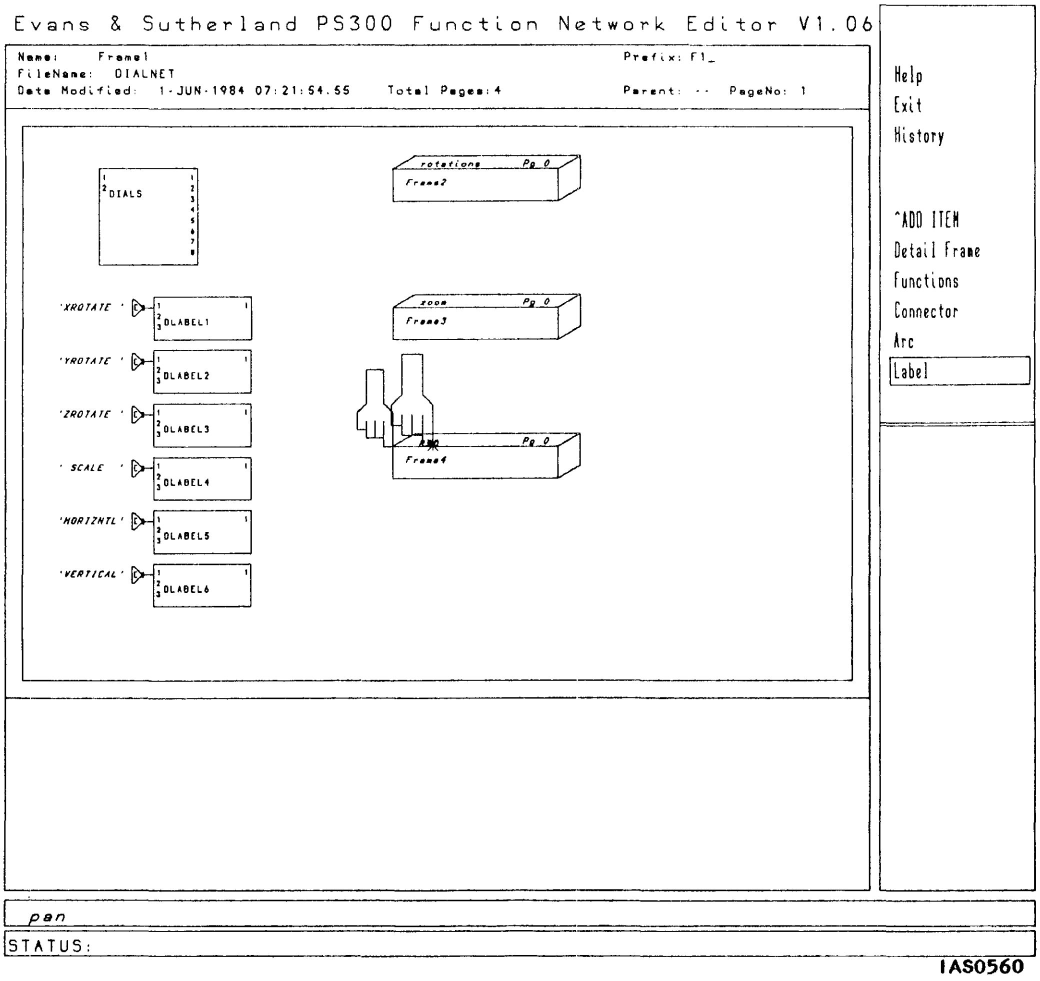
Evans & Sutherland PS300 Function Network Editor V1.06

Name: Frame1 Prefix: F1_
FileName: DIALNET
Date Modified: 1-JUN-1984 07:21:54.55 Total Pages: 2 Parent: -- PageNo: 1

STATUS:

Figure 4-7. Creating Detail Frames

The prefixes and names of detail frames, along with all other text which is displayed in italics, may be edited by selecting the Labels menu item and picking the text to be edited. Text can also be edited by pressing the ARC/TEXT function key twice. This feature can also be used to add "floating" comments. Labels are being added in Figure 4-8.



Evans & Sutherland PS300 Function Network Editor V1.06

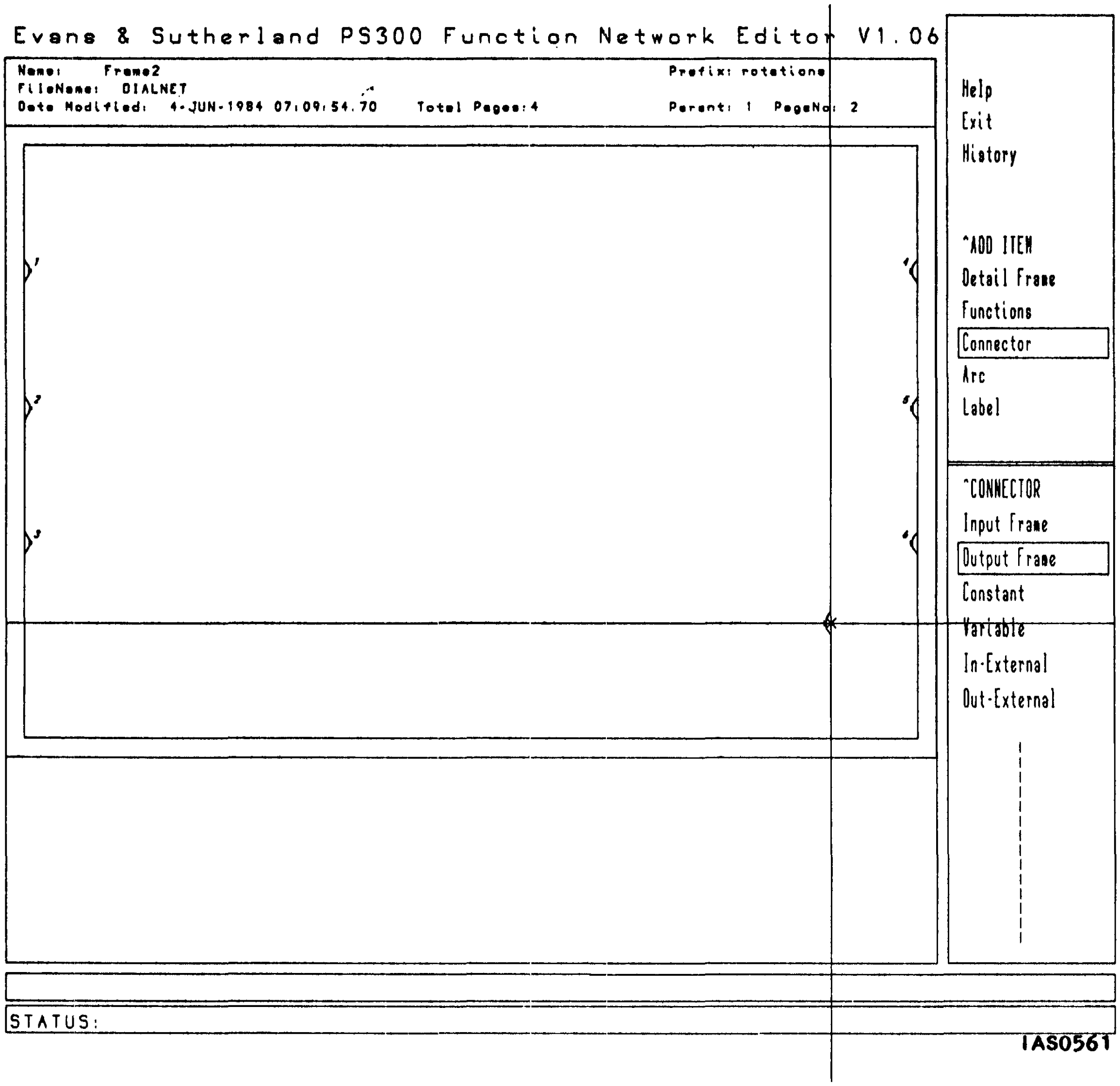
Name: Frame1 Prefix: F1_
FileName: DIALNET
Date Modified: 1-JUN-1984 07:21:54.55 Total Pages:4 Parent: -- PageNo: 1

pan

STATUS:

Figure 4-8. Editing Labels

In Figure 4-9, the GO DOWN function key has been pressed and the "rotations" detail frame has been selected to edit the "inside" of the detail box. This frame will have three inputs (from the dials for X, Y, and Z rotations) and three outputs (for the corresponding rotation matrices). Input and output frame connectors have been placed by selecting the INPUT FRAME and OUTPUT FRAME items from the CONNECTOR menu. These items may be placed only on the left and right edges of the frame, respectively.

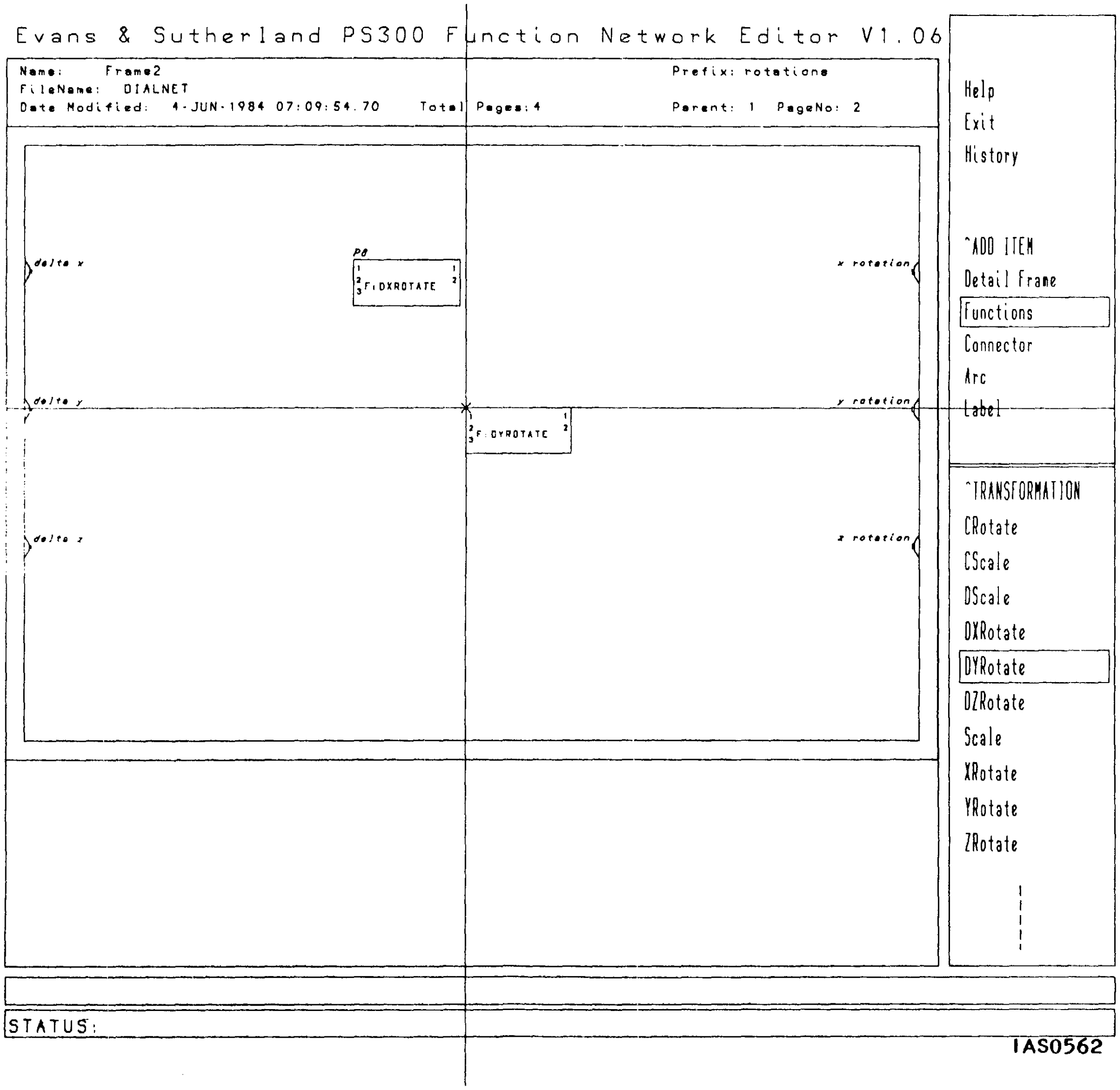

Evans & Sutherland PS300 Function Network Editor V1.06

Name: Frame2 Prefix: rotations
FileName: DIALNET
Date Modified: 4-JUN-1984 07:09:54.70 Total Pages:4 Parent: 1 PageNo: 2

STATUS:

Figure 4-9. Adding Input and Output Frame Connectors to a Detail Frame

In Figure 4-10, the input and output frame connectors have been assigned descriptive names using the LABEL function. Functions are now being placed in the detail frame. Since these are intrinsic functions, they are assigned default names by the Editor as they are instanced. These may also be edited using the LABEL function.



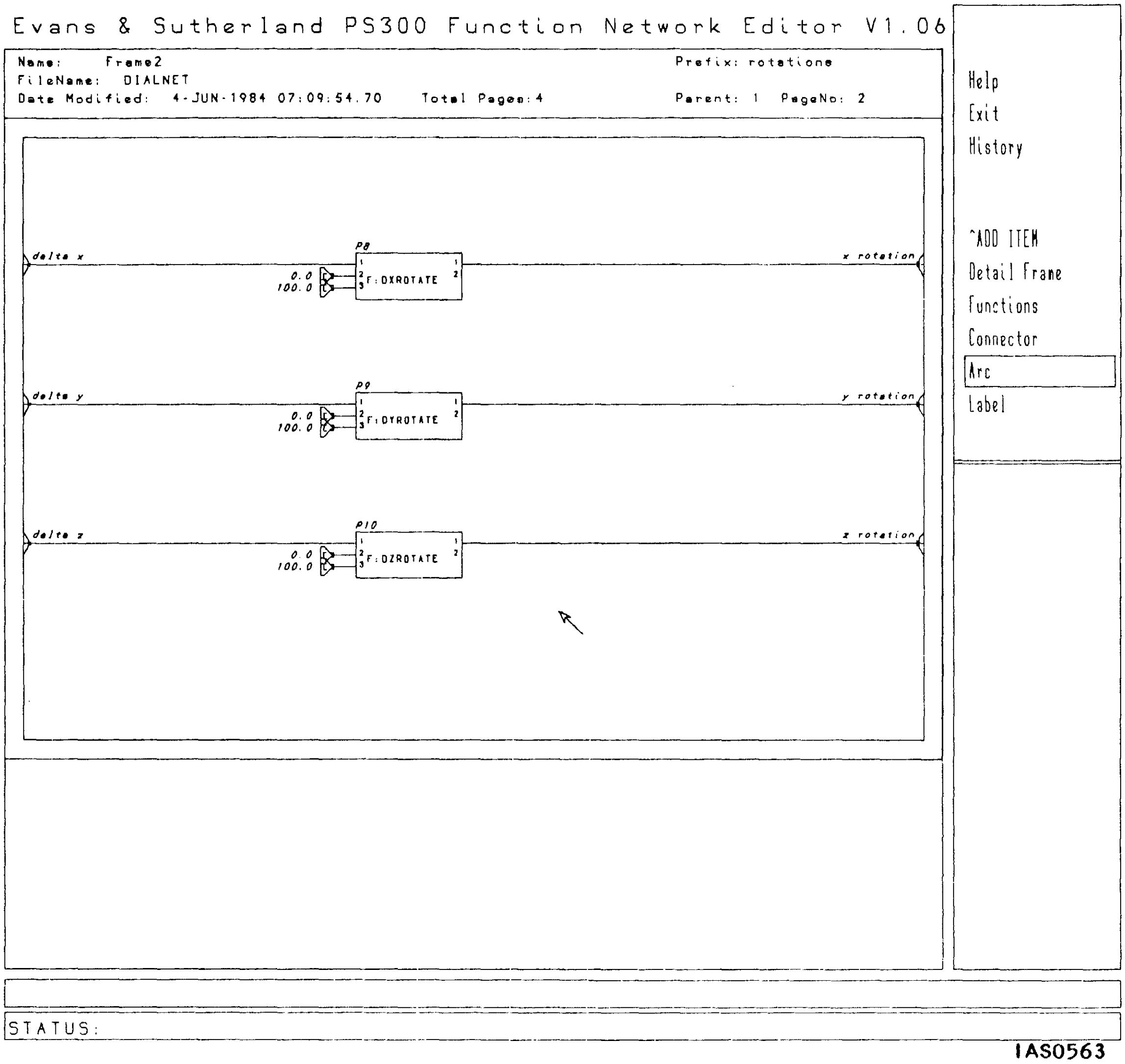
Evans & Sutherland PS300 Function Network Editor V1.06

Name: Frame2
FileName: DIALNET
Date Modified: 4-JUN-1984 07:09:54.70 Total Pages:4
Prefix: rotations
Parent: 1 PageNo: 2

STATUS:

Figure 4-10. Placing Functions in the Detail Frame

Constants and arcs have been drawn. Figure 4-11shows the completed detail frame for handling the rotations.



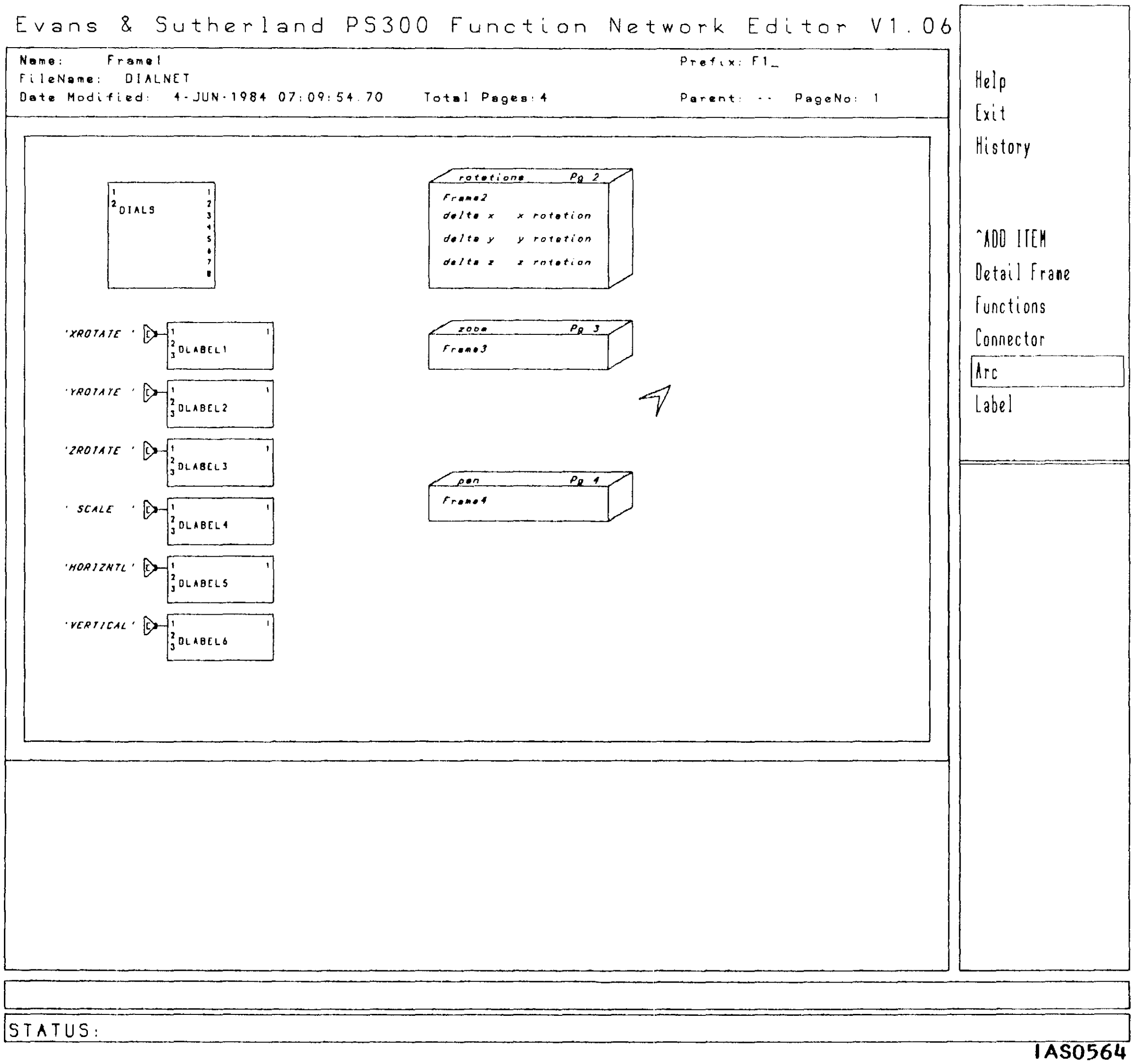
Evans & Sutherland PS300 Function Network Editor V1.06

Name: Frame2
FileName: DIALNET
Date Modified: 4-JUN-1984 07:09:54.70 Total Pages: 4
Prefix: rotations
Parent: 1 PageNo: 2

STATUS:

Figure 4-11. The Complete Detail Frame for Rotations

Figure 4-12 shows that when the GO UP function key is selected, the next higher-level frame in the context tree is displayed again. Notice that the box for the "rotations" detail frame has grown and that the input and output connectors that were placed inside now appear. The connectors appear in the same order at both levels; the size of the detail frame box on the higher level is adjusted automatically, depending on the number of connectors.



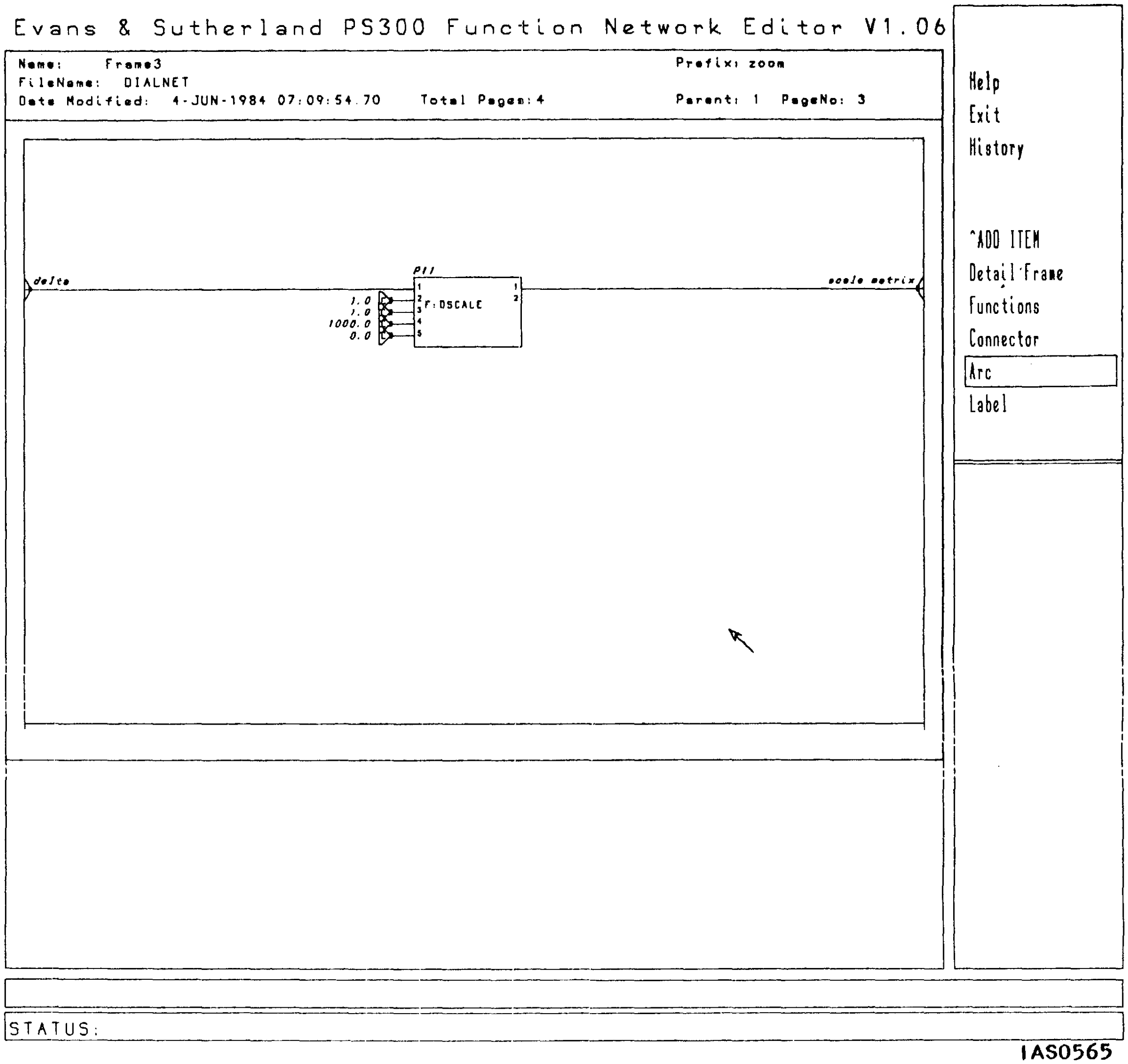
Evans & Sutherland PS300 Function Network Editor V1.06

Name: Frame1 Prefix: F1_
FileName: DIALNET
Date Modified: 4-JUN-1984 07:09:54.70 Total Pages: 4 Parent: -- PageNo: 1

STATUS:

Figure 4-12. The Next Highest Frame in the Hierarchy

The detail frame for “zoom” has been created in much the same way. Figure 4-13 shows the completed frame.



Evans & Sutherland PS300 Function Network Editor V1.06

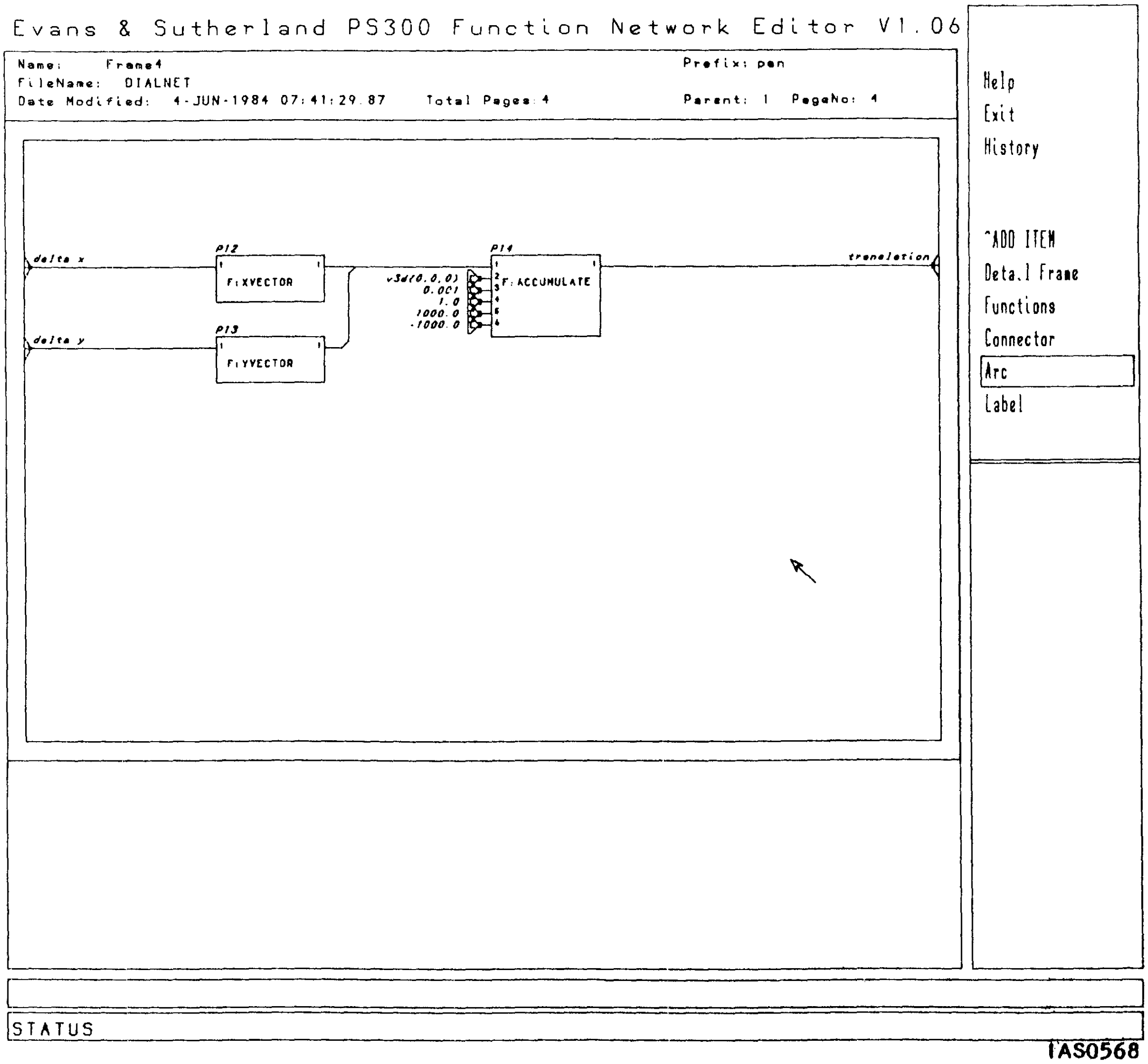
Name: Frame3
FileName: DIALNET
Date Modified: 4-JUN-1984 07:09:54.70 Total Pages:4
Prefix: zoom
Parent: 1 PageNo: 3

STATUS:

1AS0565

Figure 4-13. The Complete Detail Frame for Zooming

Figure 4-14 shows the completed detail frame for "pan"—horizontal and vertical translation. The output of the detail box is a 3D vector.



Evans & Sutherland PS300 Function Network Editor V1.06

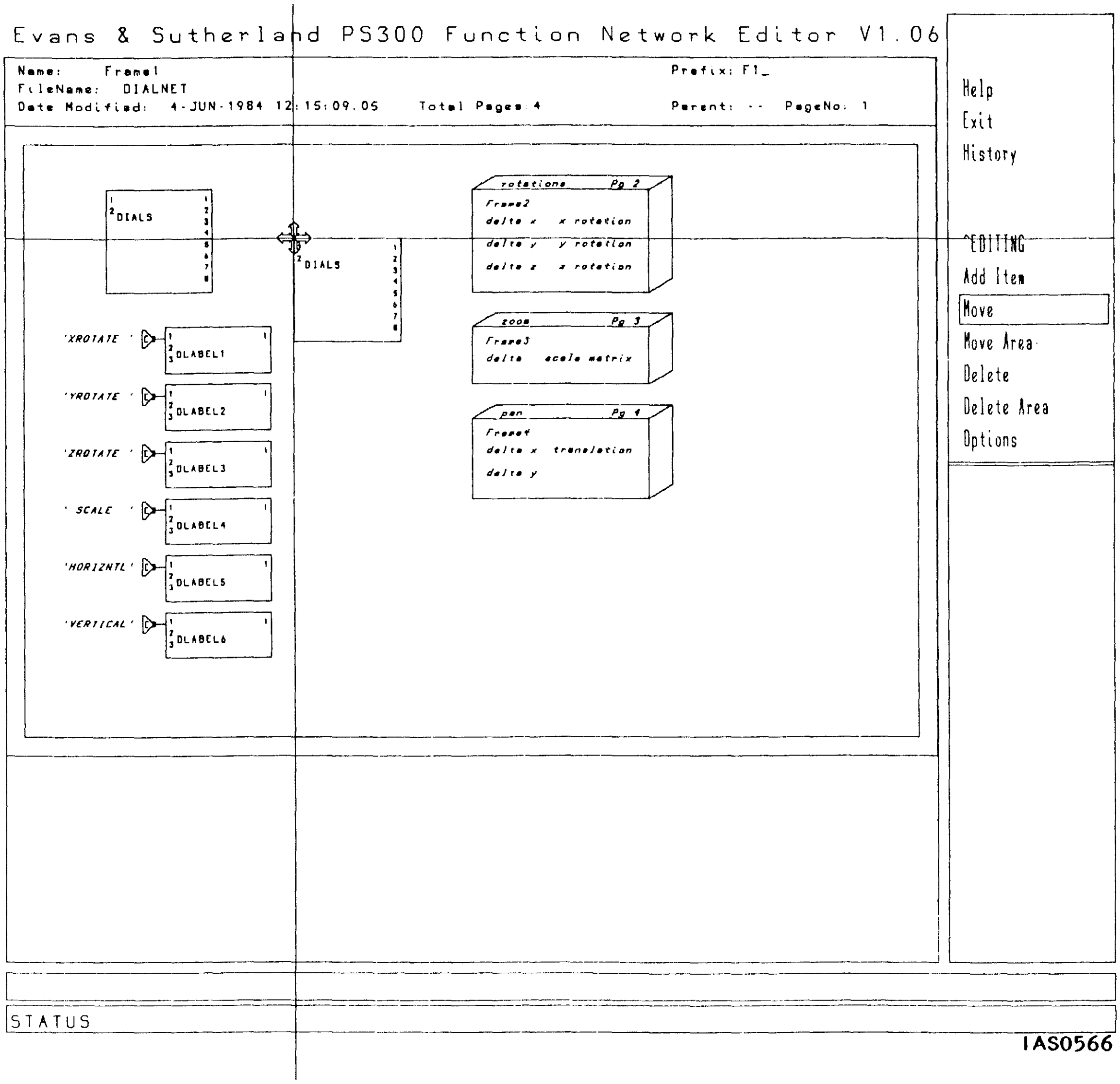
Name: Frame4 Prefix: pen
FileName: DIALNET
Date Modified: 4-JUN-1984 07:41:29.87 Total Pages: 4 Parent: 1 PageNo: 4

STATUS

TAS0568

Figure 4-14. The Complete Detail Frame for Panning

Because the sizes of the detail frame boxes change as they are edited, it is often necessary to adjust the layout of the diagram after they have been completed. In Figure 4-15, the MOVE feature is being used to reposition a function box.

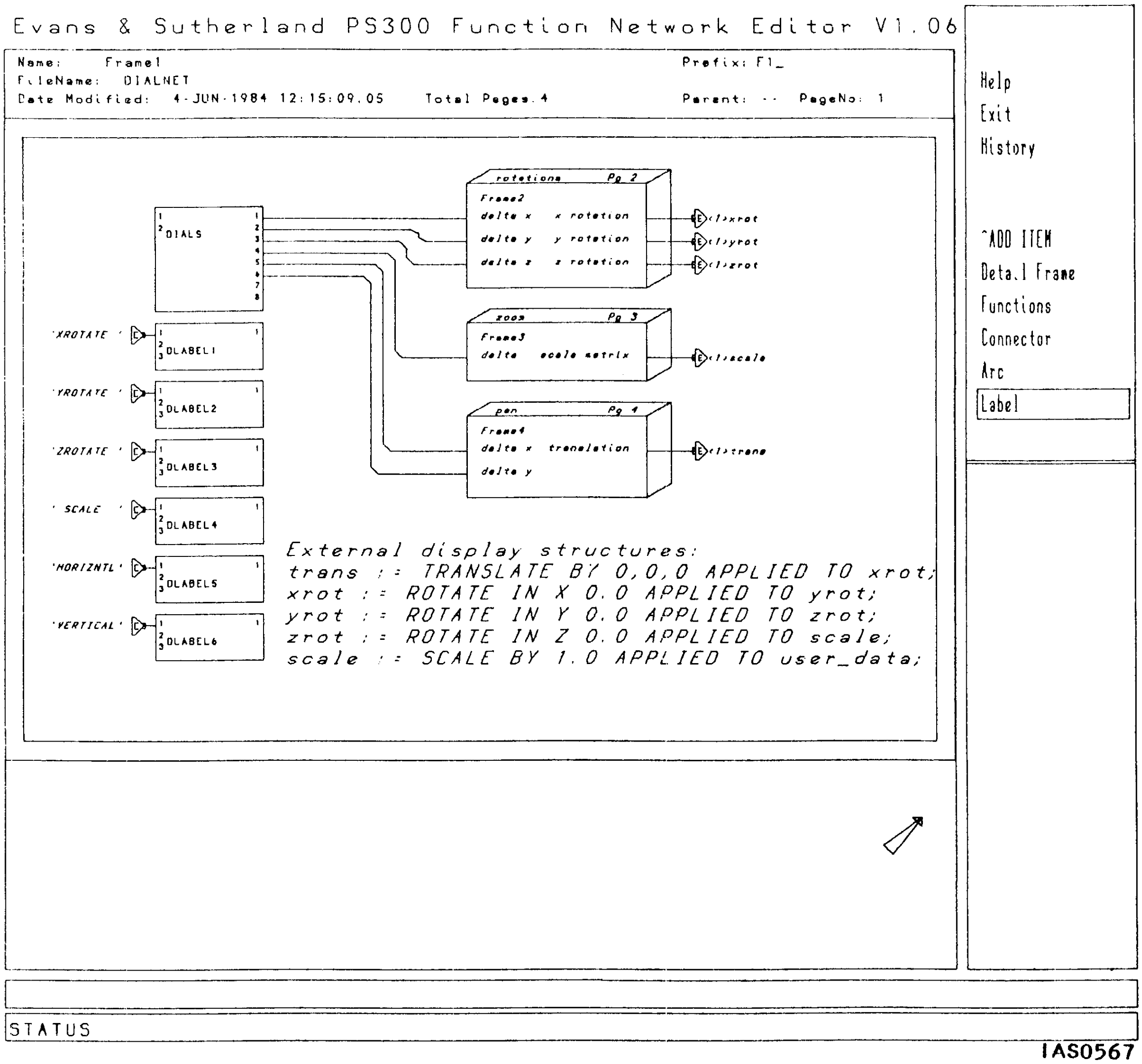


Evans & Sutherland PS300 Function Network Editor V1.06

Name: Frame1 Prefix: F1_
FileName: DIALNET
Date Modified: 4-JUN-1984 12:15:09.05 Total Pages: 4 Parent: -- PageNo: 1

Figure 4-15. Using MOVE to Reposition Items in the Diagram

Figure 4-16 shows the top level frame of the completed diagram. Arcs have been inserted to route the output from the dials to the appropriate places. Connectors to external display structures were added. The floating comment, added using the Labels feature, describes how to set up these display structures.



Evans & Sutherland PS300 Function Network Editor V1.06

Name: Frame1 Prefix: F1_
FileName: DIALNET
Date Modified: 4-JUN-1984 12:15:09.05 Total Pages.4 Parent: -- PageNo: 1

```
External display structures:
trans := TRANSLATE BY 0,0,0 APPLIED TO xrot;
xrot := ROTATE IN X 0.0 APPLIED TO yrot;
yrot := ROTATE IN Y 0.0 APPLIED TO zrot;
zrot := ROTATE IN Z 0.0 APPLIED TO scale;
scale := SCALE BY 1.0 APPLIED TO user_data;
```

IAS0567

Figure 4-16. The Top-Level Frame of the Complete Diagram

Here is a listing of the ASCII code produced by selecting the CONVERT NETWORK menu item.

To generate the file, the following options were selected.

- Use Frame Prefix (off)
- Suppress Comments (off)
- Use Macro Prefix (off)

```
{ Code generated by Network Editor 8 }
{ DIALNET }
{ Frame-Prefix Macro-Prefix }
{ Frame2:rotations }

rotationsP8:=F:DXROTATE;
rotationsP9:=F:DYROTATE;
rotationsP10:=F:DZROTATE;

SEND 100.0 TO <3>rotationsP10;
SEND 0.0 TO <2>rotationsP10;
SEND 100.0 TO <3>rotationsP9;
SEND 0.0 TO <2>rotationsP9;
SEND 100.0 TO <3>rotationsP8;
SEND 0.0 TO <2>rotationsP8;

{ Frame3:zoom }

zoomP11:=F:DSCALE;

SEND 0.0 TO <5>zoomP11;
SEND 1000.0 TO <4>zoomP11;
SEND 1.0 TO <3>zoomP11;
SEND 1.0 TO <2>zoomP11;

{ Frame4:pan }

panP12:=F:XVECTOR;
panP13:=F:YVECTOR;
panP14:=F:ACCUMULATE;

CONN panP12<1>:<1>panP14;
CONN panP13<1>:<1>panP14;

SEND -1000.0 TO <6>panP14;
SEND 1000.0 TO <5>panP14;
SEND 1.0 TO <4>panP14;
SEND 0.001 TO <3>panP14;
SEND v3d(0,0,0) TO <2>panP14;

{ Frame1:F1  }
{ External display structures:}
{ trans := TRANSLATE BY 0,0,0 APPLIED TO xrot; }
{ xrot := ROTATE IN X 0.0 APPLIED TO yrot; }
{ yrot := ROTATE IN Y 0.0 APPLIED TO zrot; }
{ zrot := ROTATE IN Z 0.0 APPLIED TO scale; }
{ scale := SCALE BY 1.0 APPLIED TO user_data; }

CONN DIALS<1>:<1>rotationsP8;
CONN DIALS<2>:<1>rotationsP9;
CONN DIALS<3>:<1>rotationsP10;
CONN DIALS<4>:<1>zoomP11;
CONN DIALS<5>:<1>panP12;
CONN DIALS<6>:<1>panP13;

CONN rotationsP8<1>:<1>xrot;
CONN rotationsP9<1>:<1>yrot;
CONN rotationsP10<1>:<1>zrot;

CONN zoomP11<1>:<1>scale;
CONN panP14<1>:<1>trans;

SEND ´VERTICAL´ TO <1>DLABEL6;
SEND ´HORIZNTL´ TO <1>DLABEL5;
SEND ´ SCALE  ´ TO <1>DLABEL4;
SEND ´ZROTATE ´ TO <1>DLABEL3;
SEND ´YROTATE ´ TO <1>DLABEL2;
SEND ´XROTATE ´ TO <1>DLABEL1;
```

TT5. FUNCTION NETWORK DEBUGGER

TT5. FUNCTION NETWORK DEBUGGER

NETPROBE

CONTENTS

Section TT5

Function Network Debugger

NETPROBE

This software package is distributed by Evans & Sutherland as a convenience to customers and as an aid to understanding the capabilities of the PS 390 graphics systems. Evans & Sutherland Customer Engineering supports the package to the extent of answering questions concerning installation and operation of the programs, as well as receiving reports on any bugs encountered while the programs are running. However, Evans & Sutherland makes no commitment to correct any errors which may be found.

One of the most critical aspects of the PS 390 graphics programmers' job is isolating and correcting problems in function networks. NETPROBE, developed at Evans & Sutherland, can be used as a function network debugger or as a guide in writing your own network debugging program, allowing you to see the data values the network generates as it runs. NETPROBE is written in DEC Version 2 Pascal for use on a VAX/VMS 3.3 and higher system.

The NETPROBE host program works in two stages: it first reads an ASCII function network file and produces a list of the actively used outputs; it uses this list to create the debugging network and display structure for up to 300 outputs. It can then be downloaded on top of the network to be debugged, and data from the function outputs are displayed. The user can optionally edit the list of outputs to reorder or modify the display, or generate a list to focus on particular segments of a function network. Each function output used in the network is displayed on a separate display line in a 15-item page, showing the name of the function and number of the output, the value of the last output, and optionally, a count of how many times the output has fired. Function keys provide control over which of twenty pages are displayed, clear the currently displayed values, and disable the display.

NETPROBE is invoked through a VMS command file (NETUSER.COM) which allows the user to initialize the PS 390, download the ASCII command files, run the Function Network Debugger, and run the Function Network Editor (if installed).

1. Getting Started

Installation instructions for NETPROBE are contained in Appendix A. When all of the files have been installed, run NETPROBE using the NETUSER command file.

To bring up the initial menu of the PS 390 utility programs enter:

```
$@[HomeDir]NETUSER
```

where the name of the directory in which NETPROBE is installed is substituted for HomeDir.

```
            Evans & Sutherland PS 300 Utilities V1.08
                          Initial Menu

      0)  Exit
      1)  Initialize the PS 300
      2)  Send a file to the PS 300
      3)  Run NetProbe - Function Network Debugger (Menu)
      4)  Run NetEdit - Function Network Editor (Menu)
      5)  Character Font Utilities (Menu)
```

Use Option 2 or an equivalent procedure of your own to first download your ASCII function network file to the PS 390. This must be done before the debugging network can be sent to the PS 390. Then select Option 3. You will be presented with the following Debug Menu:

```
 Evans & Sutherland Function Network Utility Command File V1.06
             NETPROBE:   Function Network Debugger

      0. Exit
      1. Prepare a debug module - complete and sorted
      2. Prepare a debug module in stages
      3. Send a debug module to the PS 300
      4. Label the control function keys.
```

Option 1 creates a list of output names and uses the list to create a debugging network, performing both operations in a single pass and producing a sorted debug display for all the outputs in the network. This is useful for small files. The command file prompts you for a file name, runs NETPROBE and outputs a list, and then uses the VAX Sort utility to sort the list. It immediately runs NETPROBE again and prepares the debugger network and display structure. The default extension for input file names is .300, and the extension for the output name list is .PRB.

Option 2 allows you to directly run NETPROBE to generate an output list or to use a list to create a debugging network. You are presented with the following menu:

```
NETPROBE:  Please provide a source file in one of two formats

    Original PS 390 ASCII Network commands (any extension)
        Assumes CONNECTS are contained on single lines and are the
        first non blank words on the line

                                OR

    Output name list: (no extension or .PRB)
        a list of output names and comments (“ ” or “{”)

                                OR

    1. Turn Counting Option ON (OFF)
    Enter filename, Option, or RETURN to exit:
```

If any extension other than .PRB (the output list extension) is used, it is assumed the filename you provide is an ASCII network file and an output list is created. If no extension or .PRB is used, it is assumed you are providing an output list and a debugging network is created from it.

If you enter “1”, the count option is toggled so the debugging network counts the number of times the output is fired and displays the current count. A counted debugging network is slow and should be used only for small numbers of outputs.

After using NETPROBE to generate an output list, you can also edit the list to reduce the number of outputs, improve the quality of printing, or add some outputs. To do this, exit the NetUser command file and using the text editor, edit the .PRB file you have just produced. Re-enter NetUser and the Debug Menu and select Option 2 to create a debugging network using the modified output list.

Option 3 sends both the debugger control network and the debugging network just produced to the PS 390. During the download of the debug structure, status messages appear on the bottom line of the display, including an end message. The debug control network includes the standard support network for any of the debugging structures, and includes the top-level display structure and implementation of the function key controls. You must then press SHIFT/LINE LOCAL (PS 300 Style) or CTRL/LOCAL (PS 390 Style) to enable the function keys properly.

Once the network and the debugging network are downloaded to the PS 390, the function keys can be used to control the debug display as follows:

Shift-FKey 9:	PAGE -	Display previous debug page
Shift-FKey 10:	PAGE +	Display next debug page
Shift-FKey 11:	CLEAR	Clear the current values and counts
Shift-FKey 12:	SHOW Y/N	Enable or disable the debug display

You must press SHIFT and then the associated key. If you are not actively using these function keys for your own function network, you can download labels for the function keys by selecting Option 4.

2. Additional Features

2.1 ASCII Network File - Original Input

NETPROBE reads in PS 390 command files and generates a list of the function outputs that appear in CONNECT statements. For example:

```
CONNECT a<1>:<2>b;
```

produces

```
a<1>
```

in the output list.

The CONNECT commands must be coded on single lines with no comments or other commands preceding the CONNECT command on the same line. For lines in which the CONNECT command follows a comment or another command on the same line, the outputs are not listed. For example, in:

```
{Comment} CONNECT a<1>:<1>b;
```

the output a<1> is not listed.

In:

```
   CONNECT a<1>b; CONNECT b<1>:<1>c;
```

the output b<1> is not listed.

Commands that have been commented out are ignored.

Some versions of Pascal may not tolerate a null line at the end of a file and may produce an error in reading the file. In this case, the last line should contain at least a space.

2.2 Active Output Name File

You can either run NETPROBE in one pass (Debug Menu, Option 1), or in stages (Option 2) which allows you to edit the output name file (.PRB). You might want to edit the output name file to improve the way the debug display appears, to increase the efficiency of the debugging network, and to reduce delays caused by very active networks and frequently sampled peripherals.

The output name file may contain blank lines to separate sets of display items or it may contain comments. Any line beginning with a space or left curly brace ({) is treated as a comment and empty lines (0 characters) are ignored.

Sorting facilitates lookup. The NETPROBE program discards duplicate output names whether the list is sorted or not.

The debug structure may result in a large and slow network bogged down by frequently sampled interactive devices (tablet and dial). To reduce the traffic during debugging, either edit the output name file and remove some of the outputs being monitored, or cut the sampling rate of the tablet and dials.

NETPROBE generates extensive code for each output. With moderate to large networks (over 100 outputs) in which a lot of activity is expected, split the output name intermediate file into sections and create debugging networks for each of these separately for debug sessions focused on different network segments.

The output name file can be used to check for spelling errors by listing its contents.

2.3 Debugging Network

The debugging network and display file are downloaded on top of your network and compete with it for memory, display capacity, function execution, and object names.

Memory can be reduced and function execution enhanced by shortening the output name file to focus on limited sections of the network. All NETPROBE-generated named entities (function and display structures) use a D$ prefix to help reduce naming conflicts, e.g., D$pr_1. Please do not use this prefix in your own files.

Once a debugging network has been passed to the PS 390, to eliminate it use the INIT command (refer NetUser, Option 1) and retransmit the function network.

Appendix A
Installation

Distribution Tape Format and Installation Procedure

PS 390 VAX/VMS sites receive the distribution tape (PS 390 host software) in VMS Backup format. To install the VAX PS 390 host software, first create a subdirectory for the PS 390 software and set your default to that directory. Using the VMS Backup Utility, enter the following commands:

```
$ Allocate   MTNN:
$ Mount/Foreign MTNN
$ Backup MTNN:PSDIST.BCK [...]*.*
$ Dismount MTNN:
$ Deallocate MTNN:
```

where MTNN: is the physical device name of the tape drive being used.

This will create the subdirectory A2V02.DIR which is the parent directory of the PS 390 host software.

UNIX sites will receive a 1600-bpi distribution tape in tar format. IBM sites will receive a 1600-bpi distribution tape with a block size of 6400 and a logical record length of 80.

Customizing the Command Files

To modify the two command files NETPROBE.COM and NETUSER.COM:

1 Using a text editor, search for and change the entries which are marked !*INSTALL-DEPENDENT*. These identify the home directory in which the NETPROBE files are installed.

2 In the NETUSER.COM file, if you don't intend to install the NetEdit file, comment out Option 4 in Top_Menu.

3 Exit from the text editor.

Compiling and Linking

NETPROBE is automatically compiled and linked when NETBUILD.COM is run to install the Function Network Editor. (Refer to Appendix A of

Function Network Editor for instructions on running NETBUILD.COM.) NETPROBE is compiled and linked on its own as follows:

Enter:

```
$@[HomeDir]NETPROBE
```

where the name of the directory in which NETPROBE is installed is substituted for HomeDir.

The Main Menu is presented:

```
Evans & Sutherland Function Network Debugger
       Maintenance Command File V106
                 Main Menu

0. Exit
1. Compile Debugger (NonDebug)
2. Compile Debugger (Debug)
3. Copy Debugger to Tape
Enter selection (0-3)
```

Select Option 1 to compile and link NETPROBE and its utility library. Option 2 prepares a debugging version of NETPROBE if you want to debug modifications to NETPROBE. Option 3 copies the necessary files (listed in NEFileDbg.DAT) to tape for further distribution.

Files that are Loaded

The following is a list of all the NETPROBE files that are loaded from the distribution tape (other files may be loaded that are used for NETEDIT). The files are ordered by logical groupings and in the same way they would appear if you were working in a multiple directory.

Init.300	An initialization file for the PS 390, used by NetUser
NetUser.Com+	The user command file
NELinker.Com	A command file to link NETPROBE after compilation
NEPascal.Com	A conditional Pascal compilation command file
NEFileDbg.Dat	The list of files needed in NETPROBE distribution
NetProbe.Pas	The NETPROBE source program file
NetProbe.Com+	The maintenance command file
NetProbe.300	The debugger control network

NetProbeA.300	Command file to label function keys
NEUtil.Mod	A library of support routines needed by NETPROBE
NEUtil*.Dcl	Shared declarations between NEUtil and NETPROBE

NOTE

The +'d files must be edited upon installation. NEUtil* is shared in common with the Network Editor.

Appendix B
Customization

The NETUSER.COM command file is set up to assume a default extension of '.300'. This can be modified without side effects to meet user conventions.

CAUTION

> In editing the NETPROBE.300 debug control network file, check to see if any portion of the PS 390 commands you intend to alter are referenced in NETPROBE.PAS. The commands are described in the NETPROBE.300 file header. (Refer to Appendix A for a summary of the files included in the distribution tape.)

The following are changes that can easily be made to NETPROBE.PAS:

- Items per page: change the PageSize constant.
- Maximum items: change the MaxProbes constant and also modify the display structure in NETPROBE.300 to include more pages.

NOTE

> If there are more than 512 lines, add more D$clear_all_N functions which can handle 128 outputs each.

- Placement of display structure: change the display structure in NETPROBE.300 and reset the VSpace constant if you are changing the scale of the display.

Appendix C
Porting to Other Machines

NETPROBE is written in Pascal and has been made as machine independent as possible. It is limited to standard Pascal with the following exceptions of DEC Pascal, Version 2:

1. Attributes: Attributes are ANSI extentions to Pascal which qualify how routines and constants are used and shared and include [EXTERNAL], [GLOBAL], [ASYNCHRONOUS], [ENVIRONMENT], and [INHERIT]. These must be edited out if the destination Pascal cannot handle them.
2. Condition handlers: File errors are trapped by a condition handler called OpenError, which helps to recognize nonexistent or protected files and allows the user to try again. This can either be reimplemented if possible, or commented out if you are unable to provide a condition handler and can tolerate a crash on such a condition. Ignore EHandler, another condition handler; it is used only in the Network Editor, NETEDIT.

NETPROBE consists of the debugger source program, NETPROBE.PAS, and NEUTIL.MOD, a library of support functions that it shares with NETEDIT. If you do not have NETEDIT, you do not have to worry about how modifications to NEUTIL will affect it. If you do have NETEDIT, you should rename it if you are going to make extensive changes to NEUTIL.

Some of the routines in NEUTIL are not needed and can be commented out when converting to another version of Pascal or another language. If you are unable to support modules, these two files can be merged together and the associated *.DCL files can be merged into the declaration section to provide a single support file. If you do merge files, then also modify NETPROBE.COM to directly compile and link as a single file rather than compiling and linking in the library.

TT6. DATA STRUCTURE EDITOR

TT6. DATA STRUCTURE EDITOR

STRUCTEDIT

CONTENTS

ILLUSTRATIONS

Section TT6

Data Structure Editor

STRUCTEDIT

STRUCTEDIT is a graphical display structure editor for the PS 390. This program allows you to sketch out your display structure, and then converts the diagram into ASCII PS 390 commands or a routine you can include as part of a FORTRAN, Pascal, C, or LISP program.

Unlike the PS 390 Function Network Editor, described in *TT4*, STRUCTEDIT lets you concentrate on designing the semantics of your PS 390 program, rather than requiring you to spend a lot of time designing the graphical layout of the diagram. STRUCTEDIT does the graphical layout for you, and commands are mostly concerned with conceptual operations such as inserting nodes into the display structure, copying structure, etc.

To support the automatic layout, some restrictions have been made on what you can do. The major one is that each diagram page is drawn as a strict display structure; that is, each node must have exactly one parent. Sharing of structure is achieved through the use of stubs, which are references to names which are defined elsewhere.

Another restriction has nothing to do with layout, but rather with enforcing that your PS 390 program is valid. This is that operation nodes must always be followed by a THEN or APPLIED TO node. (Note that the Graphics Support Routines will signal an error if this is not the case.) In general, STRUCTEDIT will insert stubs when necessary. You can replace the stub with another node.

1. Running Structedit

This section gives some general information about how the user interface works. Specific commands are discussed in sections 2 through 5.

1.1 Files and Pages

When you start up STRUCTEDIT, the first thing it will do is ask you for the name of the file you want to edit. It will assume a file type of ".data" if you do not supply an extension. If it can find the file, it will read it in; otherwise, it will assume you want to create a new file.

A file contains one or more pages. There is no implied hierarchy among the pages; they just provide a convenient way to break up a large program into manageable parts. There is no limit on how much you can fit on a page, although if you try to fit too much on a page it will get scaled down so much you will have a hard time reading the diagram. Using the default diagram scale, you can fit approximately 9 nodes across and 7 nodes down on the screen.

Many of the editing commands make use of a kill buffer. Generally, whenever you delete a large part of your diagram, it is saved in the kill buffer. Until you delete another part, you can paste the deleted subtree back into your diagram from the kill buffer. This also provides a handy mechanism for moving parts of structure between pages, or for making copies of structure.

1.2 Manipulating the Display

The PS 390 screen is divided into three parts. The upper part is used for displaying the current page diagram, the middle part is used for displaying menus, and the bottom of the screen is where messages appear. Figure 6-1 and Figure 6-2 show typical screen displays. You can use dials 1, 2, and 3 to translate and scale the diagram display, and dial 4 to scroll the menu area. (Nearly all the menus are small enough so that you do not need to scroll them.)

The function keys are used to manipulate the display in various ways. Most of them are used to control what menu appears in the menu window. Following is a summary of the function keys.

Key F1 - DATAEDIT

This displays the current fill-in-the-blank menu, used for editing various kinds of data, in the menu window. See the next section for more information on how to use these menus. Normally, the menu will be displayed whenever the host program is expecting you to use it.

Key F2 - BUFFERS

This displays a menu listing the various pages or buffers in the file currently being edited. You can switch to a different buffer by picking its name from the menu.

Key F3 - FILES

This menu lists various commands pertaining to file operations.

Key F4 - EDITING

This menu contains miscellaneous editing commands.

Key F5 - DATANODE

This menu contains commands for inserting the various kinds of data nodes into your diagram.

Key F6 - TRANSFRM

The TRANSFORMATIONS menu contains commands for inserting various kinds of transformation nodes into your diagram, such as SCALE and ROTATE.

Key F7 - COND REF

The CONDITIONAL REFERENCE menu contains commands for inserting various kinds of conditional reference nodes into your diagram. For example, CONDITIONAL_BIT and LEVEL_OF_DETAIL are found here.

Key F8 - ATTRIB

The ATTRIBUTES menu contains commands for inserting miscellaneous operation nodes into your diagram. Most of these nodes set various attributes such as COLOR, PICKING, and so on.

Key F9 - DIAGRAM

The DIAGRAM function key resets the scale and translate for the main diagram window to the defaults for the current page. (These defaults are calculated whenever you select a page from the buffers menu. If you have added or deleted a lot of structure from the current page, reselect the page from the buffers menu to recompute the scale factor.)

Key F10 - DISPLAY

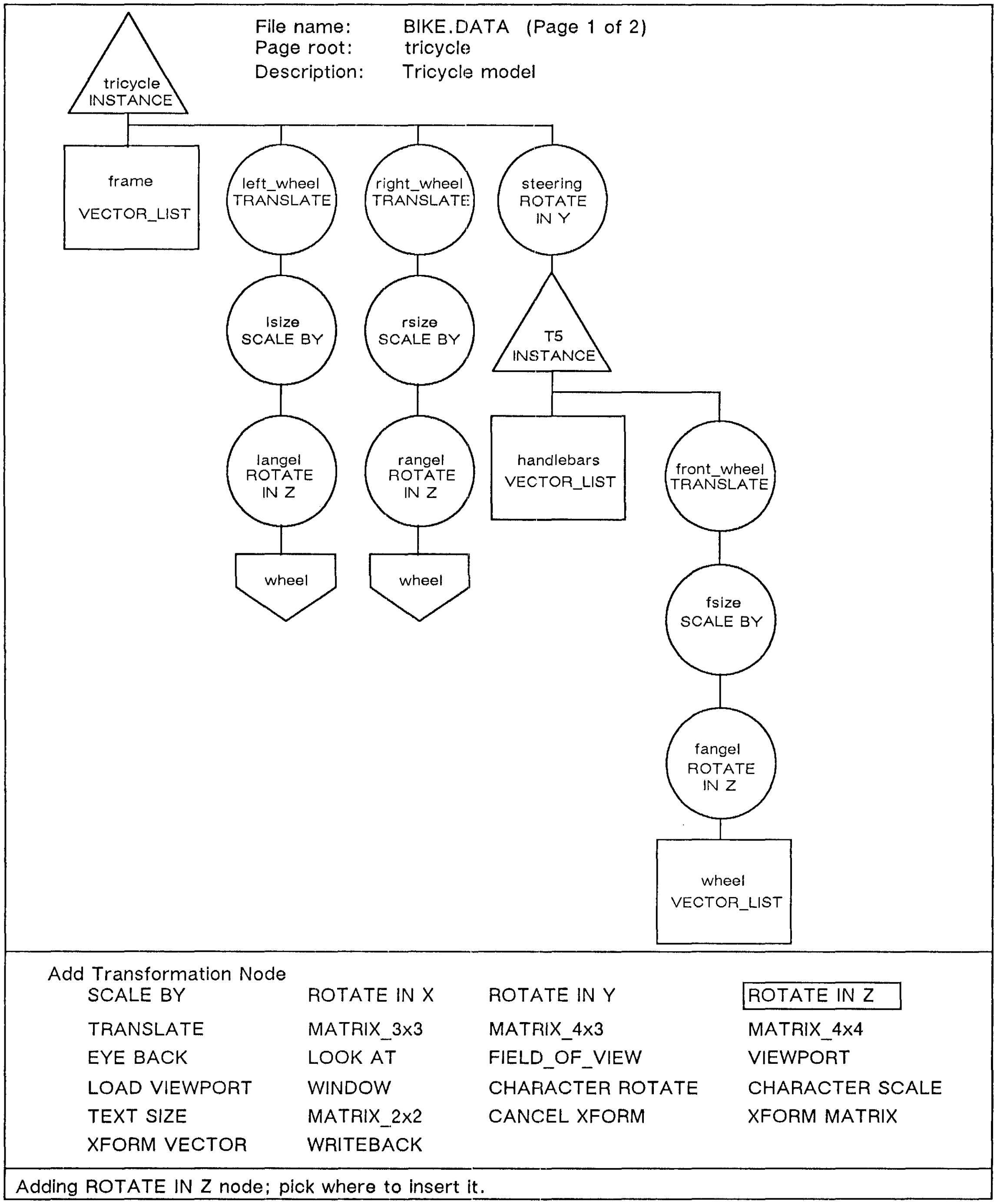
This function key toggles between the normal screen setup and a view of your model. Note that the picture of your model is only updated when you explicitly request it.

Key F11 - FULLMENU

Some of the fill-in-the-blank menus are rather large. You can use this function key to toggle between the normal screen setup and a full-screen view of the fill-in-the-blank menu window.

1.3 Menus

All of the menus controlled by the function keys are command menus. You can always pick a command from one of these menus; this will abort whatever other command is currently in progress. When you pick an item from a command menu, a box will be drawn around it to help you keep track of what the host program is currently doing. Most commands require you to enter some other data on a fill-in-the-blank menu, or to pick things from the diagram display. In addition, some commands loop, or introduce modes. Figure 6-1 shows a typical command menu.



File name: BIKE.DATA (Page 1 of 2)
Page root: tricycle
Description: Tricycle model

Adding ROTATE IN Z node; pick where to insert it.

Figure 6-1. Command Menu Example

Fill-in-the-blank menus are used extensively for things like editing the name and parameters associated with a structure node. When the host is expecting you to edit something in a fill-in-the-blank menu, it will automatically display the menu for you in the menu window. You can use the function keys to return to a command menu.

There are three different kinds of items that can appear on a fill-in-the-blank menu. The first is a multiple-choice menu. An asterisk ("*") will appear next to whatever item in the menu is selected. Picking an item selects it. The second kind of submenu is a toggle menu. This looks like a multiple-choice menu, except that you can have more than one item selected. Picking an item toggles its status. The third type of submenu is a text menu. If you pick an item from a text menu, the keyboard is connected to it so you can type in a new value. If you pick something else while the text menu is active, it will get reset to its initial value and disconnect the keyboard. Typing a carriage return signals completion and also disconnects the keyboard. If there is more than one text item in the menu, the keyboard will automatically connect itself to the next item at this point, to reduce the amount of switching between the keyboard and tablet you need to do.

The text editor uses the following control characters for editing effects:

CONTROL-A Moves the cursor to the beginning of the line.

CONTROL-B Moves the cursor back (left) one character.

CONTROL-D Deletes the character at the cursor position.

CONTROL-E Moves the cursor to the end of the line.

CONTROL-F Moves the cursor forward (right) one character.

CONTROL-K Kills (deletes) to the end of the line.

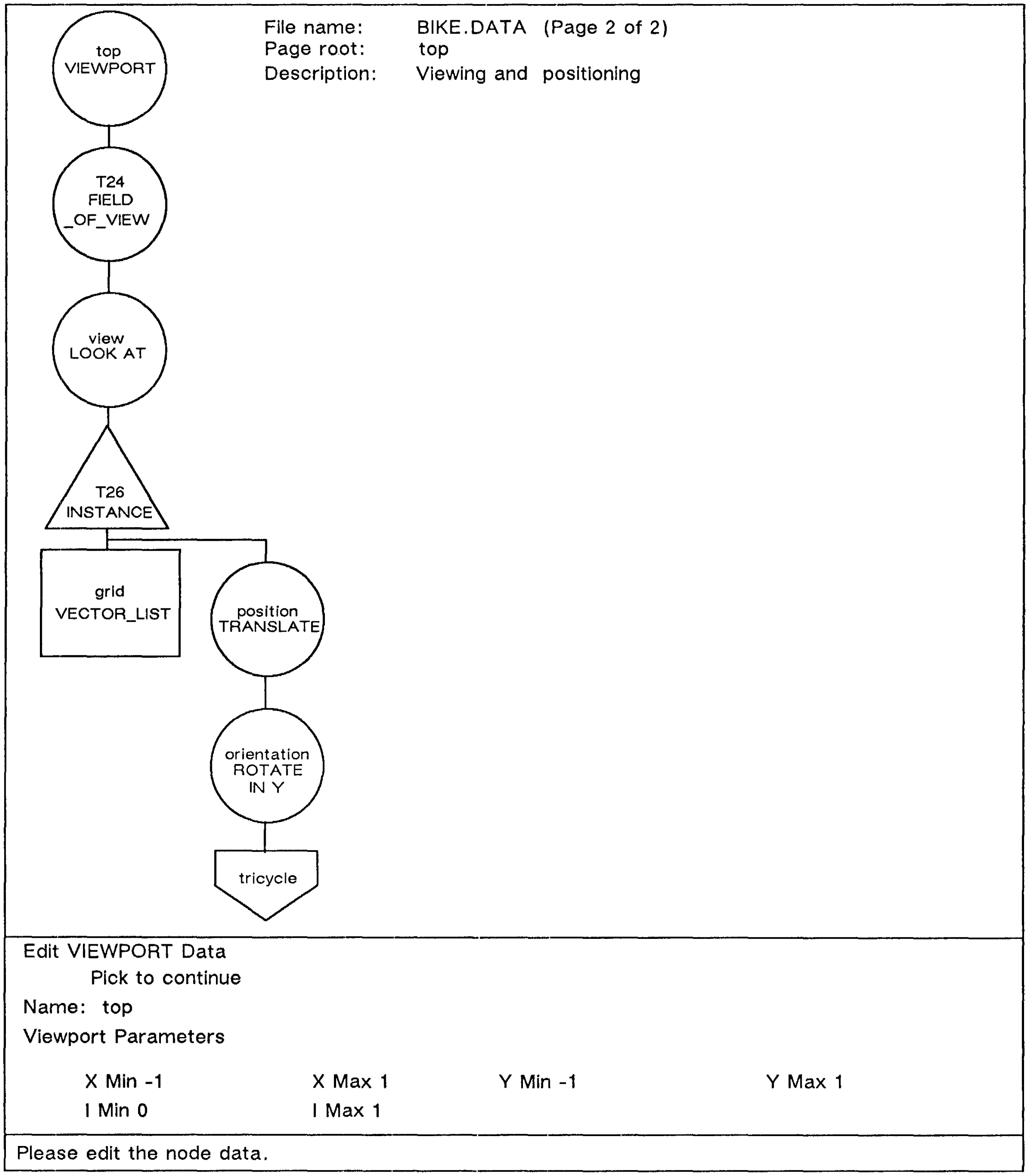
CONTROL-U Deletes the entire line.

DELETE Deletes the character to the left of the cursor.

RETURN Signals completion and disconnects the keyboard.

Fill-in-the-blank menus generally have an item marked "Pick this to continue" at the top. Picking that item indicates that you are satisfied with all of the values you have been asked to edit, and allows the command to complete. Until then, you can go back and change values you have set previously. A fill-in-the-blank menu for editing the parameters associated with a viewport node is shown in Figure 6-2.

In a few cases, STRUCTEDIT needs you to supply just a single string. Here, you do not have to pick anything, and the host program will take whatever you have typed in as soon as you hit the carriage return.



```
File name:      BIKE.DATA  (Page 2 of 2)
Page root:      top
Description:    Viewing and  positioning
```

```
Edit VIEWPORT Data
        Pick to continue
Name:  top
Viewport Parameters

       X Min -1          X Max 1        Y Min -1          Y Max 1
       I Min 0           I Max 1

Please edit the node data.
```

Figure 6-2. Fill-in-the-Blank Menu Example

1.4 Helpful Hints

Many editing commands require you to pick nodes from the diagram display window. In this case, you will see a message indicating that you should do so.

The cursor shape also indicates what input mode you are in. An arrow shape pointing to the left and up indicates that the only valid actions are picks from a command menu. An arrow shape pointing to the right and up indicates that the program is waiting for a diagram pick. An asterisk-shaped prompt indicates that there is a fill-in-the-blank menu active.

Occasionally, you will see a message indicating that garbage collection is in progress, and the program will seem to go dead for several seconds. This is completely normal; the program is just recycling its heap storage. Another message will be displayed when the garbage collection is finished.

1.5 Init Files

You can use an init file, SYS$LOGIN:STRUCTEDIT.INI, to alter some of the parameters used by the editor. A typical use of the init file is to configure the Graphics Support Routines if you are using something other than an async line.

The structure editor is written in LISP so you can actually put any random LISP code you want into your init file. However, the following are the only things that are really useful.

(setq *device-type* ´async)
This establishes the device type for use by the Graphics Support Routines. Valid values are ´async, ´parallel, or ´ethernet.

(setq *device-name* "tt:")
This establishes the name of the device used by the Graphics Support Routines.

(setq *add-node-edit* nil)
Normally, as you insert nodes into your diagram, you will be given a chance to edit the name and other parameters associated with each node. If you prefer to add all of the nodes first and then go back and edit the data as necessary, you can use this option. The new nodes will get unique names and appropriate defaults for their parameters.

```
(setq *ascii-file-type* "300")
(setq *pascal-file-type* "pas")
(setq *fortran-file-type* "for")
(setq *c-file-type* "c")
(setq *lisp-file-type* "cl")
```

These variables provide default values for the file type (or extension) used by the various code conversion options. If you want to use some other extension, just provide another value for the appropriate variable.

1.6 Converting Existing Models

Your existing .300 files can be converted to data files that the structure editor can understand using the PS 390 ASCII-to-GSR conversion program described in Section *TT8*. Specify an output format of DATA.

The conversion process basically works by trying to insert as many nodes as possible on each diagram page. Nodes that are defined but never referenced become the root nodes of the pages. You will probably want to rearrange some parts of the diagram to get a clearer picture.

PS 390 commands that do not represent namable objects are thrown away with a warning message. However, the converter does understand BEGIN...END and BEGIN_STRUCTURE...END_STRUCTURE sequences.

You should avoid converting very large models, particularly if there are long vector lists or polygon lists. In these cases, the model should be broken into smaller files.

2. Command Descriptions — Buffers

The BUFFERS menu allows you to select the current page from the pages you have defined in the file you are currently editing. The menu lists the name of the root node and the description text for each page in the file. Pick the name of the page you want to switch to.

Selecting a page from the buffers menu displays that page and also recomputes the default scale factor used by the DIAGRAM function key.

3. Command Descriptions — Files

The FILES menu contains various commands pertaining to file operations.

READ FILE

This command prompts you for the name of the file to read. The file currently being edited is saved, then the new file read in.

WRITE FILE

This command writes out a backup copy of the file currently being edited. It is suggested that you save your work from time to time, since the structure editor does not have any kind of journaling or autosave feature. It does recover automatically from most program errors. It will simply abort whatever command was in progress when the error occurred, and return you to the command loop.

MERGE FILE

This command is used to merge or include another diagram file into the current file. The pages in the included file are simply added to those already defined in the current file; no checks are made for name conflicts. It prompts you for the name of the file to read.

RENAME FILE

Use this command if you want to change the name of the current file. This command does not actually rename the file, but simply changes the name that will be used by WRITE FILE or EXIT.

PLOT CURRENT PAGE

This causes a screen dump of the current diagram page to be sent back to the host for plotting. The plot is written to a file with the same name as the file being edited, but with an extension of "plot."

The scale and translation are not reset; the diagram will appear in the plot just as it does on the screen. Menus are not plotted.

Particularly if you have a lot of structure displayed on the page or are using the RS-232 async interface, transferring the plot dump to the host can be a rather slow process. The program beeps when it is finished.

The plot data is formatted in such a way that it can be easily processed by an external plot program. Each point is written to the plot file as two integer coordinates (X ranging from 0 to 2398, and Y from 0 to 2998), plus a 0 for move and 1 for draw. You can change the scale factor from the default by sending a 2D vector to <2>m$_P315.

PLOT ENTIRE FILE

This command causes a plot dump for each page in the current file to be written. The plot for each page is written to a file with the same name as the current file, but with extensions of "plot1," "plot2," etc. The scale and translation are reset for each page, so that the entire contents of each page will appear in the resulting plots. The program beeps when all of the pages have been plotted.

CODE CONVERSION

This command is used to produce a file containing ASCII PS 390 commands or a FORTRAN, Pascal, C, or LISP routine containing calls to the Graphics Support Routines library, that you can use to download the model you have created with the structure editor. You will be asked to select what kind of output you want.

The output code will be written to a file with the same name as the file being edited, but with a different extension: "300" for ASCII, "pas" for Pascal, "for" for FORTRAN, "c" for C, and "cl" for Common LISP. (You can change the defaults in your init file; see above.) The output file contains a single procedure, with the same name as the file.

The code generator uses a postorder traversal so that code is generated from the bottom up. That is, the leaf nodes are defined first, then the nodes that reference them, and so on up to the root node of the page. The reason for this is that if you have OPTIMIZE STRUCTURE turned on when the model is downloaded, the PS 390 cannot make any optimizations unless the descendants of a set node are defined before the set node itself. Note that the code generated by the structure editor does not turn on OPTIMIZE STRUCTURE; that it is not possible for one part of the structure defined in a single file to be optimized and another part not; and that the order in which pages are processed is not guaranteed.

EXIT (saving file first)

This saves the current file, then exits the program. You will be asked to verify that you want to exit.

QUIT (without saving)

This exits the program without saving the current file. You will be asked to verify that you want to quit.

DOWNLOAD FILE TO PS 390

This command, which downloads an ASCII file to the PS 390, is useful in conjunction with the DISPLAY MODEL command. An extension of ".300" is assumed if you do not supply one explicitly. The file should not contain muxing bytes, as it is downloaded through the Graphics Support Routines. This option should only be used to download display structure definitions. Downloading a function network may interfere with the operation of the structure editor.

4. Command Descriptions — Editing

The EDITING menu contains the following miscellaneous editing commands.

DELETE NODE

This is a splicing delete; a single node is deleted from the diagram. The deleted node is not saved in the kill buffer. You will be asked to pick a node to delete. This command loops; you will remain in delete mode until you pick another command.

The behavior of this command depends on what kind of node you are deleting. If you delete a data node, STRUCTEDIT will replace it with a stub to maintain the correctness of the structure. If you delete an operation node, it will just splice the display structure tree together around the deleted node. This also happens if you delete an instance node that does not have more than one descendant. (You cannot delete an instance node with several descendants because it is generally not possible to splice in more than one descendant node. Use CUT SUBTREE or REMOVE FROM in this case.)

CUT SUBTREE

This command moves the subtree rooted at the selected node to the kill buffer, and replaces it with a stub with the same name as the root node of the deleted subtree. (This makes it convenient if you are using CUT and PASTE to move a subtree to a new page; the original display structure still points to the same name.) This command loops, so you will be asked to pick another subtree.

PASTE SUBTREE

This command replaces the subtree rooted at the selected node with a copy of the subtree in the kill buffer. The original subtree is not saved in the kill buffer; this allows you to insert multiple copies of a display structure at various places in your diagram. Note that the names of the nodes copied from the kill buffer are not changed, so you may need to edit the names if you are inserting multiple copies of structure. Again, this command loops.

Insert INSTANCE Node

This command is used to insert an instance node into your diagram. See the discussion of adding nodes in 5..

INCLUDE IN

Use this command to add another descendant (a stub) to an instance node. This command will loop, asking you to pick another instance node. The stub will be added as the rightmost descendant of the instance node.

Note that it is possible to change the the left-to-right order of the descendants of a set node, if you wish to do so. First use INCLUDE IN to add a new node on the far right. Then use REMOVE FROM and PASTE SUBTREE to replace the stub with one of the other dependants of the set node.

REMOVE FROM

Use this command to delete a descendant subtree from an instance node. The deleted subtree is saved in the kill buffer. Again, this command loops, asking you to pick the root node of the subtree to be removed.

CHANGE NAME PREFIX

When you add nodes into your diagram, the structure editor gives them default names generated from a prefix and a counter. The default prefix is T, so the names look like T1, T2, etc. You can give a file a different prefix string using this command. Note that the prefix only affects subsequent nodes added, not those already part of the diagram. Use EDIT NODE DATA to change the names of these nodes.

EDIT NODE DATA

This command allows you to edit the name and other data associated with a node. A fill-in-the-blank menu will be displayed after you have picked the node you want to edit. This command loops, asking you to pick another node to edit.

DISPLAY MODEL

This command downloads commands to the PS 390 to display your structure. You can use function key 10 to toggle between the model display and the usual menu/diagram display. Note that the model display is only updated when you specifically request it using this command.

Since the support network and display structures of the structure editor share the same name space as the display structures in your model that are downloaded by DISPLAY MODEL, you must exercise some caution to avoid conflict with the structure editor when you use this feature. To make this easier, all of the names used by the structure editor are prefixed with "m$."

CREATE NEW PAGE

This creates a new diagram page in the current file, and makes it the current page. The page contains a stub as its root node. You will be given a chance to edit the description of the page.

DELETE CURRENT PAGE

This command deletes the current page. Its contents are moved to the kill buffer. You will be switched to another page in the same file. Note that every file must have at least one page. If you try deleting the only page in your file, it will create a new page containing only a stub and switch you to that instead.

REDRAW CURRENT PAGE

If the display becomes corrupted, use this command to redraw the current page.

EDIT PAGE DESCRIPTION

This command allows you to edit the description of the current page.

FIND NAMED NODE

This command prompts for a node name, and switches to the page where the node with that name is defined. Note that stubs are name references, not definitions, and are ignored in the search.

Wildcard characters "%," "*," and "+" may appear in the search string, to match any single character, a sequence of zero or more characters, or a sequence of one or more characters, respectively. A backslash acts as an escape character to allow these characters to be searched for as literals.

5. Command Descriptions — Adding Nodes

The DATANODE, TRANSFRM, COND REF, and ATTRIB menus contain commands for inserting nodes into your diagram. You will be asked to pick the place in the diagram where you want the node to be inserted, and then a fill-in-the-blank menu will let you edit the name or other data associated with the node. These parameters are initialized with reasonable default values (including an automatically generated name), so in many cases you will not need to change anything.

These commands loop; that is, after you have completed the operation, it will ask you to insert another node of the same type.

The exact behavior of these commands depends on what kind of node you are inserting. If you are adding an operation or instance node, it gets spliced into the diagram just in front of the node you selected. If you are adding a stub or data node, the subtree rooted at the node you selected is deleted (and placed in the kill buffer), and the entire subtree is replaced with the new node.

The editing menus provided by the structure editor only allow you to modify values, and, for data nodes such as vector list and labels, add or delete items from the end of the command. Since data node definitions tend to contain a lot of textual information and very little structure, you will probably find a text editor a better alternative for defining large vector lists or polygon lists. You can reference these externally defined names inside the structure editor using a stub.

A few of the data node menus also put slight restrictions on syntax. For BSPLINE and RATIONAL BSPLINE, you cannot specify an explicit knot vector (the default knot sequence is always assumed). For POLYGON lists, you can only specify ATTRIBUTES, OUTLINE, and whether or not you want to use vertex normals and color-by-vertex at the beginning of the polygon list, and these values apply to all polygons within the list.

TT7. CHARACTER FONT EDITOR

TT7. CHARACTER FONT EDITOR

MAKEFONT

CONTENTS

ILLUSTRATIONS

Section TT7

Character Font Editor

MAKEFONT

This software package is distributed by Evans & Sutherland as a convenience to customers and as an aid to understanding the capabilities of the PS 390 graphics systems. Evans & Sutherland Customer Engineering supports the package to the extent of answering questions concerning installation and operation of the programs, as well as receiving reports on any bugs encountered while the programs are running. However, Evans & Sutherland makes no commitment to correct any errors which may be found.

MAKEFONT is a program that allows character fonts for the PS 390 to be designed or modified. Files may be read and written in formats for both standard fonts (the default font loaded when the PS 390 is booted) and user-defined alternate character fonts (a BEGIN_FONT ... END_FONT sequence). MAKEFONT itself runs on a PS 390 under VAX/VMS.

Either a 128-character or 256-character font may be created. There are features allowing merging or modification of existing fonts, as well as for creation of new characters. In addition, files can be read and written in the format used for user-defined alternate fonts.

This document provides descriptions of the commands available within MAKEFONT. It also provides a detailed description of the standard font file format and instructions for downloading standard fonts.

The files needed to run MAKEFONT are loaded from the same distribution tape as the NETEDIT (Function Network Editor) and NETPROBE (Function Network Debugger) files. For installation instructions, refer to Appendix A of Section *TT4*.

1. Running MAKEFONT

MAKEFONT is run from the command file NETUSER.COM. To bring up the menu from which MAKEFONT is selected, enter the following command.

```
[HomeDir]NETUSER.COM
```

[HomeDir] is the name of the directory in which MAKEFONT resides. The following menu is displayed.

```
               Evans & Sutherland PS 300 Utilities V1.08
                             Initial Menu

   0) Exit
   1) Initialize the PS 300
   2) Send a file to the PS 300
   3) Run NetProbe - Function Network Debugger (Menu)
   4) Run NetEdit - Function Network Editor (Menu)
   5) Character Font Utilities (Menu)
```

Select option 5 to bring up the Character Font Utilities Menu.

```
               Evans & Sutherland PS 300 Utilities V1.08
          MakeFont: PS 300 Character Font Utilities Menu

   0) Exit
   1) Run MakeFont character font editor program
   2) Convert standard font file to PS 300 S-record format
```

Select option 1 to start the MAKEFONT program. Option 2 prompts for the name of a standard font file and produces an S-record file (a file in a format which the PS 390 can read). This file can then be downloaded to diskette as the default character font. This is explained in Section 4.

When MAKEFONT is run, it first downloads menus and initializes the font. A "Ready" message appears when the host has completed the initialization, but because of delays due to things such as buffering, it takes about five minutes for the PS 390 to be ready to accept commands. The program is actually "Ready" when it responds to menu picks.

2. Main Control Menu

The main control menu for MAKEFONT consists of a character selection grid on the top half of the screen, and text strings representing various functions on the bottom half of the screen. Although only 128 characters at a time can be displayed on the character selection grid, you can edit a font with 256 characters by using function key F1 to toggle between the display of characters 0–127 and 128–255. (This action is carried out locally, so this has no effect on what MAKEFONT is doing on the host.) Figure 7-1 shows the Main Control Menu when MAKEFONT is first entered.

Use the data tablet and stylus to select menu items from the bottom half of the screen. Some of the functions available (DISCARD, DELETE, COPY, SWAP, and EDIT) require you to select one or more characters from the grid to operate on. For example, if you pick the menu item DELETE, you will be prompted to select a character to delete. You may then pick any number of characters from the selection menu to be deleted. To stop deleting, pick another function from the bottom menu. The remaining functions perform a single action; when complete, MAKEFONT will return to the "Ready" state.

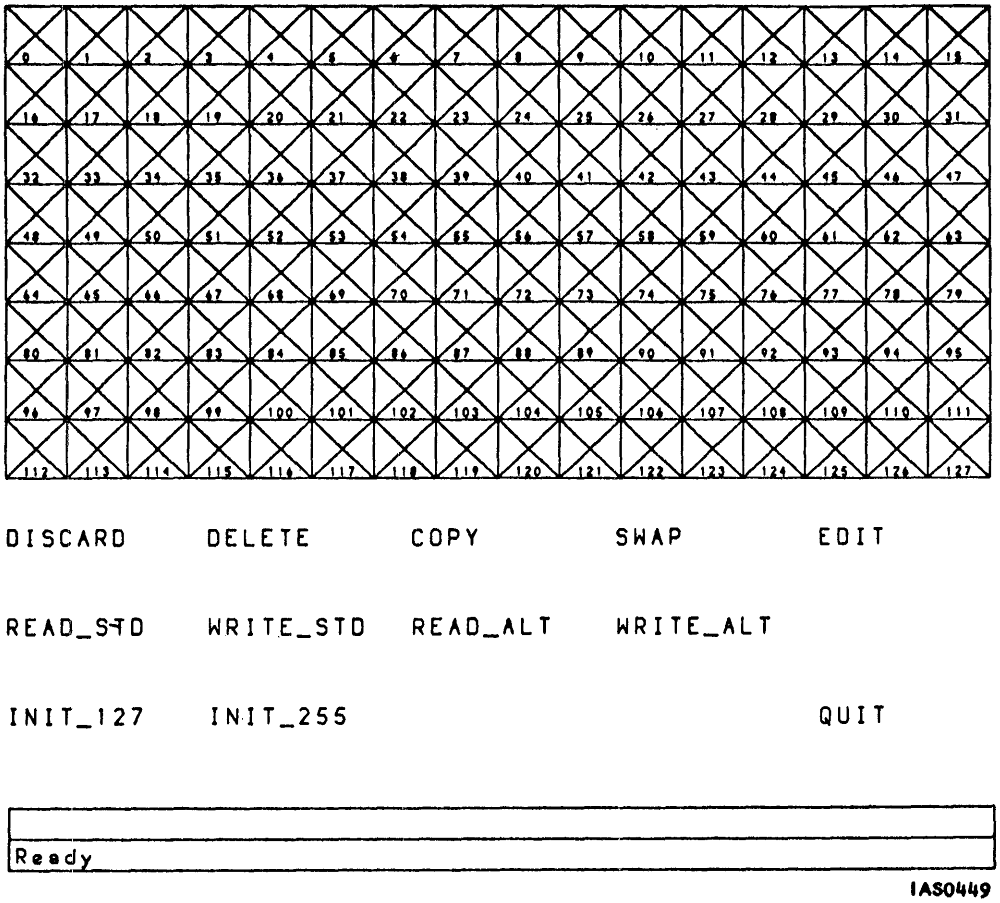


Figure 7-1. The Main Control Menu

The Main Control Menu functions are described below.

DISCARD

This function is useful for combining characters from two or more fonts into a single font. When a character font is read in from a file, only those characters in the current font which are marked for discard will be overwritten by the character definitions being read in. A discarded character has a large 'X' drawn through it on the character selection menu. When the DISCARD menu item is picked, the discard flags for all characters are reset. Selecting a character while in DISCARD mode complements its current discard status.

DELETE

This function is used to delete character definitions from the current font. Any characters which are picked while in DELETE mode are removed from the font.

COPY

The COPY function is used to duplicate a character definition at a location corresponding to a different ASCII code. After selecting COPY, you will be prompted to select the character you wish to copy, and then to select the character location to copy it to. When the copy is complete, you will be prompted to select another character to copy.

SWAP

The SWAP function is used to exchange two character definitions. After selecting SWAP, you will be prompted to select the two characters you wish to exchange, and the character stroke definitions for the two characters will be interchanged. As with the COPY function, when the swap is complete, the operation may be repeated.

EDIT

The EDIT function allows you to define or modify the stroke definition of a character. After selecting EDIT, select the character to be modified. The character edit menu (described in Section 3) will then appear. After picking EXIT or RETURN on the edit menu, you will be returned to the main control menu; at this point, another character may be selected for editing.

READ_STD

This function reads a standard font from a file. Only characters which are marked for discard will be overwritten. After selecting the function, you will be asked for the name of the file; type in a valid VAX/VMS file specification or a logical name.

WRITE_STD

Selection of this menu item writes the current font to a file, using the format for standard fonts. After selecting the function, you will be asked for the name of the file; type in a valid VAX/VMS file specification or a logical name.

READ_ALT

This function reads a user-defined alternate font from a file. The file is assumed to contain a single BEGIN_FONT ... END_FONT command. Only characters which are marked for discard will be overwritten. After selecting the function, you will be asked for the name of the file; type in a valid VAX/VMS file specification or a logical name.

WRITE_ALT

Selection of this menu item writes the current font to a file, using the BEGIN_FONT ... END_FONT format. After selecting the function, you will be asked for the name of the file; type in a valid VAX/VMS file specification or a logical name.

INIT_127

This function initializes a 128-character font (containing definitions for characters corresponding to ASCII 0 to 127). All characters are deleted and marked for discard when this function is selected. (This happens automatically when MAKEFONT is started.)

INIT_255

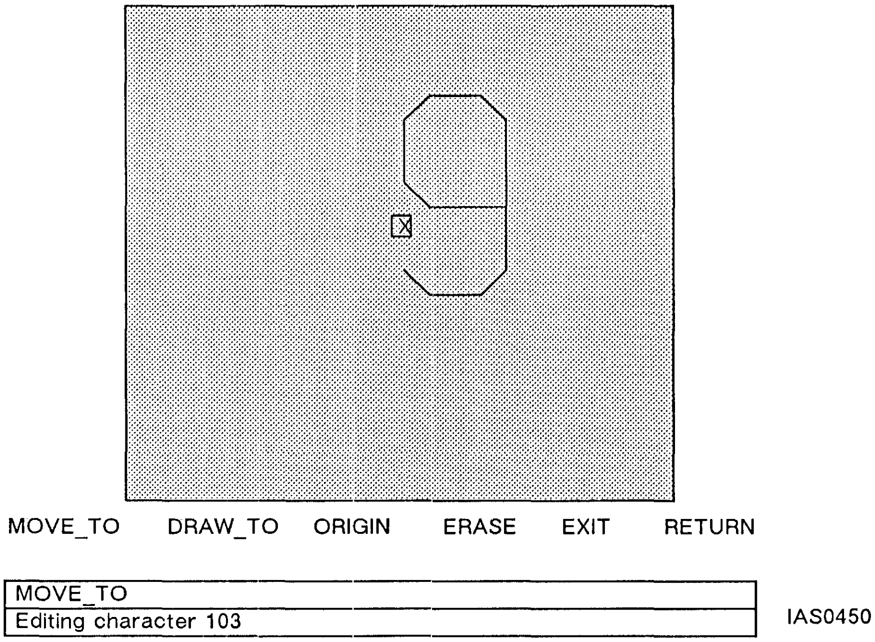
This function initializes a 256-character font (containing definitions for characters corresponding to ASCII 0 to 255). All characters are deleted and marked for discard when this function is selected.

QUIT

Selection of this menu item causes MAKEFONT to terminate; control is returned to the operating system.

3. Edit Menu

A separate menu is used to design individual characters in EDIT mode. This menu consists of the character design grid on the upper part of the screen, and text strings representing various functions on the lower part of the screen. The Edit Menu is shown in Figure 7-2.



MOVE_TO
Editing character 103

Figure 7-2. The Edit Menu

The design grid coordinates range from –64 to 64 in both X and Y. (This is because of the way the standard text fonts are defined. User-defined fonts are actually defined as real numbers between 0 and 1, but MAKEFONT does a conversion internally to integer coordinates.) Normally, characters are drawn within a "unit square" that corresponds to the upper right quadrant of the design grid. If the strokes defining the character extend beyond this area, this may cause overlap between adjacent characters.

If very large characters are being edited, Control Dial 1 may be used to adjust the scale of the grid.

Notice the blinking box on the design grid. This marks the position of the last "move" or "draw" in the character definition.

If the tablet stylus is pressed within the character design grid, a stroke will be added to the character definition. The stroke will be either a "move" or a "draw", depending on the current state.

MOVE_TO

Selecting this menu item causes the current state to be set to "move". Selecting a position in the character design grid will then cause a "move" stroke to be added to the character. This is the default state upon entering EDIT mode.

DRAW_TO

Selecting this menu item causes the current state to be set to "draw". Selecting a position in the character design grid will then cause a "draw" stroke to be added to the character.

ORIGIN

This selection adds a stroke to the character which causes a move to the origin. This is useful since all characters in a standard font should have the last position at the origin. If this rule is not observed, the characters will be drawn with incorrect spacing (although this can be a feature of the font, not a problem).

ERASE

This function causes the last stroke to be erased from the character definition. This function may be selected multiple times to erase several strokes. There is no way to erase strokes except from the end of the list.

EXIT

Selecting EXIT updates the definition of the character being edited in the font and returns the user to the Main Control Menu.

RETURN

Selecting RETURN returns the user to the Main Control Menu without saving any changes that were made to the character being edited.

4. Downloading Standard Fonts

Files containing alternate fonts (BEGIN_FONT ... END_FONT structures) created by MAKEFONT may be sent to the PS 390 parser in the usual manner.

To define and download a standard font to the PS 390, the following steps should be performed:

1. Use MAKEFONT to write the font in standard font format. The file containing the standard font definition is named CHARFONT.DAT.

2. Convert this file to S-record format using MAKEFONT menu selection 2. By convention, the S-record file is called CHARFONT.SR.
3. Download the S-record format to the PS 390 floppy using the Diagnostic Utility Program TRANSFER routine. The file must be named CHARFONT.DAT on the floppy to ensure that the previous version is overwritten. (Note: You may want to make a copy of the floppy using the Utility Program COPYDISK before overwriting the font.)
4. Boot the PS 390 using the floppy with the new font.

5. Font Storage

MAKEFONT stores a font internally as an array of pointers to character definitions. A NIL pointer indicates that the associated character has not been defined.

Character definitions are records with two fields: an integer to keep track of how many strokes there are, and an array containing the strokes. Strokes are also records containing the absolute X and Y (integer) coordinates and a Boolean indicating whether it is a move or draw. The maximum number of strokes per character (the dimension of the stroke array) is 64, but as this is a symbolic constant it can be changed if needed.

The primary advantage in using this format for internal storage of the character definitions is the ease with which characters can be changed. For instance, swapping two characters involves only swapping the two pointers in the font array.

The font files are stored on the host as ASCII text. Each record of a font file consists of a 7-digit octal number. These numbers are decoded in various ways.

The first record in the file is an integer, giving the size of the "stroke table", in 16-bit words.

The second record in the file is an integer describing how many characters are in the font : either 128 or 256. (This number will be referred to as 'n').

The remaining records in the file comprise the "stroke table". The first 'n' of these records are integers which give the offset of the corresponding character definition in the stroke table. A zero value indicates that a character has not been defined.

Then there are a some zero records in the file, generally five. After these come the actual character definitions.

Suppose that, for example, the value in location 68 of the stroke table were 599. That would mean that the definition for character 67 (68–1) begins at location 599 in the stroke table. Then the value at location 599 would be the number of strokes (moves/draws) defining the character. If location 599 had a value of 10, then locations 600 to 609 would contain the move/draw instructions for the character.

The move/draw instructions are stored with the **X** and **Y** displacements at RELATIVE distances between –63 and 63. The information is packed into a 16-bit word as follows:

Bit 0	move/draw information (0=move, 1=draw)
Bits 1–7	Y-displacement
Bit 8	unused
Bits 9–15	X-displacement

Each character definition has to end with a zero word. Also, character definitions have to be aligned on longword boundaries. This means that words 1 to 'n' in the stroke table must all have ODD values, so that the first stroke definition command of each character has an EVEN offset. More zero words are added here and there in the stroke table to fill it out.

118. ASCII TO GSR CONVERTER

TT8. ASCII–TO–GSR CONVERTER

CONTENTS

Section TT8
ASCII-to-GSR Converter

The ASCII-to-GSR conversion program is a host-resident PS 390 option which allows a programmer to combine the advantages of ASCII programming with the faster data communication speeds possible through the PS 390 GSRs. Specifically, the conversion program converts a file of ASCII PS 390 commands into a procedure which the programmer can link into a host-resident program. When the procedure is executed, the commands are downloaded to the PS 390 via the GSRs. Contact your Evans & Sutherland Account Executive for distribution and installation details.

1. FILES

When you run the ASCII-to-GSR Converter, it prompts you regarding the name of the ASCII file to convert and for the format of the output file. There are several options which are described in Table 8-1.

Table 8-1. Output Options

Output Option	File Type	Notes
PASCAL	.pas	Generates a Pascal procedure with embedded calls to the VMS Pascal GSRs.
FORTRAN	.for	Generates a FORTRAN subroutine with embedded calls to the VMS FORTRAN GSRs.
C	.c	Generates a C function with embedded calls to the UNIX C GSRs.
LISP	.cl	Generates a Common LISP function with embedded GSR calls.
DATA	.data	Generates output in a format which can be read by the PS 390 Data Structure Editor.
ASCII	.asc	Generates a file containing ASCII PS 390 commands. This is primarily a debugging operation.

Two files are produced, and any errors are reported to your terminal. The first file is a listing file having the same name as the input file but with a

".lis" extension. This file contains a line-by-line listing of the input file, along with any error or informational messages. The ".lis" file is for debug purposes and may be deleted once the ASCII file has been converted successfully. The second file contains the output of the converter.

A sample ASCII input file and output files in Pascal, FORTRAN and C formats are included in the distribution.

2. Conventions

If you do not explicitly specify a file extension for the input file, the converter will use a .300 extension. There is no restriction on the size of an input file, but it is recommended that files be under 2000 lines. Long vector or polygon lists may cause a heap-exhausted fatal error. Extremely long procedures may also be refused by the VMS Pascal compiler. The input file should not contain muxing bytes as these may cause a fatal error.

The converter recognizes most command abbreviations such as CONN for CONNECT or VEC for VECTOR_LIST. If a valid abbreviation is not recognized, use the longer form of the same name.

The main program is responsible for issuing the GSR call for connecting to the PS 390. The sample programs included in the distribution (pdriver.pas, fdriver.for, and cdriver.c) illustrate how to set up a minimal program which connects the PS 390 and calls the generated procedure.

It is assumed that all error handlers have the standard names: ERROR_HANDLER for Pascal and ERR for FORTRAN. No error handler is required for C. Error handlers should be provided separately and linked in with the generated procedure. Refer to the sample programs for examples of how to do this.

It is also assumed that you have not changed the constant declarations that establish array sizes in the linking file "proconst.pas," residing in the GSRs library.

3. Error Reporting

The converter issues diagnostic messages which are self-explanatory. Note that it does not attempt to convert !RESET, units, or other commands which have no corresponding GSR, but it does issue an appropriate warning message. In a few cases, it may produce unexpected output which will need to be corrected. For example, because no direct equivalents exist in the GSRs library, INSTANCE OF commands with multiple descendants nested inside structures actually expand into several unnamed INSTANCE nodes. WITH_PATTERN...VECTOR_LIST is treated as an ordinary vector list.

Syntax errors in the input file are usually reported, and processing is continued. Unrecoverable errors indicate serious problems with the input file. Additional messages referring to garbage collection may also occasionally appear, indicating that the program is just recycling its heap storage.

4. Name Prefixing

The converter provides an optional feature which allows you to attach a prefix to all PS 390 names. The prefix is applied to all object labels and function labels. It is not applied to names of generic functions, such as F:ACCUMULATE; to names of initial function instances, such as TABLETIN; or to pick ID's.

To attach a name prefix, you need to create an initialization file in your main login directory, called "parser.ini" for VMS, or ".parserrc" for UNIX. This file should contain the following line:

```
(setq parser::*name-prefix* t)
```

The parser program then prompts you for a name prefix before processing the input file.

If you want to prefix names with some exceptions, you may specify those exceptions by adding another command to your initialization file which lists the names to protect. For example, if you want to protect the names "cube" and "knots," add the additional command:

```
(setq parser::*reserved-names* ´("cube" "knots"))
```

White space is ignored, so the list of names may extend to more than one line; but be sure that string quotes and parentheses are matched properly. For example, the following is equivalent to the command above:

```
(setq parser::*reserved-names*
   ´("cube"
     "knots"
   ))
```

TT9. TRANSFORMED DATA
AND WRITEBACK

TT9. TRANSFORMED DATA AND WRITEBACK

CONTENTS

Section TT9
Transformed Data and Writeback

The PS 390 provides a means to retrieve transformed data. Transformed data is the matrix or vector list representation of transformation operations in a display structure.

After an object has been transformed on the PS 390, the transformed accumulated data for that object can be retrieved from a given data node and then established as a separate data or operation node in the display structure. The transformed data can also be converted to an ASCII PS 390 command string for transmission to the host.

Transformed data can be obtained either as transformed vectors or as a transformation matrix which is the concatenation of transformations currently applied to the object.

If transformed vectors are requested, a data node can be created and an ASCII PS 390 VECTOR_LIST ITEMIZED command can be generated. If a transformation matrix is requested, an operation node can be created and an ASCII PS 390 MATRIX_4x4 command can be generated for transmission back to the host.

Once the node containing a transformed vector list or 4X4 matrix node is created, those nodes can be used as primitive data nodes or operation nodes, and connections can be made into the nodes just as for any other VECTOR_LIST ITEMIZED or 4x4_MATRIX node.

Transformations explicitly reserved for characters (CHARACTER ROTATE, etc.) are excluded from both forms of retrieved transformed data.

The Writeback feature allows displayed transformed vector data to be sent back to the host. The position of the writeback node in the display structure determines which transformations are applied to the writeback data. The system-generated writeback node includes all viewing and modeling transformations. Once the host has received the data, it can be used to generate hardcopy plots or display host-generated raster images. The user is responsible for retrieval and all subsequent processing of data on the host system.

1. Transformed Data Commands and Functions

To retrieve transformed data for a given data node (or set of data nodes):

- Mark the data node by applying a XFORM VECTOR or XFORM MATRIX node.
- Request the transformed data using an instance of F:XFORMDATA.
- Optionally, convert the transformed data to an ASCII PS 390 command string using an instance of F:LIST and send this ASCII information to the host computer via HOST_MESSAGE.

1.1 The XFORM Node

The XFORM node, a type of operation node, can be placed anywhere above the data node(s) for which transformed data are to be retrieved; however, the placement of the XFORM node with respect to other transformations is critical. The syntax of the command that establishes an XFORM node is:

```
Name := XFORM specifier APPLIED TO Node_Name;
```

where:

specifier is either VECTOR or MATRIX. To retrieve a transformed vector list, use VECTOR; to retrieve a transformation matrix, use MATRIX.

If XFORM VECTOR is used, all transformations applied to the data node(s) are taken into account, whether these transformations are above or below the XFORM VECTOR node.

If XFORM MATRIX is used, however, only those transformations above the XFORM MATRIX node are taken into account. To include all transformations applied to the data node(s), then, XFORM MATRIX should be placed immediately above the data node(s).

THEN may be substituted for APPLIED TO.

Node_Name is the node to be marked for transformed data retrieval. Admissible data nodes are vector lists and curves (rational polynomials, polynomials, and B-splines). Transformed data cannot be retrieved for characters and labels.

If **Node_Name** is an instance node covering two or more data nodes and if XFORM VECTOR is requested, then the transformed data for all nodes are consolidated into a single vector list.

NOTE

> The transformed counterparts of the original data nodes do not necessarily appear in the same order in which the INSTANCE command named those nodes. However, vector integrity is maintained within each node.

The transformed object(s) must be displayed when transformed data retrieval is requested; otherwise, the request has no effect.

If transformed vector information is requested (XFORM VECTOR), no more than 2,048 consecutive transformed vectors may be retrieved.

- TRANSLATE, SCALE, ROTATE, and MATRIX_3x3 transformations applied to data are taken into account when the transformed data are retrieved.
- Character transformations are not taken into account when the transformed data are retrieved. These include CHARACTER ROTATE, CHARACTER SIZE, TEXT SIZE, CHARACTER SCALE, and MATRIX_2x2.
- WINDOW, EYE BACK, FIELD_OF_VIEW, LOOK, MATRIX_4x3, and MATRIX_4x4 transformations applied to data are taken into account when transformed data are retrieved, but it is recommended that these six transformations be removed from the object definition beforehand.
- A VIEWPORT specification has no effect on the transformed data.

1.2 The F:XFORMDATA Function

Use an instance of F:XFORMDATA to request transformed data. F:XFORMDATA has five inputs and one output. (Discussion of inputs <4> and <5>, which specify a range of transformed vectors to be retrieved, is presented in section 4.)

- Input <1> is the active input for this function. Any message sent to this input will begin retrieval of transformed data if the other inputs have been prepared correctly.

- Input <2> is a constant input which accepts a string message containing the name of a XFORM node. Transformed data will be retrieved for the object(s) marked by this XFORM node.
- Input <3> is a constant input which accepts a string message containing the name of the new data or operation node to be created. The name also appears in the ASCII command string produced by F:LIST, if any.

  If XFORM VECTOR is used and if the name at input <3> is identical to the name of the original (untransformed) data node, the transformed data replace the original data in the display structure. (The immediate effect of this redefinition is to display the object with its transformations doubly applied—once explicitly in the display data structure and once implicitly in the transformed vector list).
- Output <1> contains the transformed data. If ASCII PS 390 command information is desired for the host, connect this output directly to F:LIST. Do not attempt to connect this output to anything else (such as another data node).

  Output <1> may remain unconnected if no ASCII transformed data are desired. (A data node can be created through XFORM VECTOR without any connections from this output.)

1.3 The F:LIST Function

F:LIST converts the output of F:XFORMDATA into an ASCII PS 390 command string suitable for storage on the host computer (and for retransmission back to the PS 390). There is no need to instance F:LIST unless this ASCII information is to be retrieved. F:LIST has one input and two outputs:

- Input <1> accepts the transformed data from output <1> of F:XFORMDATA.
- Output <1> contains the transformed data, reformatted as an ASCII PS 390 command string.

  If a transformed vector list was requested, a VECTOR_LIST ITEMIZED command is output. If a transformation matrix was requested, a MATRIX_4x4 command is output.

  The name of the command is the string that was on output <3> of F:XFORMDATA at the time of the request.

- Output <2> is a Boolean TRUE completion indicator. Refer to the transformed data sample program for an application of this completion indicator.

The ASCII command string from F:LIST may be sent to a host computer via HOST_MESSAGE.

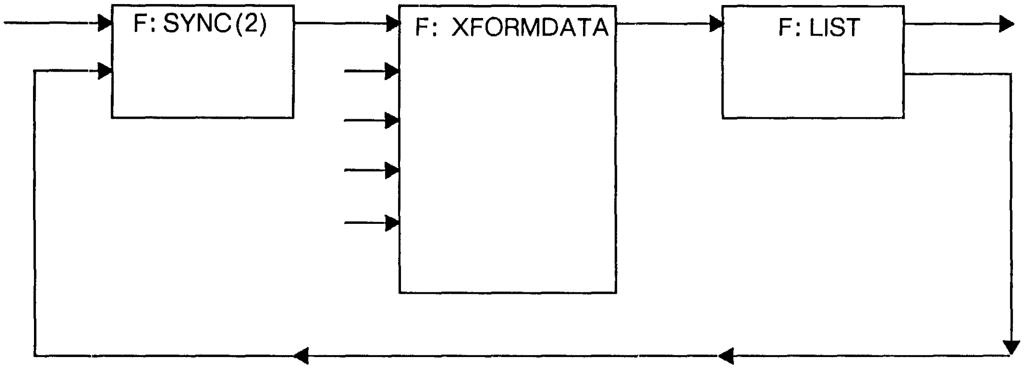
2. Excluding Certain Viewing Transformations

If WINDOW, EYE BACK, FIELD_OF_VIEW, LOOK, MATRIX_4x3, or MATRIX_4x4 transformations are applied to an object, transformed data may include inappropriate Z information. It is therefore recommended that these transformations be excluded from the object and replaced by a 4x4 identity matrix before transformed data are retrieved.

Since the default window transformation matrix is not an identity matrix, this practice is recommended even when no nodes for the above six transformations have been included explicitly in the display structure.

3. Using F:SYNC(n) to Prevent Overlapping Requests

After F:XFORMDATA is triggered, it begins supplying transformed data to F:LIST, which in turn converts the data to ASCII format. Before this process is finished, F:XFORMDATA could be triggered again, and F:XFORMDATA could supply new data before F:LIST is finished with the old. The result could be a nonsensical combination of the two requests. A suggested network to prevent overlapping transformed data requests is:



This network must be initialized before use by sending any message to input <2> of F:SYNC(2).

4. Specifying Vector Ranges for Transformed Data Retrieval

Inputs <4> and <5> of F:XFORMDATA restrict the retrieval of transformed vector data (XFORM VECTOR) to a specified range of vectors within the source vector list(s).

Input <4> (used only for VECTOR_LIST) is an integer index specifying the place in the vector list at which transformed vector retrieval is to begin. The default value is 1.

Input <5> (used only with VECTOR_LIST) specifies the number of consecutive transformed vectors to be retrieved. The default value is 2,048. No more than 2,048 consecutive vectors may be retrieved.

If inputs <4> and/or <5> are used for matrix data, they are ignored.

If the XFORM VECTOR node is applied to an instance node so that several data nodes are within the scope of the XFORM VECTOR node, transformed vectors can be retrieved from individual vector lists or portions of vector lists using the range specification. Vectors are numbered in sequence, beginning with the first vector list named in the INSTANCE command. For example, if the command sequence

```
XformIt := XFORM VECTOR APPLIED TO Z;
Z := INSTANCE OF A,B,C,D;
A := VECTOR_LIST N=5 ... ;
B := VECTOR_LIST N=6 ... ;
C := VECTOR_LIST N=10 ... ;
D := VECTOR_LIST N=8 ... ;
XformData := F:XFORMDATA;
```

has been entered, then transformed vectors for list C may be requested by using XFORMDATA inputs <4> and <5> as follows:

```
SEND FIX(12) TO <4>XformData;
SEND FIX(10) TO <5>XformData;
```

5. Transformed Data Sample Program

The following example illustrates both of the recommended features of a network for retrieving transformed data using the XFORM command: the exclusion of perspective and window transformations and the prevention of overlapping transformed data requests.

In this example, a conditional bit is used to switch between the perspective and window mappings (applied while designing the object) and the identity matrix (applied while sending the transformed object data). The untransformed object is Data; the transformed vector list to be created is Xdata.

```
Xform := BEGIN_STRUCTURE            { Set up switch mechanism }
       X :=  SET CONDITIONAL_BIT 1 ON;
             IF CONDITIONAL_BIT 1 IS ON THEN View;
             IF CONDITIONAL_BIT 1 IS OFF THEN Tran;
         END_STRUCTURE;

Tran := BEGIN_STRUCTURE             { To be used while getting }
                                    { transformed data }
             MATRIX_4x4 1,0,0,0 0,1,0,0 0,0,1,0 0,0,0,1;
             INSTANCE OF Obj;
        END_STRUCTURE;

View := BEGIN_STRUCTURE {To be used while viewing and designing}
                        {Viewing commands:  WINDOW, EYE BACK,  }
                        {FIELD_OF_VIEW, MATRIX_4x3, MATRIX_4x4, LOOK}
            INSTANCE OF Obj;
            END_STRUCTURE;

Obj := BEGIN_STRUCTURE { Set up transformed data request           }
                       { Transformation commands: ROTATE, TRANSLATE,}
                       { SCALE, and/or MATRIX_3x3                  }
             XFORM_REQUEST:= XFORM VECTOR;
             INSTANCE OF DATA;
       END_STRUCTURE;

XformData := F:XFORMDATA;   { Build transformed data network }
Sync2 := F:SYNC(2);
List := F:LIST;
CONNECT Sync2<1>:<1>XformData;
CONNECT XformData<1>:<1>List;
CONNECT List<1>:<1>HOST_MESSAGE;  { Send transformed data to host }
CONNECT List<2>:<2>Sync2;          { "Task completed" flag }
SEND <any message> TO <2>SYNC2;
SEND ´Obj.Xform_Request´ TO <2>XformData;
SEND ´Xdata´ TO <3>XformData;
DISPLAY Xform;
```

When the object has been designed and transformed properly and you are ready to send data to the host, the commands

```
SEND FALSE TO <1>Xform.X;
SEND <any message> TO <1>Sync2;
```

(or an equivalent function network) send the transformed data to the host. Since the perspective and window transformations are replaced by the identity matrix during this time, the displayed object becomes distorted or disappears during transmission. When the entire list has been sent, enter

```
SEND TRUE TO <1>Xform.X;
```

(or route the completion indicator of F:LIST to this input) to redisplay the object and continue designing).

6. Writeback Feature

The following sections describe how to use the Writeback feature. These sections contain:

- A description of the user interface for the Writeback feature. The user interface consists of the WRITEBACK operation node and the WRITEBACK initial function instance.
- Constraints on the use of the WRITEBACK operation node.
- Descriptions of the WRITEBACK function.
- A list of the commands that need to be interpreted by host-resident code to filter writeback data retrieved from the PS 390.
- An example of the sequence of data sent back to the host.
- An example of a host program that retrieves, processes, and files writeback data from the PS 390.

The Writeback feature is implemented by:

- Creating the WRITEBACK operation node (or using the system-generated writeback node, WB$).
- Activating the WRITEBACK operation node.
- Connecting the WRITEBACK function to a function network.

6.1 WRITEBACK Operation Node

When the PS 390 is booted, a WRITEBACK operation node is created. It is named WB$ and is placed above every user-defined display structure. This node can be triggered if an entire displayed picture is to be included in the writeback data. If writeback of only a portion of the picture is desired, the user must place other WRITEBACK nodes appropriately in the display structure.

A user-defined WRITEBACK operation node is created by the command:

```
Name := WRITEBACK [APPlied to Name1];
```

The WRITEBACK node has one input. A TRUE sent to input <1> of the WRITEBACK node triggers writeback for the display structure below the node. This trigger is sent by the user, for example:

```
SEND TRUE TO <1>name;
```

triggers that WRITEBACK node. Of course the node could be triggered through a function network using a function key, etc.

A WRITEBACK operation node delimits the display structure from which the writeback data will be collected. Only the data nodes below the WRITEBACK operation node in the display structure will be transformed, clipped, viewport scaled perspective divided (as delineated by the placement of the WRITEBACK node), and sent back to the host.

6.2 WRITEBACK Operation Node Constraints

Only a displayed structure can be enabled for writeback. This means that the WRITEBACK operation node must be traversed by the display processor and therefore must be included in the displayed portion of the structure. The default WRITEBACK node WB$ is displayed as part of every displayed structure. But, if the user creates another WRITEBACK node and if this node is triggered before being displayed, the following error message will result:

```
E 8   ACP cannot find your operate node
```

Any number of WRITEBACK nodes can be placed within a structure. However, only one WRITEBACK operation can occur at a time. If more than one node is triggered, the WRITEBACK operations are performed in the order in which the corresponding nodes were triggered.

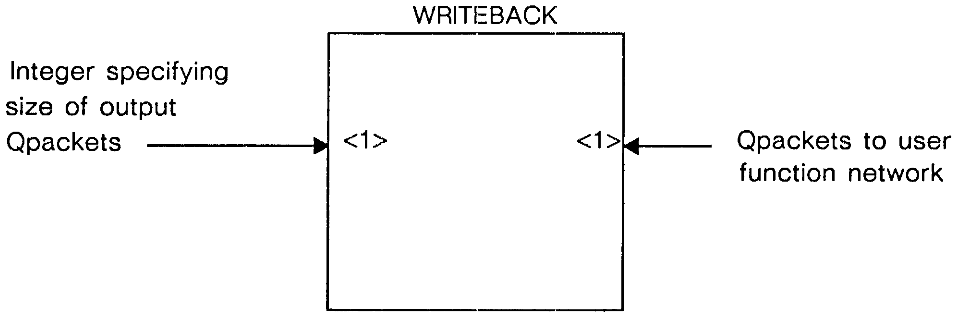
The terminal emulator and message_display information will not be returned to the host.

Before triggering the WRITEBACK operation, disable the SCREENSAVE function by entering the command

```
        SCREENSAVE:= nil;
```

7. WRITEBACK Function

An initial function instance, WRITEBACK, is created by the system at boot up.


WRITEBACK
Integer specifying size of output Qpackets
<1>
<1>
Qpackets to user function network

WRITEBACK sends encoded writeback data received from the display processor. The writeback data is prefixed by a start-of-writeback command, followed by the encoded data, followed by an end-of-writeback or end-of-frame command.

WRITEBACK has one user-accessible input queue. Input <1> accepts integers specifying the size of Qpackets to be output by the function. The default size is 512 bytes per Qpacket. The minimum and maximum size are 16 bytes per Qpacket and 1024 bytes per Qpacket, respectively. If the size specified by the user is not within this range, the default size will be used by the system.

The input value should be chosen such that the actual size of the Qpacket sent to the I/O port is less than or equal to the present input buffer size on the host computer.

If the CVT8TO6 function is used to send the binary data to the host, then the number of the bytes sent to the host is approximately 3/2 * the number of bytes sent by the WRITEBACK function.

For example, if the integer sent to <1> of the WRITEBACK function is 80, the largest Qpacket sent to the host will be 80 * 3/2 = 120. Qpackets, where the size is not a multiple of 4, will be padded to the next multiple of 4. For instance, Qpacket sizes of 77, 78, and 79, sent to CVT8TO6 will all have output sizes of 120.

WRITEBACK has one user-accessible output queue. Output <1> passes the encoded writeback data out as Qpackets until the end-of-writeback or end-of-frame command is seen.

This function is not activated by the normal input queue triggering mechanism. It is activated by sending a TRUE to any WRITEBACK operation node.

WRITEBACK will return all data below the WRITEBACK operation node. Host-resident code will be responsible for recognizing the start-of-writeback and end-of-writeback or end-of-frame commands.

Attribute information, such as color, must be interpreted by host code to ensure that the hardcopy plots are correct.

On the PS 390, viewport translations will not be applied to the data. Correct computation of the position of endpoints requires that the host program add a viewport center to each endpoint. The initial viewport center is established with a VIEWPORT CENTER command. The VIEWPORT CENTER command is sent following the start-of-writeback command. Any changes to the viewport center will be indicated through this sequence of commands: CLEAR DDA, CLEAR SAVE POINT, position endpoint, CLEAR SAVE POINT. The position endpoint becomes the new viewport center.

Also, on the PS 390, several commands such as ENABLE PICK and ENABLE BLINK are sent to the host. These will not typically be needed by the host program. However, these commands come directly from the refresh buffer and are not filtered by the PS 390. Host-resident code must filter the writeback data and strip out nonessential information.

7.1 Data Packets Returned

Data packets sent from the WRITEBACK function contain the following information:

- If bit 15 of the first word is 0, it signals that the data that follows is a command. For example, if the first word is H#0200 (Hex 0200) then the Line Generator status will follow.

bits	15	14	0
	0	command	
	parameter		

- If bit 15 of the first word is 1, it indicates that intensity, x and y coordinate information will follow. Intensity can range from 0 to 127. The format of the data is:

bits	15	14	13	12 -- 6	5 -- 0
	1	d	//	inten	////////
bits	15 - 13			12 --	0
	///////			y coord	
bits	15 - 13			12 --	0
	///////			x coord	

if d = 1, it is a DRAW
if d = 0, it is a MOVE

NOTE

In the illustrations of data format, the slash character is used to illustrate blocks of data that are unused.

7.2 Command Descriptions

The following list describes the commands that the host-resident code might have to interpret before it can recognize and filter writeback data received from the PS 390. These commands can be intermixed with vector data.

It is important to note that each command contains at least three 16-bit words. For example, if a command only has one parameter then the third word is unused, but it is still sent to the host. If a command has 3, 4, or 5 parameters, then 6 words will be sent for that command.

START-OF-WRITEBACK

code in hex = H#0B00
# 2816

Parameters:
Line texture (one word)
LGS (one word)

Marks the beginning of the writeback segment, of which there is guaranteed to be only one.

The texture and line generator status are included here. They follow the format shown below.

B00	
▨	Texture
LGS	

END-OF-WRITEBACK

code in hex = H#0C00
# 3072

Parameters:
None

Marks the end of the writeback segment.

C00	
0	0/1
▨	

0 = finished successfully, 1 = cannot finish operation because of insufficient memory.

LINE GENERATOR STATUS

code in hex = H#0200
# 512

Parameters:
Status word (one word)

Indicates dot mode (bit 8) and which display is selected (bits 0-3). Normally, only the dot mode bit must be referenced.

200
LGS
▨

Line Generator Status Register (LGS):

/// ///	/// ///	/// ///	/// ///	/// ///	/// ///	/// ///	SHO EPT	/// ///	/// ///	/////// ///////		SCOPE SELECT			
												D	C	B	A
15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00

Bit	Logical Names	
		B A
08	SHOWENDPT	Dot mode
03	BLANKD	Blank scope D (1 blanks the scope 0 enables the scope)
02	BLANKC	Blank scope C
01	BLANKB	Blank scope B
00	BLANKA	Blank scope A

COLOR

code in hex = H#0400
# 1024

Parameters:
Color value (one word)

400	
Hue	Saturation
/////////////	/////////////

/// ///	HI			HUE			LO	/// ////	HI	SAT		LO	/////////// ///////////		
15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00

TEXTURE

code in hex = H#0500
# 1280

Parameters:
Texture value (one word)

500	
/////////////	Texture
/////////////	/////////////

Line Generator Texture Register:

//////////////////////////////////// ////////////////////////////////////								TEXTURE BIT PATTERN							
15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00

H#007F or H#00FF both default to a Solid line.

CLEAR DDA

code in hex = H#0100
256

Parameters:
None

PICK BOUNDARY

code in hex = H#0300
# 768

Parameters:
Four Boundary Values (4 words)

CLEAR SAVE POINT

code in hex = H#0600
# 1536

Parameters:
None

SET PICK ID

code in hex = H#0700
# 1792

Parameters:
Pick ID Pointer (two words)

RESERVED

code in hex = H#0800
# 2048

ENABLE PICK

code in hex = H#0900
# 2304

Parameters:
None

DISABLE PICK

code in hex = H#0A00
# 2560

Parameters:
None

SET BLINK RATE

code in hex = H#0D00
# 3328

Parameters:
Blink Rate (one word)

ENABLE BLINK

code in hex = H#0E00
# 3584

Parameters:
None

DISABLE BLINK

code in hex = H#0F00
# 3840

Parameters:
None

END-OF-FRAME

code in hex = H#1700
# 5888

Parameters:
None

Signifies that the current update cycle is completed and that any following data is part of the next update frame. This also signifies end of the writeback segment.

VIEWPORT CENTER code in hex = H#1800

Parameters:
x center (one word)
y center (one word)
z center (one word)
spare (two words)

bits 15 0
coordinates 2's complement vector

This value has to be added to each x,y coordinate pair. This information is necessary to calculate the actual coordinates of the data which has been viewport scaled. Every time a new viewport is traversed by the ACP, a new viewport center command will be sent.

NOTE

Codes H#1900 - H#1F00 are reserved for future commands. Code H#0000 is defined as a no-op, and naturally has no parameters.

7.3 Sequence of Data Sent Back to the Host

The following example illustrates the sequence of data and the data in byte format sent to the host during a WRITEBACK operation.

B00	Start-of-writeback command
▨ Texture	
LGS	
400	Color command
Hue Saturation	
▨	
Intensity	VECTORS
Y	
X	
:	
:	
:	
:	
:	
:	
:	
:	
200	Line Generator Status command
LGS	
▨	
500	Texture command
▨ Texture	
▨	
400	Color command
Hue Saturation	
▨	
Intensity	VECTORS
Y	
X	
:	
:	
:	
:	
:	
:	
:	
:	
:	
C00	End-of-writeback command
0/1	0 = finished successfully, 1 = cannot
▨	finish because of insufficient memory

Data in Byte Format

```
0B  00    Start-of-writeback command
00  FF    Texture
04  70    LGS
04  00    Color command
80  00    Hue/Saturation
00  00    Not used
00  FF    Intensity
1Y  FF       Y
1X  FF       X
00  FF    Intensity
2Y  FF       Y
2X  FF       X
  :          :
  :          :
  :          :
02  00    LGS command
04  70    LGS
00  00    Not used
05  00    Texture command
00  FF    Texture
00  00    Not used
04  00    Color command
80  00    Color
00  00    Not used
00  FF    Intensity
1Y  FF       Y
1X  FF       X
  :          :
  :          :
  :          :
0C  00    End-of-writeback command
00  00    Finished successfully
00  00    Not used
```

8. Sample WRITEBACK Program

```
PROGRAM Writeback(Input,Output,Outfile,Devfile);

{ Program to read writeback data from a PS 390. This program sets up a  }
{ function network to get the writeback data and processes the data and }
{ creates a data file on the host with the data from the PS 390.}

CONST
 %INCLUDE ´PROCONST.PAS´
 Max_buf = 1024;

TYPE
 Int16 = -32768..32767;
 Max_line = VARYING [Max_buf] OF CHAR;
 %INCLUDE ´PROTYPES.PAS´

VAR
 OUTFILE : TEXT;
 DEVFILE : TEXT;
 DEVSPEC : P_VARYINGTYPE;
 OUTNAME : P_VARYINGTYPE;
 WBNAME  : P_VARYINGTYPE;
 COMMAND : INT16;
 INDEX : INTEGER;
 LEN : INTEGER;
 Inline : P_VARYBUFTYPE;
 vx,vy,vz : REAL;
 In_DDA : BOOLEAN := FALSE;

 %INCLUDE ´PROEXTRN.PAS´

  PROCEDURE ERR (ERROR: INTEGER);
  {}
  { ERROR HANDLER ROUTINE }
  {}
    BEGIN { ERR }
      {}
      WRITELN(´ ERROR :=´,ERROR);
      HALT;
      {}
    END; { ERR }
  PROCEDURE Setup;
  { Create function network to send writeback data to host }
  { This uses F:cvt8to6 to send 6-bit data to the host }
      BEGIN
      PFnInst('cvt','cvt8',Err);
      Pconnect ('Writeback',1,1,'cvt',Err);
      Pconnect ('cvt',1,1,'host_message', Err);
      PsndStr (CHR(36),2,'cvt',Err);
      PsndFix (48,1,'writeback', Err);
      PNameNil('screensave',Err);
      PPurge( Err);
      END;

{ Utility procedures }
  PROCEDURE Six_to_eight( Inbuf :  Max_line;
   VAR Outbuf : P_VARYBUFTYPE);
  { Data from PS 390 is in six-bit packed format. This procedure }
  { unpacks data }

  CONST Base = 36;

  TYPE
    Cheat_4 = PACKED RECORD CASE Boolean OF
   TRUE : ( i: UNSIGNED);
   FALSE : ( c: PACKED ARRAY [1..4] OF CHAR);
 END;

  VAR
    w : Cheat_4;
    c_out,cycle_count,buf_index,il,tc : INTEGER;
    first : BOOLEAN;

  BEGIN
    buf_index := 1;
    first := TRUE;
    cycle_count := 1;
    c_out := 4;
    outbuf := '';
    WHILE buf_index <= len DO
      BEGIN
 tc := ORD(Inbuf[buf_index]) - base;
 IF first THEN
   IF tc < 0 THEN
     c_out := 4+tc
   ELSE
     BEGIN
       first := FALSE;
       w.i := tc;
       cycle_count := SUCC(cycle_count);
     END { ELSE tc >= 0 }
```

```
ELSE
   BEGIN
     w.i := w.i * (2**6);
     w.i := UOR(w.i ,tc);
     cycle_count := SUCC(cycle_count);
   END; { ELSE }
 IF cycle_count > 6 THEN
   BEGIN
     FOR il := 4 DOWNTO (5-c_out) DO
       Outbuf := outbuf + w.c[il];
     cycle_count := 1;
     first := true;
   END;
 buf_index := SUCC(buf_index);
      END; { WHILE }
  END;

  PROCEDURE Next_Block;
  { Get a block of data from the PS 390 and convert from six to eight }
  { bit format }

  VAR Inbuff : Max_line;

  BEGIN
    PGETWAIT(Inbuff,err);
    Index := 1;
    Len := LENGTH(Inbuff);
    Six_to_eight ( Inbuff, Inline);
    Len := LENGTH(Inline);
  END;

  PROCEDURE Get_Value( VAR a : INT16);
  { Convert two bytes of input buffer to 16 bit integer }

  VAR i : INTEGER;

  BEGIN { Get_Value }
    a := 0;
    FOR i := 1 TO 2 DO
      BEGIN
 Index := Index + 1;
 IF Index > Len THEN
   Next_Block;
 a := a * 256 + ORD(Inline[Index]);
      END;
  END;{ Get_Value }
```

```
{ Procedures for processing refresh buffer commands }

  PROCEDURE Clear_DDA;
  {CLEAR DDA - %X0100 }
  {Parameters - None }
  {Indicates start of sequence to set viewport center }
  {This sequence is CLEAR DDA, CLEAR SAVE POINT, Vector, CLEAR SAVE POINT}

  VAR a,b : Int16;

  BEGIN
    In_DDA := TRUE;
    Get_value ( a );
    Get_value ( b );
    Writeln(Outfile,'{Clear DDA}');
  END;

  PROCEDURE Write_LGS;
  { WRITE LINE GENERATOR STATUS - %X0200 }
  { Parameters - Status word (one word)  }
  { Bit   8 : Dot mode. }
  { Bit   6 : Fast sweep ( Opposite of 7) }
  { Bits  5 -  4: Contrast selection (00-min,11-max) }
  { Bits  3 -  0: Scope select( 1 disables,0 enables) }

  VAR lgs,a : Int16;

  BEGIN
    Get_value ( lgs );
    Get_value ( a );
    Writeln(Outfile,'{Write LGS:',HEX(lgs),'}');
  END;

  PROCEDURE Write_Pick_Bound;
  { WRITE PICK BOUNDARY - %X0300 }
  { Parameters - Left, Right, Bottom, Top }

  VAR l,r,b,t,a : Int16;

  BEGIN
    Get_value ( l );
    Get_value ( r );
    Get_value ( b );
    Get_value ( t );
    Get_value ( a );
    Writeln(Outfile,'{Write_Pick_bound:',HEX(l),HEX(r),HEX(b),HEX(t),'}');
  END;
```

```
PROCEDURE Write_Color;
{ WRITE COLOR - %X0400 }
{ Parameters - Color value (one Word) }
{ Bit  15 : Not Used }
{ Bits 14 - 8 : Hue (High order in 14) }
{ Bit   7 : Not Used  }
{ Bits  6 - 3 : Sat (High order in 3) }
{ Bits  2 - 0 : Not Used }

VAR c,a : Int16;

BEGIN
  Get_value ( c );
  Get_value ( a );
  Writeln(Outfile,'{Write_Color:',HEX(c),'}');
END;

PROCEDURE Write_Texture;
{ WRITE TEXTURE - %X0500 }
{ Parameters - Texture value (one word) }
{ Bits 15 - 7 : Not Used }
{ Bits  6 - 0 : Texture bit pattern }

VAR t,a : Int16;

BEGIN
  Get_value ( t );
  Get_value ( a );
  Writeln(Outfile,'{Write_Texture:',HEX(t),'}');
END;

PROCEDURE Clear_Save_Point;
{ CLEAR SAVE POINT - %X0600 }
{ Parameters - None }

VAR a,b : Int16;

BEGIN
  Get_value ( a );
  Get_value ( b );
  Writeln(Outfile,'{Clear_Save_Point:}');
END;

PROCEDURE Set_Pick_Id;
{ SET PICK ID - %X0700 }
{ Parameters - Pick Id Pointer (two words) }

VAR a,b : Int16;
BEGIN
  Get_value ( a );
  Get_value ( b );
  Writeln(Outfile,'{Set_Pick_Id:',HEX(a),HEX(b),'}');
END;

PROCEDURE Set_Lightpen_Mode;
{ SET LIGHTPEN MODE - %X0800 }
{ **---------------------** }
{        PS 350 ONLY         }
{ **---------------------** }
{ Parameters - Control mask  }
{   Tracking cross y   }
{   Tracking cross x   }
{   Delta distance     }
{   Delta frames     }

VAR cm,x,y,dd,df : Int16;

BEGIN
  Get_value ( cm );
  Get_value ( x );
  Get_value ( y );
  Get_value ( dd );
  Get_value ( df );
  Writeln(Outfile,'{Set_Lightpen_mode:',HEX(cm),HEX(x),HEX(y),
    HEX(dd),HEX(df),'}');
END;

PROCEDURE Enable_Pick;
{ ENABLE PICK - %X0900}
{ Parameters - None }

VAR a,b : Int16;

BEGIN
  Get_value ( a );
  Get_value ( b );
  Writeln(Outfile,'{Enable_Pick:}');
END;

PROCEDURE Disable_Pick;
{ DISABLE PICK - %X0A00   }
{ Parameters - None    }

VAR a,b : Int16;

BEGIN
  Get_value ( a );
  Get_value ( b );
  Writeln(Outfile,'{Disable_Pick:}');
END;
```

```
PROCEDURE Enable_Writeback;
{ ENABLE WRITEBACK - %X0B00 }
{ Parameters - Line Texture }
{   Line Gen Status}

VAR a,b : Int16;

BEGIN
  Get_value ( a );
  Get_value ( b );
  Writeln(Outfile,´{Enable_Writeback:´,HEX(a),HEX(b),´}´);
END;

PROCEDURE Disable_Writeback;
{ DISABLE WRITEBACK - %X0C00 }
{ Parameters - None  }

VAR a,b : Int16;

BEGIN
  Get_value ( a );
  Get_value ( b );
  Writeln(Outfile,´{Disable_Writeback:}´);
END;

PROCEDURE Set_Blink_Rate;
{ SET BLINK RATE - %X0D00 }
{ Parameters - Blink rate }

VAR a,b : Int16;

BEGIN
  Get_value ( a );
  Get_value ( b );
  Writeln(Outfile,´{Set_Blink_Rate:´,HEX(a),´}´);
END;

PROCEDURE Enable_Blink;
{ ENABLE BLINK - %X0E00 }
{ Parameters - None  }

VAR a,b : Int16;

BEGIN
  Get_value ( a );
  Get_value ( b );
  Writeln(Outfile,´{Enable_Blink:}´);
END;
```

```
PROCEDURE Disable_Blink;
{ DISABLE BLINK - %X0F00 }
{ Parameters - None   }

VAR a,b : Int16;

BEGIN
  Get_value ( a );
  Get_value ( b );
  Writeln(Outfile,'{Disable_Blink:}');
END;

PROCEDURE End_Of_Frame;
{ END OF FRAME - %X1700    }
{ Parameters - None       }

VAR a,b : Int16;

BEGIN
  Get_value ( a );
  Get_value ( b );
  Writeln(Outfile,'{End_Of_Frame:}');
END;

PROCEDURE Viewport_Center;
{ VIEWPORT CENTER - %X1800}
{ Parameters - x center    }
{    y center    }
{    z center    }

VAR xc,yc,zc,a,b : Int16;

BEGIN
  Get_value ( xc );
  Get_value ( yc );
  Get_value ( zc );
  Get_value ( a );
  Get_value ( b );
  vx := xc;
  IF (vx >= 32768) THEN vx := vx - 65536.0;
  vx := vx/32767;
  vy := yc;
  IF (vy >= 32768) THEN vy := vy - 65536.0;
  vy := vy/32767;
  vz := zc;
  IF (vz >= 32768) THEN vz := vz - 65536.0;
  vz := vz/32767;
  Writeln(Outfile,'{Viewport_Center:',vx:6:6,' ',vy:6:6,' ',vz:6:6,'}');
END;
```

```
PROCEDURE Process_Vector;
{ Vector - Bit 15 of command = 1 }
{ Word 1 ( command )    }
{ Bit  15 : Always one for vector }
{ Bit  14 : 1 = Draw, 0 = Move }
{ Bits 12 - 6 : Intensity/2  }
{ Bits  5 - 0 : Not Used  }
{ Word 2 ( y coord)    }
{ Bits 15 - 13: Not Used  }
{ Bits 12 -  0: Y coordinate  }
{ Word 3 ( x coord)    }
{ Bits 15 - 13: Not Used  }
{ Bits 12 -  0: X coordinate  }

VAR a,b : Int16;
  un : UNSIGNED;
  pl : CHAR;
  int,x,y : REAL;

BEGIN
  Get_value ( a );
  Get_value ( b );
  un:=command;
  pl:=´l´;
  IF (UAND(un,%X4000) = 0) THEN pl := ´p´;
  un := UAND(un,%X1FC0);
  int := un;
  IF In_DDA THEN
    vz := int/8128.0
  ELSE
    int := (int/8128.0 + vz) * 2;
  un := a;
  un := UAND(un,%X1FFF);
  y := un;
  IF (y >= %X1000) THEN y := y - %X2000;
  IF In_DDA THEN
    vy := y / %XFFF
  ELSE
    y := y / %XFFF + vy;
  un := b;
  un := UAND(un,%X1FFF);
  x := un;
  IF (x >= %X1000) THEN x := x - %X2000;
  IF In_DDA THEN
    vx := x / %XFFF
  ELSE
    x := x / %XFFF + vx;
  IF In_DDA THEN
    BEGIN
  Writeln(Outfile,'{New View Center:',vx:6:6,' ',vy:6:6,' ',vz:6:6,'}');
  In_DDA := FALSE;
      END
    ELSE
    Writeln(Outfile,'{Vec ',pl,' ',x,',',y,' i=',int,'}');
  END;

  PROCEDURE Unknown;
  VAR a,b : Int16;

  BEGIN
    Get_value ( a );
    Get_value ( b );
    Writeln(Outfile,'{Unknown:',HEX(command),HEX(a),HEX(b),'}');
  END;

BEGIN  { Writeback }
  Write ('Enter Output File Name:');
  Readln(Outname);
  Write ('Enter Writeback Operate Node Name:{WB$ is default node}');
  Readln(wbname);
  open(Outfile,Outname,new);
  rewrite(Outfile);

  { Look for file specifying line for pattach procedure }
  { Example of record in PSDEV.DAT: }
  { 'logdevnam=tt:/Phydevtyp=async' }
  open(devfile,'psdev',old);
  reset(devfile);
  readln(devfile,devspec);
  close(devfile);

  PATTACH(devspec,err);  { Attach to PS 390 }
  Setup;   { Setup writeback network }
  PNAMENIL('SCREENSAVE', ERR);
  PPURGE(ERR);
  PSndBool(TRUE,1,wbname, Err); { Trigger write back operate }

  Next_block;   { Read in first block of writeback data }

  Index := 0;
  Command := 0;
  vx := 0.0;
  vy := 0.0;
  vz := 0.0;

  { Process writeback buffers until END OF FRAME or END WRITEBACK }
  WHILE (Command <> %X0C00) AND (Command <> %X1700) DO
```

```
    BEGIN
      Get_value(Command);
      IF (Command > 32767) THEN { If bit 15 of command if set }
 Process_vector
   ELSE
   CASE (Command DIV 256) OF
     %X01 : Clear_DDA;
     %X02 : Write_LGS;
     %X03 : Write_Pick_Bound;
     %X04 : Write_Color;
     %X05 : Write_Texture;
     %X06 : Clear_Save_Point;
     %X07 : Set_Pick_Id;
     %X08 : Set_Lightpen_Mode;
     %X09 : Enable_Pick;
     %X0A : Disable_Pick;
     %X0B : Enable_Writeback;
     %X0C : Disable_Writeback;
     %X0D : Set_Blink_Rate;
     %X0E : Enable_Blink;
     %X0F : Disable_Blink;
     %X17 : End_Of_Frame;
     %X18 : Viewport_Center;
     OTHERWISE Unknown;
   END; { CASE }
    END;
  PFNINST('SCREENSAVE', 'SCREENSAVE', ERR  PDETACH(ERR);
  PPURGE(ERR):
  {}
END. { Writeback }
```

9. Summary

Transformed data can be retrieved from a given data node and then established as a separate data or operation node in the display structure. The transformed data can also be converted to an ASCII PS 390 command string for transmission to the host. To retrieve transformed data you must:

- Mark the data node by applying a XFORM_VECTOR or XFORM_MATRIX node in the display structure. The syntax for the XFORM node command is:

  ```
  Name := XFORM specifier APPLIED_TO_Node_Name;
  ```

- Request the transformed data using an instance of the F:XFORMDATA function.

To send the transformed data to the host you can convert the data to an ASCII PS 390 command string with an instance of the F:LIST function and send the data to the host via HOST_MESSAGE.

Writeback allows displayed transformed data to be sent back to the host. The transformations applied to the writeback data are determined by the position of the Writeback node in the display structure.

A WRITEBACK operation node is created when the PS 390 is booted and placed above every user-defined display structure. This node can be triggered if an entire displayed picture is to be included in the writeback data. If writeback of only a portion of the picture is desired, the user must place other WRITEBACK nodes appropriately in the display structure. A user-defined WRITEBACK operation node is created by the command:

```
Name := WRITEBACK [APPlied to Name 1];
```

A WRITEBACK operation node delimits the display structure from which the writeback data will be collected.

TT10. CRASH DUMP FILE

TT10. CRASH DUMP FILE

CONTENTS

ILLUSTRATIONS

Section TT10
Crash Dump File

A crash dump file is written to the diskette in drive 1 when a system crash occurs. This file is always named **Crash.dat;1** and occupies only 1 block on the diskette. If the file already exists it will be overwritten by the new crash information. If the file doesn't exist, it will be created. If there is insufficient room on the disk for the file, no crash dump file will be written.

1. Crash Dump File

The file consists of the 8 Data, the 8 Address registers, system version, system type, program counter, error type, error number, 59 32-bit stack entries, and the 68000 status register. The following figure shows the structure of the data in the crash file. Section 2 gives more information on some of these values.

<table>
<tr><td colspan="2">DO</td></tr>
<tr><td colspan="2">D1</td></tr>
<tr><td colspan="2">D2</td></tr>
<tr><td colspan="2">D3</td></tr>
<tr><td colspan="2">D4</td></tr>
<tr><td colspan="2">D5</td></tr>
<tr><td colspan="2">D6</td></tr>
<tr><td colspan="2">D7</td></tr>
<tr><td colspan="2">A0</td></tr>
<tr><td colspan="2">A1</td></tr>
<tr><td colspan="2">A2</td></tr>
<tr><td colspan="2">A3</td></tr>
<tr><td colspan="2">A4</td></tr>
<tr><td colspan="2">A5</td></tr>
<tr><td colspan="2">A6</td></tr>
<tr><td colspan="2">A7</td></tr>
<tr><td colspan="2">Sysver</td></tr>
<tr><td colspan="2">Systype</td></tr>
<tr><td colspan="2">PC</td></tr>
<tr><td>Errtyp</td><td>Errnum</td></tr>
<tr><td colspan="2">●
Stack (236 Bytes)
●</td></tr>
<tr><td>Unused</td><td>SR</td></tr>
</table>

Figure 10-1. Data in Crash File

Section 3 gives an example of a host Pascal program that reads back the crash file from the PS 390. This information can be helpful in determining the cause of a crash.

This program uses constant input <2> of the READDISK Function to prevent the logging of crash files that were already recorded, by reading and then immediately deleting this file. If there is a true on input <2> after the file specified on input <1> is read, the file is deleted. The existence of a crash file would indicate that a crash had occurred since the last time the host program was run.

2. Crash Dump Information

There are three crash error types in the PS 390. Each type has a set of error numbers associated with the type. The three types are:

- System Errors - Type 1
- Traps - Type 2
- Exceptions - Type 3

The following is the list of errors for each type.

Type 1 – System Errors

1	Track number out of range
2	Disk drive not ready
3	Disk remains busy after a seek
4	Block number out of range
6	Lost data during read
7	Record not found during read
8	Data CRC error during read
9	ID CRC error during read
B	Lost data during write
C	Record not found during write
D	Data CRC error during write
E	ID CRC error during write
F	Write fault

10	Disk is write protected
11	Lost data during format
12	Write fault during format
14	Disk drive number out of range
15	Seek error
16	Drive not ready during read
17	Drive not ready during write
18	Disk not at track ø after restore command
19	Disk busy after restore command
1A	Track number out of range during format
1B	Drive not ready during format
1C	Disk write protected during format
1D	Time out during read
1E	Time out during write
1F	Time out during format
64	Wait maybe called with nil argument
65	Wait maybe called with a non–function
66	Wait maybe, already a function waiting
67	Wait maybe, parameter function waiting elsewhere
68	Q ship to an unrecognized Namedentity
69	Msgcopy, Message type shouldn't be copied
6A	Msgcopy, Msg type Has structure, unknown to Msgcopy
6B	Send, ´Me´ = nil
6C	Send, ´Me´ not a function instance
6D	Send, No such output port for this function
6E	Rem_conn/Add_conn, A1 = nil
6F	Add_conn, A2 = nil
70	Findqueue, Named item = nil
71	Findqueue, illegal queue number (queue no. < ø or queue no. > no. of inputs for function)
72	Allinpwait, Nmin > Nmax
73	Allinpwait, Nmin < 1

74	Tmessage, Waiting and n = Ø
75	Cmessage, Waiting and n = Ø
76	Lookmessage, Waiting and n = Ø
77	Allinputs, Nmin > Nmax
78	Allinputs, Nmin < 1
79	Fcnnotwait, Me = nil
7A	Findqueue, found a nil queue!
7B	Waitnextinput, n = Ø
7C	Anyoutputs, Me = nil
7D	Anyoutputs, illegal outset number
7E	Anyoutputs, no outset where there should be
7F	Fdispatch, function failed to re-queue after running
80	Text_text, B1 < Ø
81	Char_text, b < Ø
85	Error during disk read
8D	Initial structure not correct
8E	AnnounceUpdate List tail = nil;head < > nil
8F	FormatUpdate, update pointer non nil
90	FormatUpdate, Ready Head not nil but Tail is
91	Bad code file -- illegal Op
92	ByteIndex Invalid Acpdata type
93	FormatUpdate, PASCAL Head not nil but Tail is
94	Vec_size, Invalid Acpdata type
95	KillUpdate, Updfetch was < Ø
96	KillUpdate, update pointer non nil
97	Vec_bias, Invalid Acpdata type
99	CntCapacity, Invalid Acpdata type
9C	Unknown brand of Namedentity
9D	Hasstructure, has Qdatatype not found in Destroy
9E	Amuhead not a Qalphapair
A1	AppendVector, Invalid Acpdata type
A3	Nomemsched, Bad .Status for a fcn

A9	Bad update list on ACP time-out
AA	ACP Timeout during initialization
AB	Crashprepare, Name CRASH$ has not been defined
AC	DecUpdsync, C_header ^ .Updsync < ø
AD	FormatUpdate, Someone waiting in C_header ^ .Updswait already
AF	Someone else waiting in C_header ^ .Killer already
BO	Non-nil Qwait of a dying function
B3	Microcode won't fit into ACP
B4	Implementation limit on delta waits (2**31)
B8	Detected internal inconsistency
B9	Detected error (passed a bad parameter)
BA	Diskette's parsecode table inconsistent with parser
BD	Bad boundary on binary data xfer
BF	Default Devsts contains errors
C0	Inwait, f is already waiting or not a function
C1	Outwait, f is already waiting or not a function
C9	User generic function stack overflow
CA	Ug_run_cnt has become negative
CB	User generic function has bad alpha (on private queue)
CC	Bad format of MSGLIST .DAT detected
CD	MSGLIST (or code using it) has probably been corrupted
CF	Apparent datastructure incompatibility
D0	Bad MemOKindex detected
D1	Routine passed bad parm (e.g., a nil ptr)
D2	Lines to IBM system not active
D3	Floppy disk file INITGPIO.DAT; not found or unable to read
D4	Floppy disk file GPIOCODE.DAT; not found or unable to read
D5	Floppy disk file IBMFONT.DAT; not found or unable to read
D6	Floppy disk file IBMKEYBD.DAT; not found or unable to read
D7	Floppy disk file IBMASCII.DAT; not found or unable to read
D8	IBM GPIO timeout
D9	Number of minimum inputs is negative

DA	Number of maximum inputs < Number of minimum inputs
DB	Number of maximum inputs > Number of inputs for function
DC	Sendlist detected a bad list
DE	Sendmess: message to be sent is NIL
DF	Caller did not have a lock set already
E0	Curfcn in improper state to call Getinputs
E1	Cleanin, Curfcn in improper state to call Cleaninp (e.g., have you first called Getinputs?)
E2	Somebody remembered a forgotten non-fcninstance
E5	Alpha not already locked by caller
E6	Confusion in discarding bad message
E7	Lock not already set by caller
E9	RemOne, Curfcn does not have that many inputs
EA	RemOne, Message to be deleted and message pointed to by Curinputs is not the same
EB	Lock not already set in Gatheraupdate call
ED	Get2locks detected lock already set
EE	Error in semantic routine for polygon vertex
EF	Destination Alpha was not already locked
F0	Parent not already locked in add/remove from set
F1	Child not already locked in add to set
F3	Alpha not already locked in Gpseudoaupdate
F6	Confusion about locks or decausages
F7	Unknown tap reason
F8	Unanticipated state at which to see shoulder tap
F9	Illegal number of inputs
FC	No existing DCB found for this user
FD	Timeout, Message on input 1 disappeared before fcn could get it
FE	Error while initializing disk drive
FF	Error while reading disk header
100	Error while reading disk directory
101	THULE.DAT not found on disk
102	Error while reading THULE.DAT

103 Curfcn was not active at entry

104 Viewport not in structure

105 Real_simple, number of digits requested out of range (n < 1 or n > 9)

106 Getnextone, illegal queue specified

107 Getnextone, msg on head of queue and specified by Curinput do not agree

108 Getnextone, no message on queue, but Curinput < > NIL

109 ContBlock, nil block

10A Timeout when waiting for all on-line JCPs

10B Rehash only works first time, only time now.

10C No processor has right to issue this tap

10D GetVector, Not an Acpdata block

10E GetVector, Not a vector Acpdata block

10F Invalid qpacket received

110 Tolerance on FCnearzero is absurd

111 Set construct of father has no dummy control block

112 Function code has to be of type CI to have elements included and removed

113 ShadeEnviron node encountered in non PS 340

114 Unknown command received from Raster Backend; expected writeback.more or writeback.done.

115 Error in reading HMSCODE.DAT from disk.

116 Error in trying to get file info for HMSCODE.DAT.

117 Error in reading HMSVEC.DAT from disk.

118 Error in trying to get file info for HMSVEC.DAT.

119 Error in reading HMSCOL.DAT from disk.

120 Error in trying to get file info for HMSCOL.DAT.

121 Error in reading HMSCURS.DAT from disk.

122 Error in trying to get file info for HMSCURS.DAT.

123 Error in reading HMSFILT.DAT from disk.

124 Error in trying to get file info for HMSFILT.DAT.

125 No TurnOnDisplay (wrong CONFIG.DAT file for PS 390).

126 Can't follow alpha: TurnOnDisplay.

127 Raster Backend Timeout. The Raster Backend did not clear HMSmailbox[0] after it was sent an attention. A second attention to try to recover also received no response.

Type 2 - Traps

0 No mass memory on line, or too little to come up

1 More OKINTs than NOINTs or > 128 NOINTs

2 Free storage block size bad (on request or in free list)

3 Attempt to Activate a non-function (or nil) or bad software detected during startup (most commonly, incompatible datastru.sa detected but perhaps invalid startup routine sequencing (if someone has been mucking around with it))

4 NEW call failed to find memory, within NOMEMSCHED

5 Attempt to queue where a function is already waiting

6 Systemerror(n)

7 Badfcode(Fcn)

8 Mass Memory Error Interrupt

9 Utility Routine not included in this linked system

A Probable multiple DISPOSE of the same block

B Block exponent not big enough

C Attempt to divide with a divisor which is too small in FixLongDivide (twice the dividend must be less than the divisor)

D (Used by Motorola PASCAL)

Type 3 - Exceptions

0 Reset: Initial SSP

1 Reset: Initial PC

2 Bus Error (i.e. attempt to address nonexistent location in memory)

3 Address Error (i.e. attempt to access memory incorrectly, for example an instruction not starting on a word boundary).

4 Illegal instruction

5 Zero Divide

6 CHK Instruction

7 TRAPV Instruction

8	Privilege violation
9	Trace
10	Line 1010 Emulator
11	Line 1111 Emulator
24	Spurious interrupt

3. Crash Dump Program

Following is an example of a Pascal host program that writes the information from the diskette crash file into a host file.

```
PROGRAM CRASH (Input,Output,Outfile);

CONST
    %INCLUDE 'PROCONST.PAS/NOLIST'

TYPE
    %INCLUDE 'PROTYPES.PAS/NOLIST'

    cheat_4 = RECORD

         CASE Boolean OF
              TRUE : (i : Integer);
              FALSE : (c : Array[1..4] OF CHAR)

         END;

    cheat_2 = RECORD
         CASE Boolean OF
              TRUE : (i : [WORD] 0..1024);
              FALSE : (c : Array[1..2] OF CHAR)

         END;

    Buffer = RECORD
         CASE Boolean OF
              TRUE : (b : P_VaryBuftype);
              FALSE : ({ Length of P_VaryBuftype is in Dummy}
                   Dummy        : [WORD] 0..1024;
                   Dreg         : Array[0..7] of Cheat_4;
                   Areg         : Array[0..7] of Cheat_4;
                   SVer         : Cheat_4;
                   Stype        : Cheat_4;
```

```
                 PC          : Cheat_4;
                 Errtyp      : Cheat_2;
                 Errnum      : Cheat_2;
                 Stack       : Array[1..59] of Cheat_4;
                 Not_Used    : Cheat_2;
                 SR          : Cheat_2)

        END;

VAR
     Devtyp         : Integer;
     Inbuff         : P_VaryBuftype;
     OutBuff        : Buffer;
     Found          : BOOLEAN;
     Outfile        : text;

%INCLUDE 'PROEXTRN.PAS/NOLIST'

%INCLUDE 'VAXERRHAN.PAS/NOLIST'

PROCEDURE Init_ps300;

    {

    FUNCTIONAL DESCRIPTION:

    Initialize the comm link to the PS 390

    }

    VAR
        a, Modify : P_Varyingtype;

    BEGIN
    Write('Enter Type of Interface (1=Async, 2=Ethernet, 3=Parallel):');
    Readln( Devtyp );
    Write('Enter Device name :');
    Readln( a ); CASE Devtyp OF
         1 :
              Modify := 'LOGDEVNAM=' + a + '/PHYDEVTYP=ASYNC';
         3 :
              Modify := 'LOGDEVNAM=' + a + '/PHYDEVTYP=ETHERNET';
         2 :
              Modify := 'LOGDEVNAM=' + a + '/PHYDEVTYP=PARALLEL'
         OTHERWISE
         END;
    PAttach( Modify, PI_Error_handler)
    END;
```

```
PROCEDURE Trigger_read;

    {

    FUNCTIONAL DESCRIPTION:

    Create instance of function network to retrieve CRASH.DAT
    file from disk. The network will convert the data block
    to six-bit format and break it into packets of 72 bytes
    which will be put on host_message.

    }

    VAR

         a : CHAR;

PROCEDURE BREAKUP;
{ Code generated by Network Editor 1.08 }
{ This function network takes an incoming qpacket and breaks it }
{ into smaller packets to be sent over an terminal line since }
{ most terminal handlers have some limit to the input length }
{ BREAKUP }
BEGIN
{ Frame1: }
    PFnInstN (´Break_sync´, ´SYNC´, 2, PI_Error_handler);
    PFnInst (´Break_route´, ´BROUTEC´, PI_Error_handler);
    PFnInst (´Add_constant´, ´CONSTANT´, PI_Error_handler);
    PFnInst (´Break_add´, ´ADDC´, PI_Error_handler);
    PFnInst (´Breakup´, ´TAKE_STRING´, PI_Error_handler);
    PFnInst (´In_length´, ´LENGTH_STRING´, PI_Error_handler);
    PFnInst (´Len_compare´, ´GTC´, PI_Error_handler);
    PFnInst (´Route_string´, ´BROUTE´, PI_Error_handler);
    PFnInst (´Route_start´, ´BROUTE´, PI_Error_handler);
    PFnInst (´cvt´, ´CVT8TO6´, PI_Error_handler);
    PFnInst (´rd´, ´READDISK´, PI_Error_handler);
    PFnInst (´prnt´, ´PRINT´, PI_Error_handler);
    PFnInst (´Breakup_in3´, ´CONSTANT´, PI_Error_handler);
    PConnect (´Break_sync´, 1, 1, ´Breakup´, PI_Error_handler);
    PConnect (´Break_sync´, 1, 2, ´Break_route´, PI_Error_handler);
    PConnect (´Break_sync´, 2, 2, ´Breakup´, PI_Error_handler);
    PConnect (´Break_sync´, 2, 2, ´Break_sync´, PI_Error_handler);
    PConnect (´Break_sync´, 2, 2, ´Break_add´, PI_Error_handler);
    PConnect (´Break_route´, 1, 1, ´Add_constant´, PI_Error_handler);
    PConnect (´Break_route´, 1, 2, ´Route_string´, PI_Error_handler);
    PConnect (´Add_constant´, 1, 1, ´Break_add´, PI_Error_handler);
    PConnect (´Break_add´, 1, 2, ´Break_add´, PI_Error_handler);
    PConnect (´Break_add´, 1, 2, ´Route_start´, PI_Error_handler);
    PConnect ('Break_add', 1, 1, 'Len_compare', PI_Error_handler);
    PConnect ('Breakup', 1, 1, 'cvt', PI_Error_handler);
    PConnect ('Breakup', 2, 1, 'Break_route', PI_Error_handler);
    PConnect ('Breakup', 2, 1, 'Breakup_in3', PI_Error_handler);
    PConnect ('In_length', 1, 2, 'Len_compare', PI_Error_handler);
    PConnect ('Len_compare', 1, 1, 'Route_string', PI_Error_handler);
    PConnect ('Len_compare', 1, 1, 'Route_start', PI_Error_handler);
    PConnect ('Route_string', 2, 1, 'Breakup', PI_Error_handler);
    PConnect ('Route_start', 2, 2, 'Breakup', PI_Error_handler);
    PConnect ('cvt', 1, 1, 'host_message', PI_Error_handler);
    PConnect ('rd', 1, 1, 'Break_sync', PI_Error_handler);
    PConnect ('rd', 1, 1, 'In_length', PI_Error_handler);
    PConnect ('rd', 2, 1, 'prnt', PI_Error_handler);
    PConnect ('prnt', 1, 1, 'host_message', PI_Error_handler);
    PConnect ('Breakup_in3', 1, 3, 'Breakup', PI_Error_handler);
    PSndStr(CHR(36), 2, 'cvt', PI_Error_handler);
    PSndFix (48, 3, 'Breakup',PI_Error_handler);
    PSndFix (48, 2, 'Breakup_in3',PI_Error_handler);
    PSndFix (48, 2, 'Add_constant',PI_Error_handler);
    PSndFix (1, 2, 'Break_sync',PI_Error_handler);
    PPutPars('Set priority of prnt to 9; ',PI_Error_handler);
END;

BEGIN
IF Devtyp = 1
THEN
    Breakup
ELSE
    BEGIN
    PFnInst ('rd', 'READDISK', PI_Error_handler);
    PFnInst ('prnt', 'PRINT', PI_Error_handler);
    PConnect ('rd', 2, 1, 'prnt', PI_Error_handler);
    PConnect ('prnt', 1, 1, 'host_message', PI_Error_handler);
    PConnect ('rd', 1, 1, 'host_message', PI_Error_handler);
    PPutPars('Set priority of prnt to 9; ',PI_Error_handler);
    END;

Write(' Do you want to delete CRASH.DAT after reading?');
Readln( a );
IF (a = 'Y') OR (a = 'y')
THEN
    Psndbool( TRUE, 2, 'rd', PI_Error_handler)
ELSE
    Psndbool( FALSE, 2, 'rd', PI_Error_handler);
Psndstr( 'CRASH', 1, 'rd', PI_Error_handler)
PPurge( PI_Error_handler );
END;
```