PS 300 DOCUMENT SET

VOLUME 2b

GRAPHICS PROGRAMMING

Many concepts in this volume are proprietary to Evans & Sutherland, and are protected as trade secrets or covered by U.S. and foreign patents or patents pending.

Evans & Sutherland assumes no responsibility for errors or inaccuracies in this document. It contains the most complete and accurate information available at the time of publication, and is subject to change without notice.

EVANS & SUTHERLAND COMPUTER CORPORATION
P.O. Box 8700, 580 Arapeen Drive
Salt Lake City, Utah 84121

PS 300 TUTORIAL MODULES

ADVANCED CONCEPTS

This volume consists of six tutorial modules which detail more advanced concepts of PS 300 graphics programming. Because it builds on fundamental information detailed in Volume 2A, you should read Volume 2A first.

Each tutorial module covers a PS 300 programming concept or group of related concepts. Because each module details a separate, advanced skill, the modules may be read in any order desired.

Note that this volume contains the module USING THE PS 340. This information is specific to users of the PS 340; it need not be read by other users.

The following provides a capsule description of each module:

CONDITIONAL REFERENCING describes how detail can be added to or deleted from a view on the screen.

FUNCTION NETWORKS II describes more advanced ways to use function networks (refer to FUNCTION NETWORKS I in Volume 2A for fundamental uses of function networks). This includes multiple uses of dials (via function keys), labeling dial LEDs, limiting a model's motion, and storing/retrieving variables.

TEXT MODELING details how to create character strings, how to use commands and functions to manipulate character strings, and how to create and use different character fonts.

PICKING describes how to use the data tablet to activate a given action by picking an object being displayed.

TRANSFORMED DATA is the vector list or matrix representation of transformations which have been applied to an object. This module details how to retrieve transformed data so that it can be manipulated as a separate entity in the model's display tree or retrieved by the host computer.

USING THE PS 340 describes how to define polygonal objects, including how to perform rendering operations for both vector and raster displays.

In addition to the tutorial modules, this volume contains reference material, including sample programs illustrating various PS 300 programming techniques, and a glossary of terminology specific to the PS 300.

The appendix contains a paper by Dr. Alan L. Davis. The PS 300 function network facility bears a striking resemblance to data-flow concepts and theory that have been the subject of research for numerous years. Dr. Davis was contracted to write a self-contained tutorial discussion of data-driven programming in general and PS 300 function network programming in particular. It is hoped that this paper will assist the PS 300 user in writing well-formed function network programs which are efficient, easy to test and debug, and easy to modify.

CONDITIONAL REFERENCING

SELECTING PORTIONS OF A MODEL FOR DISPLAY

CONTENTS

CONDITIONAL REFERENCING

ILLUSTRATIONS

This module introduces and explains ***Conditional Referencing***--ways to display selected branches of a display tree without displaying other branches.

Conditional referencing is useful, for example, if you have a model of an assembly that you would like to add parts to or take parts from, showing various stages of development or assembly.

There may be layers of detail in your model that you would like to be able to overlay or strip off. An example of adding detail might start with an outline map of the United States, then sequentially add major rivers, mountain ranges, state borders, major cities, county borders, etc.

You might also want to display different views of an object at different times to animate an object, or alternately display and blank an object at a selectable rate (blinking).

These kinds of operations are achieved with conditional referencing, using three methods: conditional-bit settings, level-of-detail settings, and rate settings.

To use conditional referencing, a minimum of two nodes must be placed in a display tree. The first node (called a SET node) sets a condition:

THE CONDITION IS 1

The second node (called an IF node) tests the condition and makes the traversal of the branch (and therefore the display of data indicated by that branch) dependent on the condition set in the first type of node:

IF THE CONDITION IS 1 THEN DISPLAY Object1

IF THE CONDITION IS 2 THEN DISPLAY Object2

Figure 1 shows these nodes in a display tree. These nodes are attribute nodes and follow the same rules of placement and of use as operate nodes.

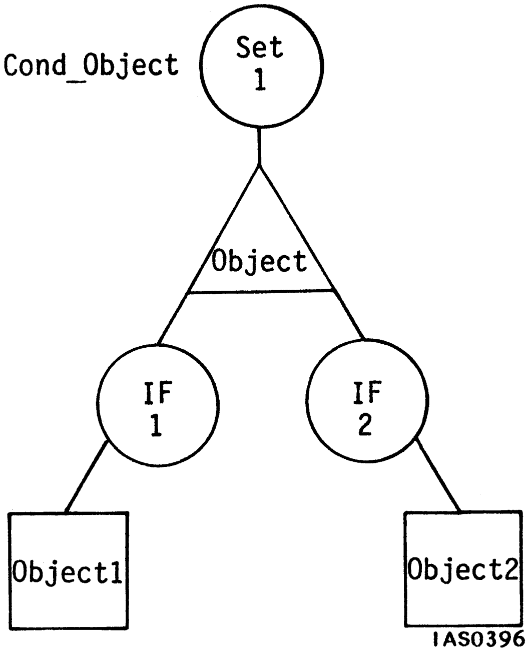


Figure 1. Display Tree Including Conditional Referencing Nodes

In the above example, displaying the SET node (Cond_Object) will result in Objectl being displayed and Object2 not being displayed. This is because the condition is not satisfied for the branch with Object2. By changing the condition from l to 2 in the SET node, Object2 will be displayed and Objectl will not be displayed.

The values in both the SET node (Cond_Object) and the IF nodes (Objectl, Object2) can be changed interactively. For example, the two branches could be alternately displayed by toggling the numbers in the SET node between l and 2.

The SET and IF nodes and the commands to create them are explained in subsequent sections.

OBJECTIVES

In this module, you will learn to display selected parts of your display tree using:

- *Conditional-bit* attribute settings
- *Level-of-detail* attribute settings
- *Rate* attribute settings

PREREQUISITES

Before reading this module, you should be familiar with the rules for using operation nodes in display structures ("Modeling" module) and the differences between matrix operations and attribute operations ("Graphics Principles"). This module uses the Robot example created in the "Modeling" and "PS 300 Command Language" modules.

USING CONDITIONAL-BIT ATTRIBUTE SETTINGS

Conditional bits are used to display selected branches of a display tree, independent of whether other branches are displayed. Branches of a display tree that have IF nodes that are not satisfied by the condition are not traversed by the display processor and are therefore excluded from displayed data.

The SET CONDITIONAL_BIT node is used to set any of 15 conditional bits (0-14). By placing the SET CONDITIONAL BIT node above an instance node, bit settings affect all branches under the instance node.

The SET node is created with the SET CONDITIONAL_BIT command. The syntax is as follows:

Name := SET CONDITIONAL_BIT ***n switch*** APPLIED TO ***Name1***;

where:

n is an integer from 0 to 14, corresponding to the conditional bit to be set ON or OFF.

switch is either ON or OFF.

Name1 is the descendent node of the conditional bit node.

all bits default to OFF.

For example, the following command creates a SET node and sets BIT 2 ON applied to Car.

```
Pattern := SET CONDITIONAL_BIT 2 ON THEN Car;
Car := INSTANCE OF Body, Wheels;
```

When you create a SET node, you explicitly set one bit on or off. However, all 14 bits default to off. So if you enter the command:

```
Name := SET CONDITIONAL_BIT 1 ON APPLIED TO Name1;
```

then bit 1 is on, and bits 2-14 are off. All bits can be changed by sending values to an input of the SET node.

Inputs to the SET CONDITIONAL_BIT node are as follows:

Boolean-------------> <1> Sets the original bit (n) set by the command to be ON (T) or OFF (F).

Integer-------------> <2> Sets bit number input (0-14) ON.

Integer-------------> <3> Sets bit number input (0-14) OFF.

Integer-------------> <4> Disables bit number input (0-14) from being affected by this node.

Integer-------------> <5> Toggles bit number input (0-14).

The SET node controls the states of the conditional bits and it is only through the set node that the conditions of all 15 bits are changed. If bit 5 was originally set to ON and then you want to set it to OFF, it could be done in any of the following three ways:

- Sending the integer 5 to input<3> of the SET node.
- Sending a false to input<1> of the SET node.
- Sending the integer 5 to input<5> of the SET node.

Of course, the SET node is useless unless you have an IF node that tests the condition set by the SET node. The IF node tells under which condition a branch will be traversed for display.

IF nodes are created with the IF CONDITIONAL_BIT command. The syntax is as follows:

Name := IF CONDITIONAL_BIT ***n switch*** APPLIED TO ***Name1***;

where:

n is an integer from 0 to 14, indicating which bit to test.

switch is the setting to be tested, ON or OFF.

name1 is the descendent of the IF node.

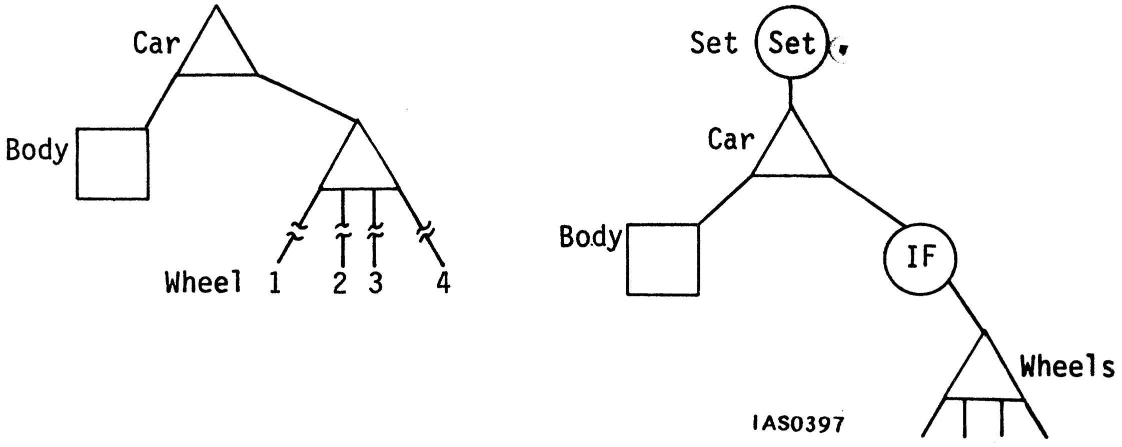
The IF CONDITIONAL_BIT node has one input that accepts an integer (0-14) to change the bit number in the node.

In the following command sequence, when Car is displayed Wheels would also be displayed.

```
Set :=   SET CONDITIONAL_BIT 4 ON APPLIED TO Car;
PREFIX Wheels WITH IF BIT 4 IS ON;
```

If bit 4 Car is set to Off or the condition in Wheels is changed to Off, then the test in Wheels would fail and Wheels would not be displayed.

The display tree for Car that this command sequence creates in shown in Figure 2.



(Original Display Tree)

(After Conditional Referencing)

Figure 2. Car Display Trees

Figure 3 is a display tree for a molecule for which conditional referencing will be implemented.

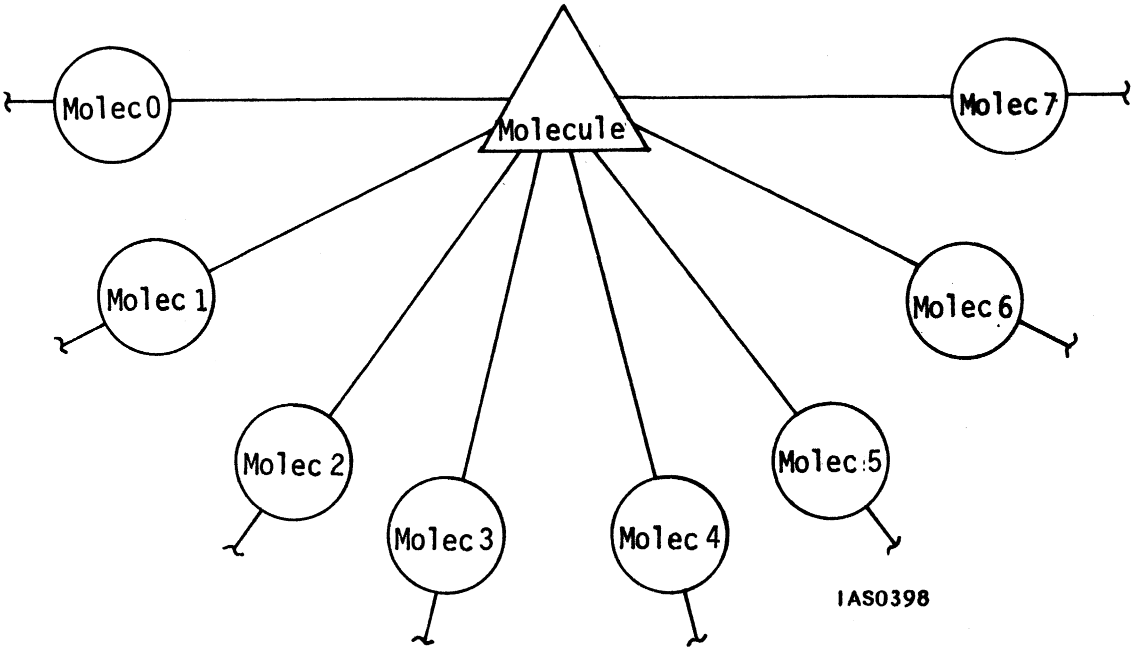


Figure 3. Molecule Display Tree

In Figure 3 notice that the Molecule is made up of an instance node pointing to 8 SET COLOR nodes for parts of the molecule. The eight parts can be controlled separately for display by placing a SET node and eight IF nodes in the structure.

The molecule will be set with the following conditions.

Bit No.	Condition	Result
1	Off	Branch 1 (Molec1_Color) will be displayed
2	Off	Branch 2 (Molec2_Color) will be displayed
3	Off	Branch 3 (Molec3_Color) will be displayed
4	Off	Branch 4 (Molec4_Color) will be displayed
5	Off	Branch 5 (Molec5_Color) will be displayed
6	Off	Branch 6 (Molec6_Color) will be displayed
7	Off	Branch 7 (Molec7_Color) will be displayed
8	Off	Branch 8 (Molec8_Color) will be displayed

The display tree to implement this is shown in Figure 4.

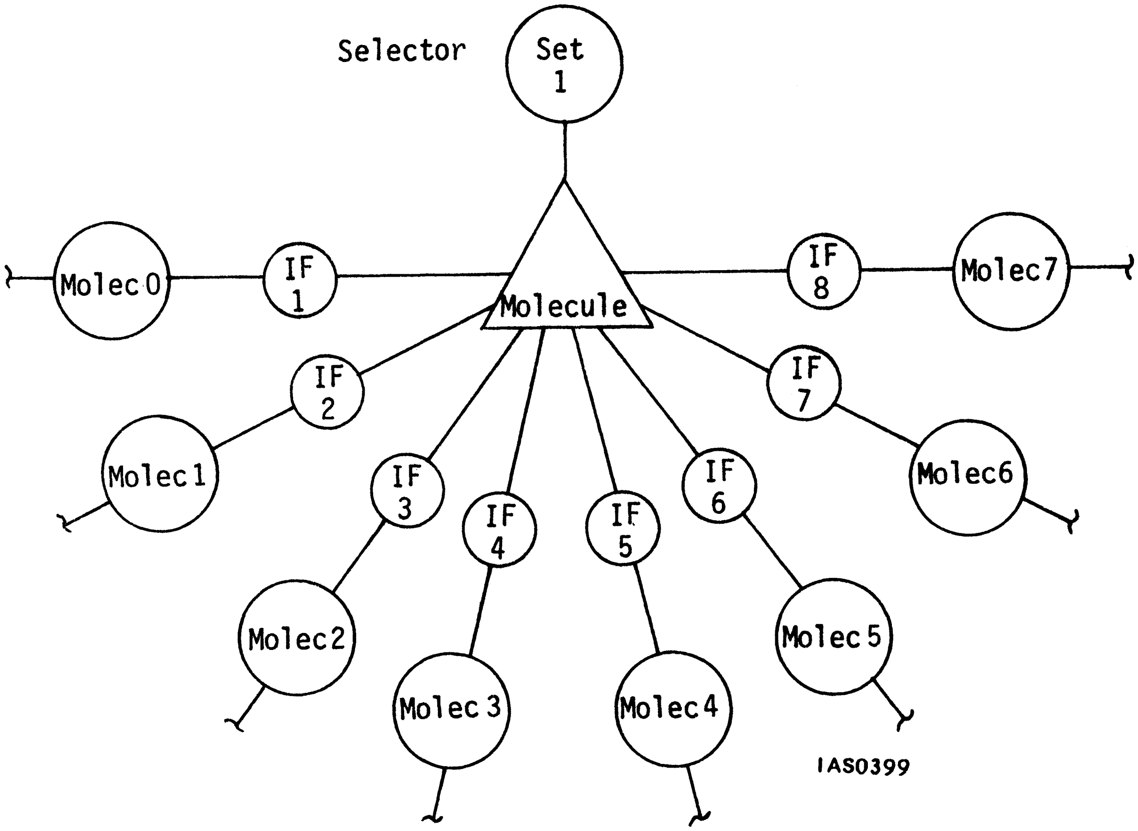


Figure 4. Display Tree For Conditional Referencing in Molecule

Exercise

Add conditional-bit referencing to the display tree for Molecule. The first step is to place a SET node above the instance node Molecule. Do this by entering:

```
Selector := SET CONDITIONAL_BIT 1 Off THEN Molecule;
```

Remember, even though the command says to set only conditional bit 1 off, this one node may be used to separately control the on/off condition of all 15 conditional bits. Also, note that the condition of the other 14 bits defaults to off.

Next place nodes at the top of each branch under the instance node so that the branches will be separately selectable for display. To do this, redefine Molecule as follows:

```
Molecule := BEGIN_STRUCTURE

                    IF BIT 1 IS OFF THEN Molec0_Color;
                    IF BIT 2 IS OFF THEN Molec1_Color;
                    IF BIT 3 IS OFF THEN Molec2_Color;
                    IF BIT 4 IS OFF THEN Molec3_Color;
                    IF BIT 5 IS OFF THEN Molec4_Color;
                    IF BIT 6 IS OFF THEN Molec5_Color;
                    IF BIT 7 IS OFF THEN Molec6_Color;
                    IF BIT 8 IS OFF THEN Molec7_Color;

             END_STRUCTURE;
```

You have built the display tree that allows conditional-bit referencing in Molecule. Notice that the molecule is displayed because all conditional bits are set off. To remove parts of the molecule from display, bits must be set on.

To control the on/off condition of the eight bits that affect the branches of this display tree, a function network can be used to connect the function keys to the SET node named Selector. That network is shown in Figure 5.

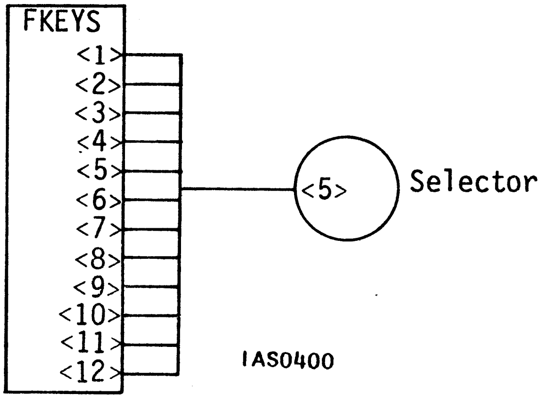


Figure 5. Function Network for Conditional Bit Control

FKEYS will output integers corresponding to the number of the pressed function key. Input<5> to the SET CONDITIONAL_BIT node toggles the setting of the bit corresponding to the integer received. For example, if bit 6 is off, pressing Function Key 6 will turn bit 6 on.

Enter the following commands to build the network.

```
        CONNECT FKEYS<1>:<5>Selector;
```

The display tree is now designed to allow conditional display of parts of the molecule (Molec0 through Molec7). Also, the function keys have been connected to control this display.

One step remains in this particular case. The values used to define the molecule are large. The molecule has a diameter of some 45,000 units. To see the molecule, put a window around it and disable depth cueing by entering:

```
    Molecule_View :=  WINDOW
                      X=-22500:22500
                      Y=-22500:22500
                      FRONT BOUNDARY =-22500
                      BACK BOUNDARY = 22500 APPLIED TO Intensity;
        Intensity :=  SET INTENSITY ON 1:1 APPLIED TO Selector;
```

now,

```
    DISPLAY Molecule_View;
```

Press SHIFT/LINE LOCAL to activate the function keys. Use keys F1 through F8 to toggle the display of the parts of the molecule.

When you are finished enter:

```
    REMOVE Molecule_View;
```

USING LEVEL_OF_DETAIL CONDITIONAL REFERENCING

The conditional-bit method shown for the molecule is usually used when you need to separately control the display of branches of your display tree in a variety of sequences. In the level-of-detail method, the parts of a model are always displayed and removed in a predetermined sequence.

Level-of-detail is usually used to overlay detail on your picture. For example, progressive detail could be added to an outline of a sphere (world) to add continents, mountain ranges, states, etc.

Level-of-detail can also be used to run animation sequences comprised of a series of separate picture definitions.

Unlike conditional-bit referencing where 15 variables (bits) are set, only one variable is set using the level-of-detail method. All IF nodes are tested against that one variable in the SET node.

The command to create a SET LEVEL_OF_DETAIL node is as follows.

Name := SET LEVEL_OF_DETAIL TO ***n*** APPLIED TO ***Name1***;

where:

n is an integer from 0 to 32767 indicating the level of detail value.

Name1 is the descendent of the SET node.

the default level of detail (n) is 0.

Inputs for updating the SET LEVEL_OF_DETAIL node are as follows:

Integer-------------->	<1>	Changes the level of detail (0-32767) to the value of the received integer.

Determining the Order for Overlaying Detail

Because level-of-detail controls the display of branches in a determined order, the conditional statements are expressed as relationships rather that the two-state (on/off) type used in conditional-bit references.

These relationships are:

Less Than	<
Less Than Or Equal To	<=
Equal To	=
Not Equal To	<>
Greater Than Or Equal To	>=
Greater Than	>

and are specified in the IF LEVEL_OF_DETAIL node. The command to create this IF node is as follows:

Name := IF LEVEL_OF_DETAIL *relationship n* THEN ***Namel***;

where

relationship is the relationship to n to be tested (<, <=, =, <>, >=, >).

n is an integer from 0 to 32767 indicating the number (along with the previous relationship) to compare against the current level of detail setting.

namel is the descendent of the IF LEVEL_OF_DETAIL node.

the default (n) is 0.

The IF LEVEL_OF_DETAIL node has one input that accepts an integer (0-32767) to change the value in the node.

With the following command sequence,

```
A := SET LEVEL_OF_DETAIL to 3 THEN B;
B := IF LEVEL_OF_DETAIL = 3 THEN C;
C := VECTOR_LIST .....;
```

initially when A is displayed, C is also displayed. If the level of detail is changed to something other than 3, then the test in B fails and C is not displayed.

An example of adding detail is to start with a sphere and add continents, mountain ranges, and countries. To display the parts of the world in this order (and turn them off in the reverse order):

Sphere
Continents
Mountain Ranges
Countries

the sphere needs to be displayed first and remain on while all subsequent parts are displayed.

The Continents need to be added next, the Mountain Ranges and then the Countries. If Sphere is displayed whenever there is a value of 1 or greater in the SET NODE, and the subsequent parts are displayed for values equal or greater than 2, 3, and 4, respectively, the desired effect is achieved.

The display tree that sets up such a level-of-detail condition is shown in Figure 6.

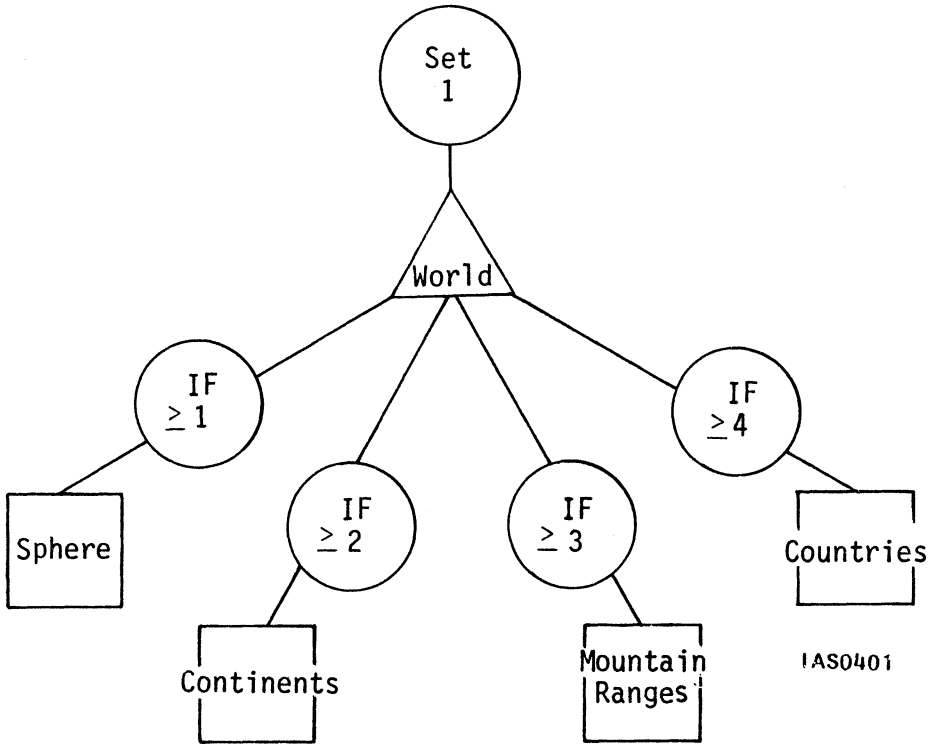


Figure 6. Level-of-Detail Structure for the World

By changing the value of the integer in the SET node, the parts of the Sphere can be laid on and stripped off. If the integer 2 is sent to the SET node, then the Sphere and the Continents are both displayed because both branches of the display tree meet the condition tested against the SET node. If the integer 3 is sent to the SET node, the Sphere, the Continents, and the Mountain ranges are all displayed. If the integer 4 is sent to the SET node, the entire structure is displayed. The details of the Sphere can be stripped off by decreasing the value in the SET node.

Using Level-Of-Detail Settings to Animate An Object

An example of using level-of-detail settings for animation is in the turbine blade portion of the PS 300 Demonstration Package. The turbine blade is defined as a sequence of turbine blades in slightly different positions. A clock is used to advance the level of detail settings resulting in the display sequence and the apparent motion of the turbine blade. The structure that sets this up is similar to the one shown in Figure 7.

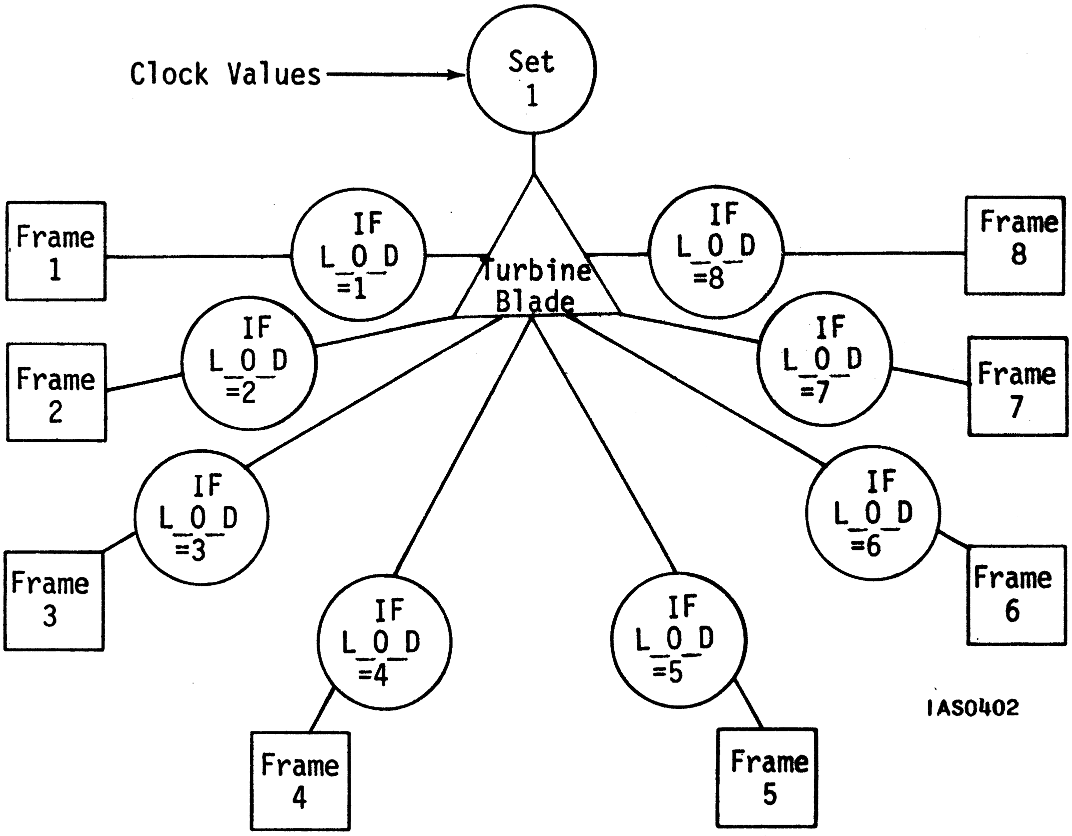


Figure 7. Turbine Blade Structure

The topmost node is the one supplied with clock values through a function network to step through the sequence of pictures corresponding to the referenced branches in the display tree.

Note that in animation, detail is not laid over a displayed picture. Instead, sequences of pictures are displayed.

Exercise

Load the tutorial tape and select ANIMATED_CYLINDER from the menu on the left side of the screen.

This demonstration is a good example of how level-of-detail settings can be used for local animation.

USING RATE ATTRIBUTE SETTINGS

The third type of conditional referencing allows you to blink an object or display tree branch under control of the refresh rate of the PS 300 display, an internal PS 300 clock, or an external clock. This type of conditional referencing can cause an object to blink or to be displayed alternately with another object. (For example, one part might be displayed for one second, then that part is removed while another part is displayed for a second, etc.)

Like the other types of conditional referencing, blinking requires two nodes. One node sets a blink rate in terms of phase on and off durations. The other if node tells tell whether an object or branch will be displayed during the on phase or the off phase.

Creating The Set Rate Node

The command to create the SET RATE node is:

Name := SET RATE *phase_on phase_off [initial state] [delay]*
APPLIED TO ***Name1***;

where:

phase_on phase_off are integers designating the durations of the on and off phases, respectively, in refresh frames.

initial_state is either ON or OFF, indicating the initial phase.

delay is an integer designating the number of refresh frames in the initial state.

Name1 is the descendent of the SET RATE node.

the default [initial state] is OFF.

Inputs for updating the SET RATE node are as follows:

INTEGER-------------------> <1> Changes the phase_on value.

INTEGER-------------------> <2> Changes the phase_off value.

BOOLEAN-------------------> <3> Changes the initial_state ON(T) / OFF(F).

INTEGER-------------------> <4> Changes the delay.

A command similar to SET RATE, called SET RATE EXTERNAL, allows you to alter the PHASE attribute via an external source such as a function network or a message from the host computer. Refer to the ***Command Summary*** for specific details of this command.

Creating the IF PHASE Node

The command to create the IF node to test the ON/OFF state of the phase is as follows:

Name := IF PHASE IS *state* THEN ***Name1***;

where:

state is the phase setting under which name1 is displayed (ON or OFF).

name1 is the descendent of the IF PHASE node.

If there is no SET RATE node or SET RATE EXTERNAL node higher in the structure, the "state" of the PHASE node will always be OFF.

For example, with the command sequence

```
Shape := SET RATE  10  15 THEN Blink_Shape;
Blink_Shape := IF PHASE ON THEN Sphere;
Sphere := VECTOR_LIST ....;
```

If Shape is displayed, Sphere will be displayed for 10 refresh frames and not displayed for 15 refresh frames repeatedly.

If the command sequence is

```
Shape := SET RATE  10  15 THEN Blink_Shape;
Blink_Shape := If PHASE OFF THEN Sphere;
Sphere := VECTOR_LIST .... ;
```

If Shape is displayed, Sphere will be displayed for 15 refresh frames and not displayed for 10 refresh frames repeatedly, since the condition is to display the vector list when the phase is OFF.

Exercise

This exercise uses the Robot created in the "PS 300 Command Language" module.

To demonstrate the effects of blinking, add blinking nodes above robot. The blink rate in this exercise will be based on the PS 300 refresh rate. First, define a node that sets the rate by entering:

```
Blink_Robot :=  SET RATE 120  60 APPLIED TO If_Robot;
```

This sets the ON phase to 120 refreshes and the OFF phase to 60 refreshes.

Now place a node that determines whether the robot will be displayed in the ON phase (and blanked in the OFF phase) or displayed in the OFF phase (and blanked in the ON phase). Display robot in the ON phase, by entering:

```
If_Robot := IF PHASE IS ON THEN ROBOT;
```

Robot should now blink at a rate of about 2 seconds on and one second off, when you:

```
DISPLAY Blink_Robot;
```

Then:

```
REMOVE Blink_Robot;
```

Some Uses for Timed Blinking

One practical use of the rate setting commands, other than the visual effects produced, is that they can synchronize the refresh rate of the display to a movie camera to make sure that the frame rate of the camera matches the frame refresh rate of the screen, allowing the camera to always be taking a frame as the picture is refreshed.

Stereo views can be created using a split screen (two viewports side by side), each half containing the same image and viewed with the EYE projection (refer to the "Viewing Operations" module). Then each viewport can be displayed alternately with the other viewport. By placing an opaque divider between the viewports so each eye can see only one viewport, a 3D effect can be generated.

SUMMARY

Conditional Referencing allows you display selected branches of a display tree without displaying other branches. These kinds of .operations are achieved using three methods: conditional-bit settings, level-of-detail settings, and rate settings.

To use conditional referencing, a minimum of two nodes must be placed in a display tree. The first node sets up the condition on which all subsequent references are tested. The second sets up the condition to be tested against the set condition.

Using Conditional Bit Settings

The conditional-bit method shown is used when you need to separately control the display of branches of your display tree in a variety of sequences.

The SET CONDITIONAL_BIT node sets any of 15 conditional bits (0-14). By placing the set conditional bit node above an instance node, bit settings affect all branches under the instance node.

This node is created with the SET CONDITIONAL_BIT command. The syntax is as follows:

```
Name := SET CONDITIONAL_BIT n switch APPLIED TO Name1;
```

where:

n is an integer from 0 to 14, corresponding to the conditional bit to be set ON or OFF.

switch is either ON or OFF.

name1 is the descendent node of the conditional bit node.

all bits default to OFF.

IF nodes (to test the condition of the SET node) are created with the IF CONDITIONAL_BIT Command. The syntax is as follows:

Name := IF CONDITIONAL_BIT ***n switch*** APPLIED TO ***Name1***;

where:

n is an integer from 0 to 14, indicating which bit to test.

switch is the setting to be tested, ON or OFF.

name1 is the descendent of the IF node.

Using Level of Detail Conditional Referencing

When using the level-of-detail method, the parts of the model are always displayed and removed in a set sequence. Level-of-detail is usually used to overlay detail on your picture.

Level of detail can also be used to run animation sequences comprised of a series of separate picture definitions.

Unlike conditional-bit referencing where 15 variables (bits) are set, only one variable is set using the level-of-detail method. All IF nodes are tested against that one variable in the SET node.

The command to create a set level-of-detail node is as follows.

Name := SET LEVEL_OF_DETAIL TO ***n*** APPLIED TO ***Name1***;

where:

n is an integer from 0 to 32767 indicating the level-of-detail value.

name1 is the descendent of the SET node.

the default level of detail (n) is 0.

Determining The Order for Overlaying Detail

Because level-of-detail controls the display of branches in a determined order, the conditional statements are expressed as relationships rather that the two-state (on/off) type used in conditional-bit references.

These relationships are:

Less Than	<
Less Than Or Equal To	<=
Equal To	=
Not Equal To	<>
Greater Than Or Equal To	>=
Greater Than	>

and are specified in the IF LEVEL_OF_DETAIL node. The command to create this IF node is as follows:

```
Name := IF LEVEL_OF_DETAIL relationship n THEN Name1;
```

where:

relationship is the relationship to be tested (<, <=, =, <>, >=, >).

n is an integer from 0 to 32767 indicating the number (along with the previous relationship) to compare against the current level of detail setting.

name1 is the descendent of the IF LEVEL_OF_DETAIL node.

the default is (n) 0.

Using Level-Of-Detail Settings to Animate An Object

An example of using level-of-detail settings for animation is in the turbine blade portion of the ***PS 300 Demonstration Package***. The turbine blade is defined as a sequence of turbine blades in slightly different positions. A clock is used to advance the level-of-detail settings resulting in the display sequence and the apparent motion of the turbine blade.

Blinking and Alternately Displaying parts of an Object

The third type of conditional referencing, rate attribute settings, allows you to blink an object or display tree branch under control of the refresh rate of the PS 300 display, an internal PS 300 clock, or an external clock. This type of conditional referencing can cause an object to blink or to be displayed alternately with another object. (For example, one part might be displayed for one second, then that part is removed while another part is displayed for a second, etc.)

Like the other types of conditional referencing, blinking requires two nodes. One node sets a blink rate in terms of phase ON and OFF durations. The other IF node tells whether an object or branch will be displayed during the ON phase or the OFF phase.

Creating The Set Rate Node

The command to create the SET RATE node is:

Name := SET RATE *phase_on phase_off [initial state] [delay]*
APPLIED TO ***Name1***;

where:

phase_on phase_off are integers designating the durations of the on and off phases, respectively, in refresh frames.

initial_state is either ON or OFF, indicating the initial phase.

delay is an integer designating the number of refresh frames in the initial state.

name1 is the descendent of the SET RATE node.

the default [initial state] is OFF.

A command similar to SET RATE, called SET RATE EXTERNAL, allows you to alter the PHASE attribute via an external source such as a function network or a message from the host computer. Refer to the ***Command Summary*** for specific details of this command.

Creating the IF PHASE Node

The command to create the IF node to test the ON/OFF state of the phase is as follows:

Name := IF PHASE IS *state* THEN ***Name1***;

where:

state is the phase setting to be tested (ON or OFF).

name1 is the descendent of the SET RATE node.

If there is no SET RATE node or SET RATE EXTERNAL node higher in the structure, the state of the PHASE node will always be OFF.

You now know how to make conditional references to parts of your display tree. You know that two nodes are required for each conditional reference. The first node sets up the condition on which all subsequent references are tested. The second sets up the condition to be tested against the set condition.

The flexibility and ease of use of conditional referencing within the display structure makes what is often a difficult operation on other graphics machines easy on the PS 300.

FUNCTION NETWORKS II

SWITCHING NETWORKS

CONTENTS

FUNCTION NETWORKS II

ILLUSTRATIONS

This module consists of four sections that build on ideas about function networks introduced in the "Function Networks I" module.

In "Function Networks I" you used the PS 300 dials to manipulate a robot. Each dial was connected to a node in the robot display tree so that moving the dial caused Robot to move in a specific way. One dial was needed for each manipulation.

In this module, you will learn how to use a dial for multiple interactions. This can be done using function networks and PS 300 function keys. Pressing a function key allows you to use the same dial for different kinds of interactions in different modes.

The module also details how to send a label to the LEDs above each dial. These labels remind you of a dial's function and can change interactively each time a new function key is pressed.

In addition, you will learn about several useful tasks which function networks can perform. These include limiting the robot movement so that it remains "true to life," and using variables to store values coming from a network.

Because the function networks in this module will differ from those created in "Function Networks I," it is suggested that you save the code from this module in a separate file on your host. To avoid errors, do not combine these two sets of code.

OBJECTIVES

In this module you will learn how to:

- Make a single input device (the dials) control multiple interactions.
- Label the dials so that the label changes when the dial's function changes.
- Set limits on the motion of a model.
- Use variables to store values.

PREREQUISITES

Before beginning this module, you should be familiar with the concepts presented in the following modules: "Modeling," "PS 300 Command Language," and "Function Networks I."

MAKING A SINGLE INPUT DEVICE CONTROL MULTIPLE INTERACTIONS

In "Function Networks I," you constructed a function network for the display tree shown in Figure 1.

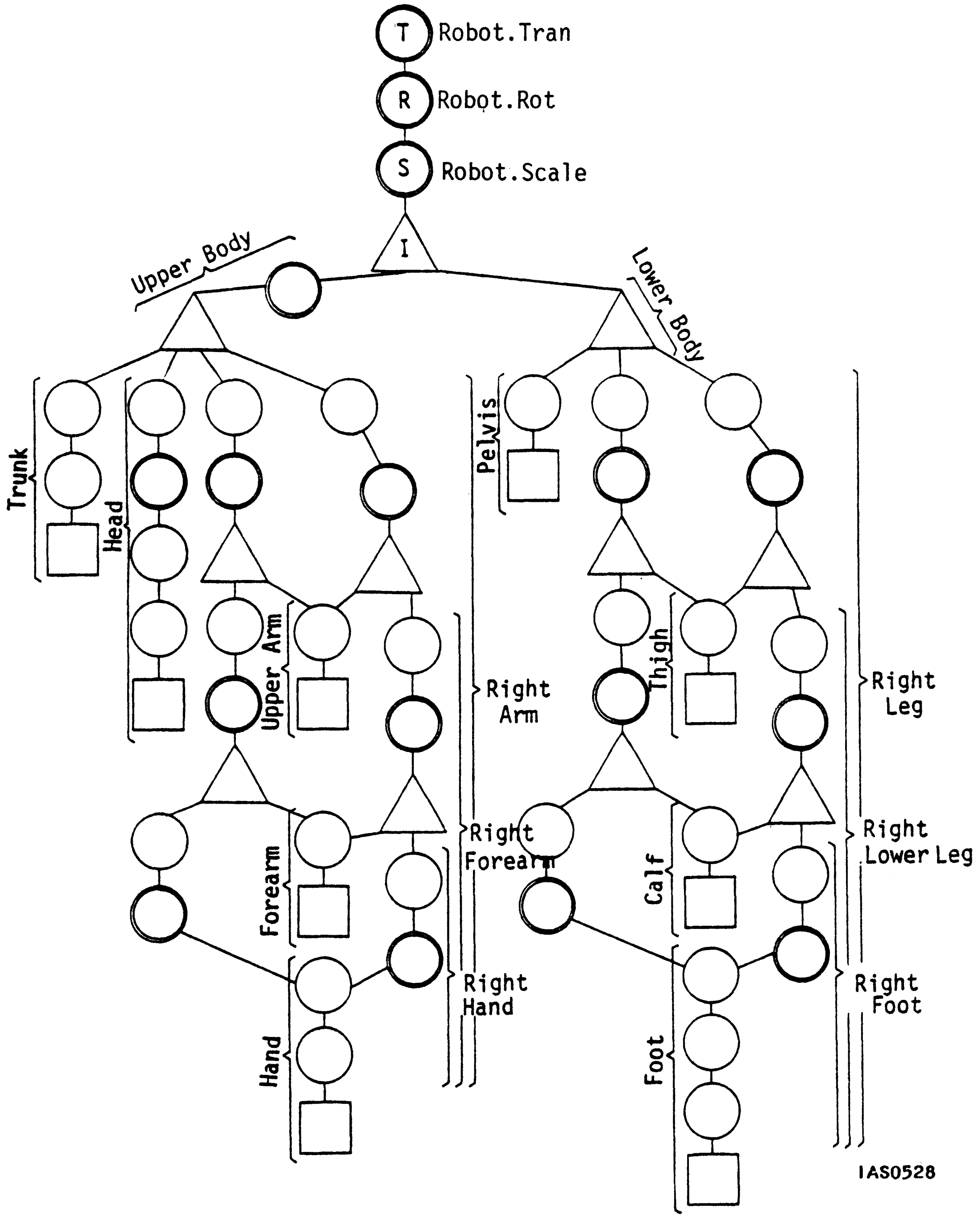


Figure 1. Robot Display Tree

This function network supplied interactions for the top three nodes of the display tree: Robot.Scale, Robot.Rot, and Robot.Tran. Seven dials were required to manipulate the robot: three to rotate it in the X, Y, and Z planes, three to translate it in X, Y, and Z, and one dial to scale the model.

Only one free dial remains, but no other interactive nodes in the robot display tree have yet been connected to functions. To supply X, Y, and/or Z rotations for all the other interactive nodes would require dozens of other dials. This section illustrates how to solve this problem by making one set of eight dials perform like many sets.

The first step in doing this is to determine exactly how many additional dials you will need (how many more interactions in the model you want to control). In addition to Robot.Rot, the robot has 14 rotation nodes. Ten of them require three dials each (three rotations for X, Y, and Z). The two nodes for elbows and the two for knees only use X rotations, requiring only one dial each. The result is a total of 34 additional interactions. To handle these interactions, each dial would have to be connected to about six nodes.

There is nothing to prevent you from connecting a dial to more than one destination. For example, you could hook dial 1, already updating X rotations for the Robot.Rot node, to other rotate nodes. But of course turning that one dial would cause multiple unrelated updates.

Following is one way the dials might logically be assigned to control the interactions.

In Mode 1, the dials would work as presently assigned:

Whole model:	1. Xrot	2. Yrot	3. Zrot	4. Scale
	5. Xtran	6. Ytran	7. Ztran	8. Not Assigned

Mode 2:

Head:	1. Xrot	2. Yrot	3. Zrot	4. Not Assigned
Trunk:	5. Xrot	6. Yrot	7. Zrot	8. Not Assigned

Mode 3:

Right arm:	1. Xrot	2. Yrot	3. Zrot	4. Elbow Xrot
Left arm:	5. Xrot	6. Yrot	7. Zrot	8. Elbow Xrot

Mode 4:

Right hand:	1. Xrot	2. Yrot	3. Zrot	4. Not assigned
Left hand:	5. Xrot	6. Yrot	7. Zrot	8. Not assigned

Mode 5:

Right leg:	1. Xrot	2. Yrot	3. Zrot	4. Knee Xrot
Left leg:	5. Xrot	6. Yrot	7. Zrot	8. Knee Xrot

Mode 6:

Right foot:	1. Xrot	2. Yrot	3. Zrot	4. Not Assigned
Left foot:	5. Xrot	6. Yrot	7. Zrot	8. Not Assigned

This configuration leaves several dials unassigned in a few modes. Obviously, you could assign every dial in every mode, but this organization establishes a pattern that makes the dials' functions easy to remember.

Another way to diagram this same dial assignment would be as follows. The names of the nodes on the right are linked to the dials on the left.

DIALS[1]------Xrot	Whole body (1)
	Head (2)
	Right arm (3)
	Right hand (4)
	Right leg (5)
	Right foot (6)
DIALS[2]------Yrot	Whole body (1)
	Head (2)
	Right arm (3)
	Right hand (4)
	Right leg (5)
	Right foot (6)
DIALS[3]------Zrot	Whole body (1)
	Head (2)
	Right arm (3)
	Right hand (4)
	Right leg (5)
	Right foot (6)

DIALS[4]	Whole body scale(1) Right elbow Xrot (3) Right knee Xrot (5)
DIALS[5]	Whole body Xtran (1) Trunk Xrot (2) Left arm Xrot (3) Left hand Xrot (4) Left leg Xrot (5) Left foot Xrot (6)
DIALS[6]	Whole body Ytran (1) Trunk Yrot (2) Left arm Yrot (3) Left hand Yrot (4) Left leg Yrot (5) Left foot Yrot (6)
DIALS[7]	Whole body Ztran (1) Trunk Zrot (2) Left arm Zrot (3) Left hand Zrot (4) Left leg Zrot (5) Left foot Zrot (6)
DIALS[8]	Left elbow Xrot (3) Left knee Xrot (5)

If the connections were made from the dials as shown, a dial would control several interactions simultaneously. If you turned Dial 4, for instance, the robot would become larger or smaller, or its right knee and elbow would move. Dial 1, connected to six nodes, would cause six separate X rotations in the model.

What is needed now is the equivalent of a switch in a railroad yard to route values so that they are not routed down all function network paths at once. For example, you might want to send values to the Robot.Rot node only in dials Mode 1, or just to Head.Rot node in Mode 2.

Associated with all the function keys is one system function, FKEYS. FKEYS has one output. When you press a function key, the number of that key is output. For example, pressing key #4 causes an integer 4 to be output.

The value could be output to an instance of function F:CROUTE(n) (see Figure 2). This switching function allows you to channel the values from the dials (or anything else) to any number (n) of destinations.

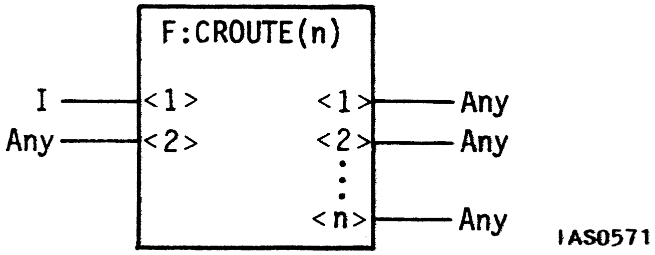


Figure 2. F:CROUTE(n) Function

Specifically, when F:CROUTE(n) receives an integer from 1 to n on input <1>, it routes what it receives on input <2> to the output with the same number as the integer. So if you instance F:CROUTE, connect FKEYS to input <1> of the function instance, connect the dials to input <2>, and press Function Key 5, the values from the dials arriving on input <2> will travel out on output <5> (see Figure 3).

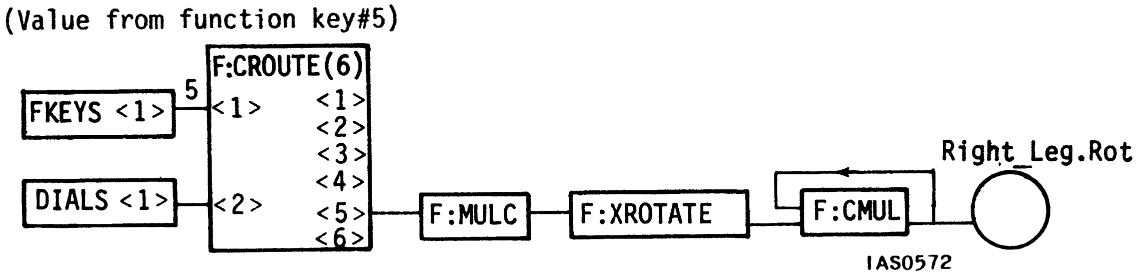


Figure 3. F:CROUTE(n) Network -- Example 1

Pressing Function Key 3 routes the values from Dial 1 to output <3> (Figure 4).

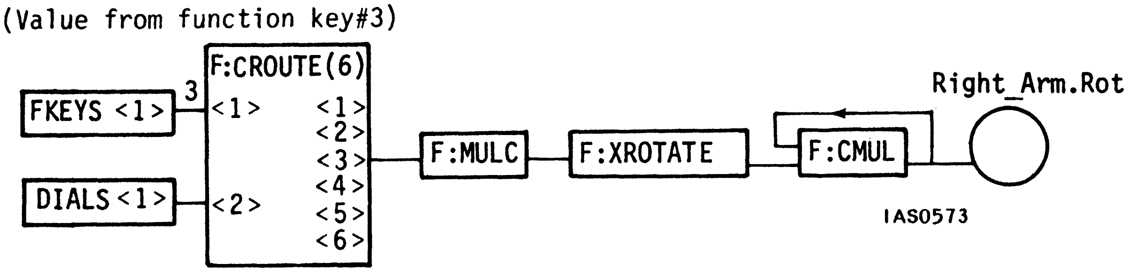


Figure 4. F:CROUTE(n) Network -- Example 2

In this example, the number of destinations from a routing function is the same as the number of modes among the function switches. For Dial 1, that is six modes, so Dial 1 will use an instance of F:CROUTE(6), as shown in the above diagrams.

Not all dials need to work in all six modes. Dial 4, for example, only works in 3 modes, so you might try using an instance of F:CROUTE(2). Dial 4 has to operate in Mode 5, however, so you must use 5 as a minimum value for n, as shown below. The unused outputs (for modes in which Dial 4 is unassigned) are left unconnected (Figure 5).

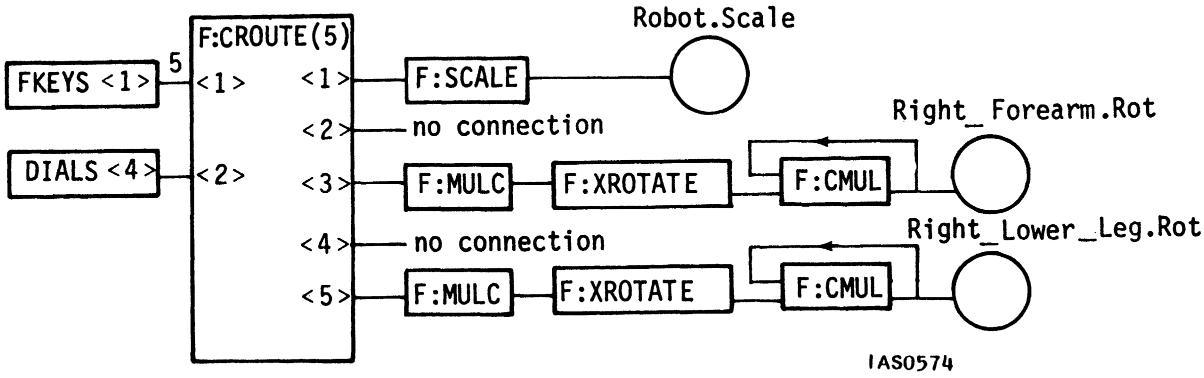


Figure 5. F:CROUTE(n) Network With Unused Outputs

The diagram indicates that the values from Dial 4 will be routed to the scaling node, Robot.Scale, when FKEYS sends 1 to F:CROUTE(5) input <1>. Values from Dial 4 will go to the right knee when a 5 arrives on input <1> and to the right elbow when a 3 arrives. If you push Function Keys 2 or 4 to go into Mode 2 or 4, Dial 4 has no effect.

Dial 8 is similar to Dial 4, but instead of working in three modes, it only works in two. One of the two modes it works in is Mode 5, so be sure to use an instance of F:CROUTE(5) with Dial 8 too.

Connect all six modes for Dial 1 to the outputs of F:CROUTE(6) so that FKEYS will control routing for this dial. Figure 6 illustrates Dial 1's F:CROUTE(n) network.

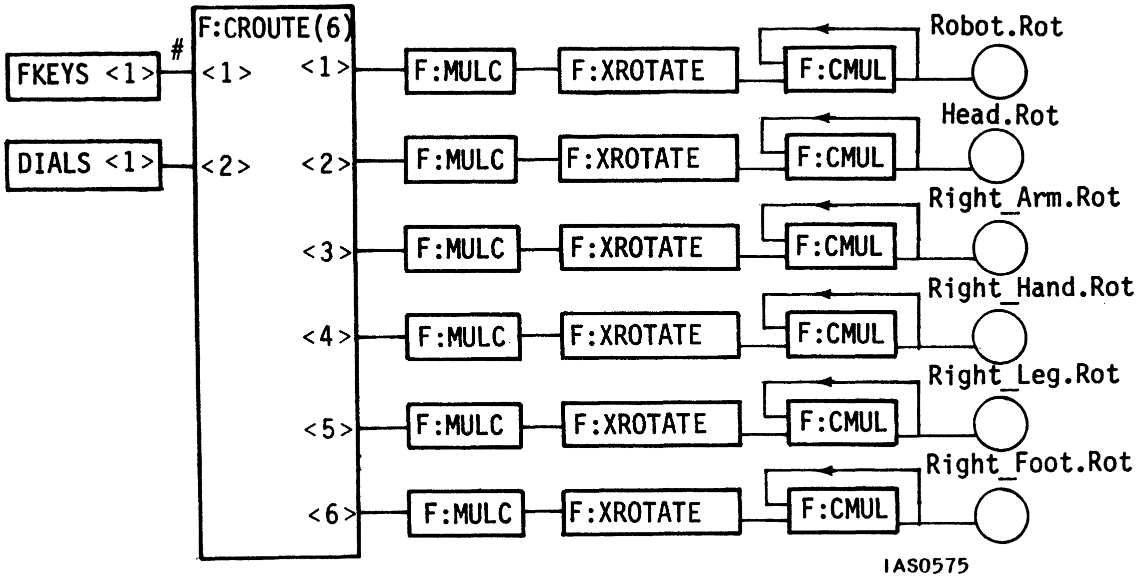


Figure 6. Dial 1's F:CROUTE(n) Network

Notice that the MULC and XROTATE functions in all six modes are exactly alike. The CMUL functions are not, since each one accumulates rotations for a different rotation node. What is exactly alike can be used once on the left side of the routing function, as shown in Figure 7.

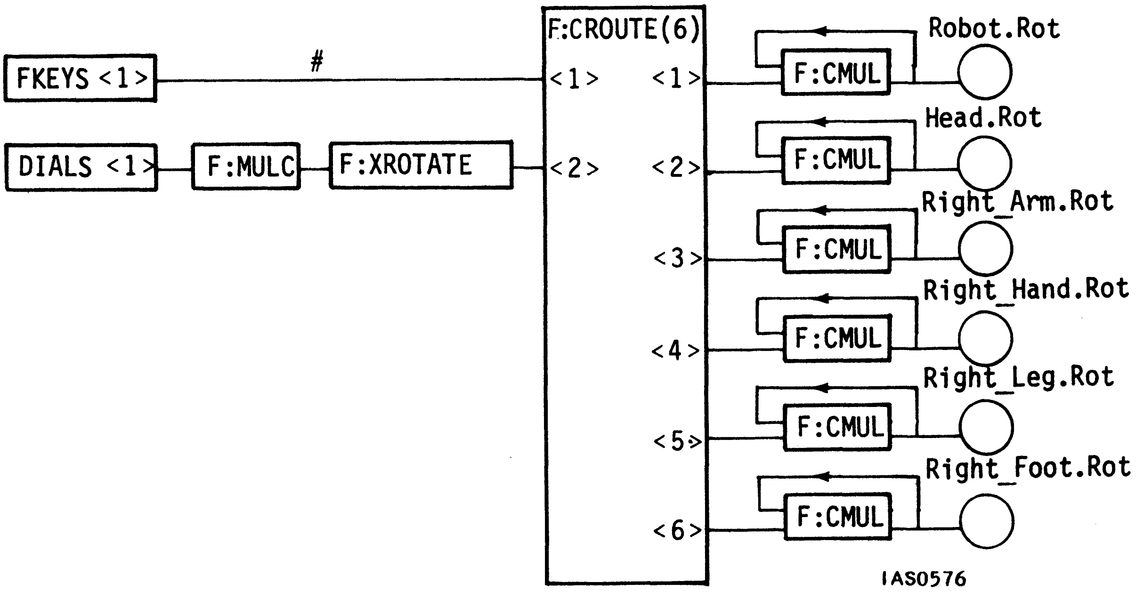


Figure 7. Dial 1's F:CROUTE(n) Network With Shared Functions

Either of the above two configurations would work. The second one is much less trouble to diagram and program, since it requires only one instance of F:MULC and F:XROTATE instead of six. The previous two diagrams show that a routing function is necessary only when a path must split, and that occurs when functions need to be unique, as in the case of the F:CMULs.

Now diagram networks Dials 2 and 3 using the diagram from Dial 1 as a guide. Since all three dials have the same destination nodes, you can route them through the same switching function, as in Figure 8.

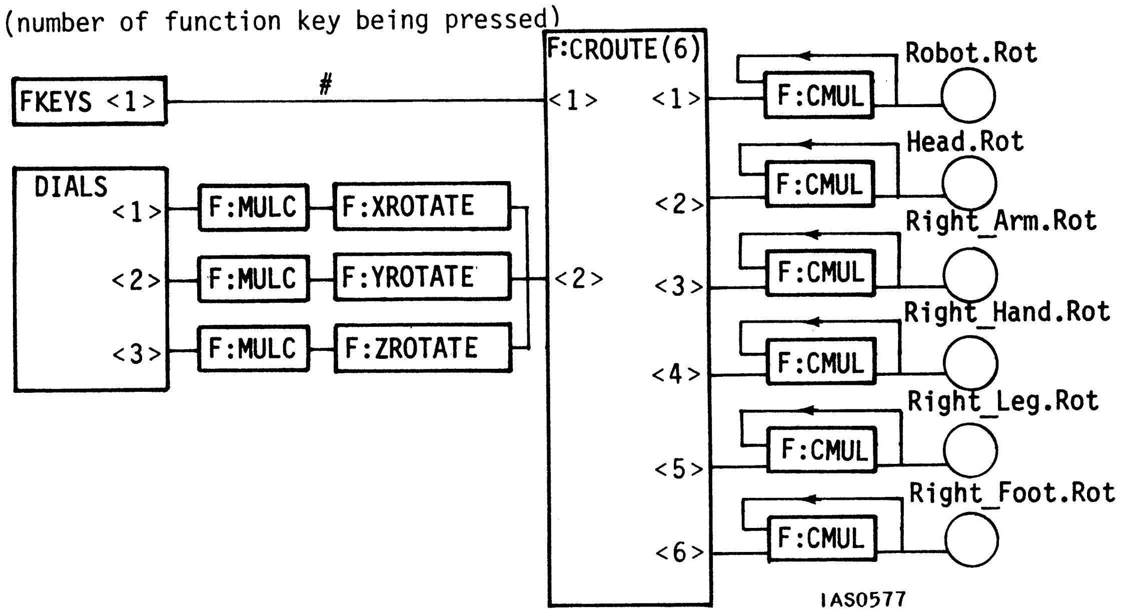
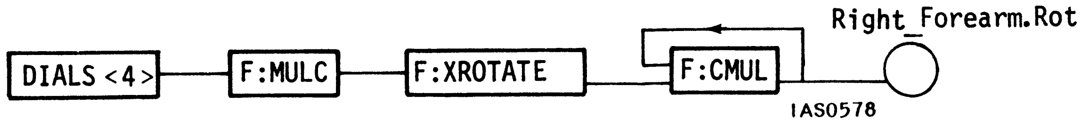


Figure 8. Final Network for Dials 1-3

This diagram completely accounts for the first three dials in all six modes. To implement it in the PS 300, you only need to fill in detail familiar from "Function Networks 1": connections, function instance names, and so on.

Next, look at Dial 4. Since it performs rotations, you might think to use the same rotation network for it as the first three dials, namely:



No other dials feed into that node, though, or the other rotate node for the knee that Dial 4 controls. So it would be simpler to use the F:DXROTATE function here. It is the function that combines all features of F:MULC, F:XROT, and F:CMUL into one package. The network for Dial 4 can be diagrammed as in Figure 9.

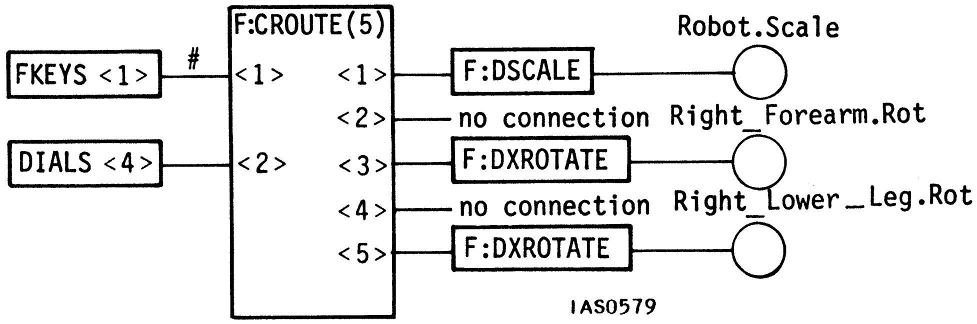


Figure 9. Final Network for Dial 4

With Dial 4, there are no functions on the right of the routing function that can be shared and moved over to the left, as with F:MULC and F:ROTATE functions used with Dials 1, 2, and 3. The above diagram completely specifies what Dial 4 will do in all modes. And to implement it, you must supply function instance names, initial values, and so on.

Dials 5, 6, and 7 do almost exactly what Dials 1, 2, and 3 do, but to the left side of the model. And in Mode 1, they translate instead of rotate. In Mode 1, all three dials feed into one node, Robot.Tran.

In the other five modes, they do X, Y, and Z rotations. Figure 10 illustrates how a routing function for Dial 5 might work.

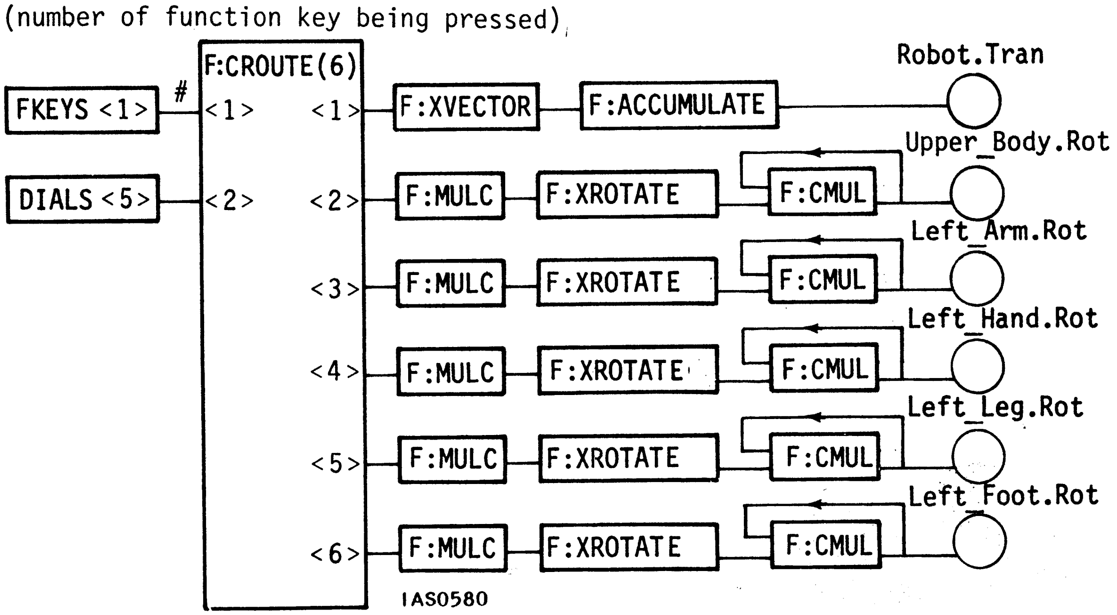


Figure 10. Sample Function Network for Dial 5

Of course, the diagram would be similar for Dials 6 and 7, with Y and Z rotations substituted for X.

Note that the MULC and XROT functions in Modes 2 through 6 above are exactly the same and could be shared as in Figure 11.

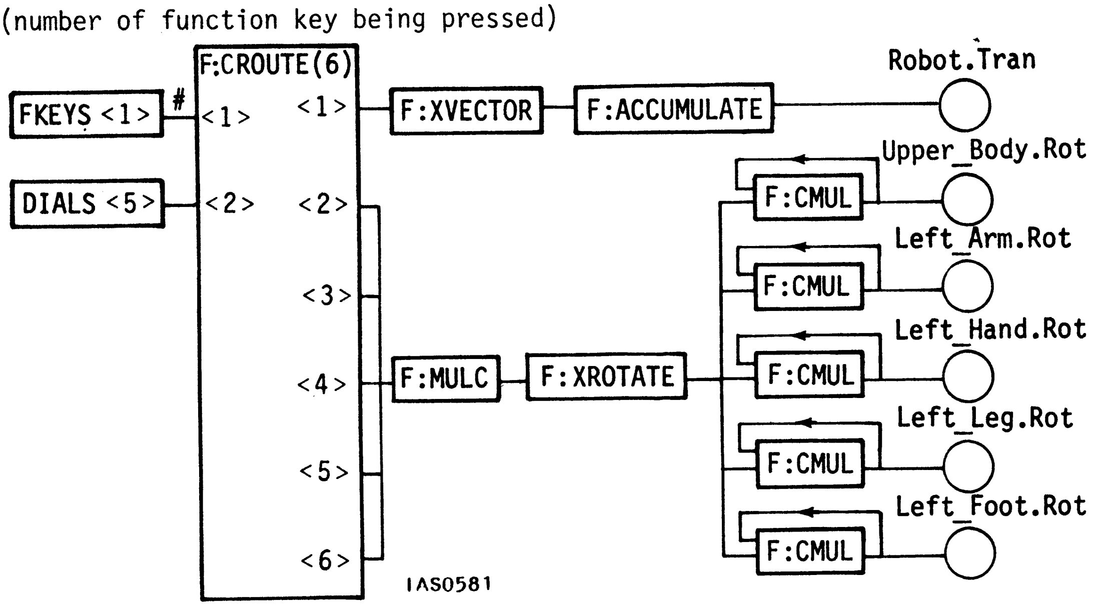


Figure 11. Dial 5 Network With Shared Functions

This will save you having five sets of MULC and XROT functions when one can do the job. But the output from XROT will have to be routed, so you'll need another routing function. The final network for Dial 5 is shown in Figure 12.

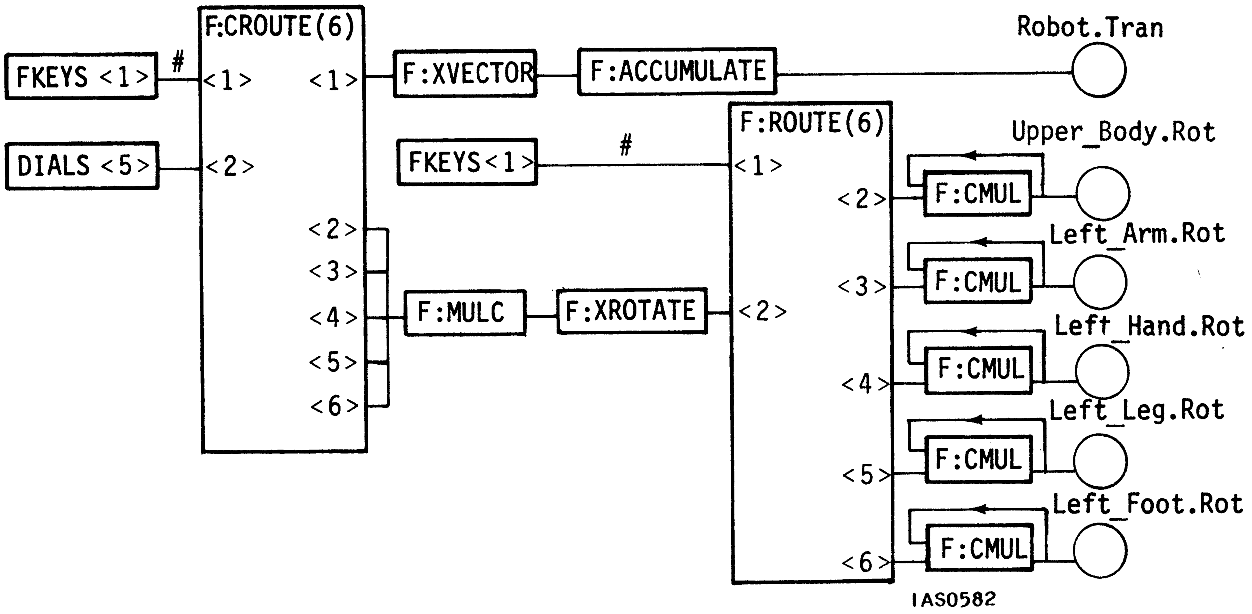


Figure 12. Final Function Network for Dial 5

Functionally, this completely specifies what Dial 5 does.

Exercise

Complete the network for Dials 6 and 7 using Dial 5 as a pattern. Then diagram the network for the Dial 8, using Dial 4 as a pattern.

Next, code the networks for all eight dials. Include all the details, such as instancing functions, connecting functions, and sending initial values to functions when needed. Remember that the DIALS and FKEYS functions have already been instanced by the system and do not need to be named by you. To save these commands, do this in a text file.

Once the commands to implement the network for one dial are detailed, you can copy them over again for each of the other dials and delete or add only the details you want. For example, all the commands to implement this network for Dial 1 (X rotations) are the same as for Dial 2, except you need to change X to Y and so on.

Figure 13 illustrates the final function network for Dials 1-8.

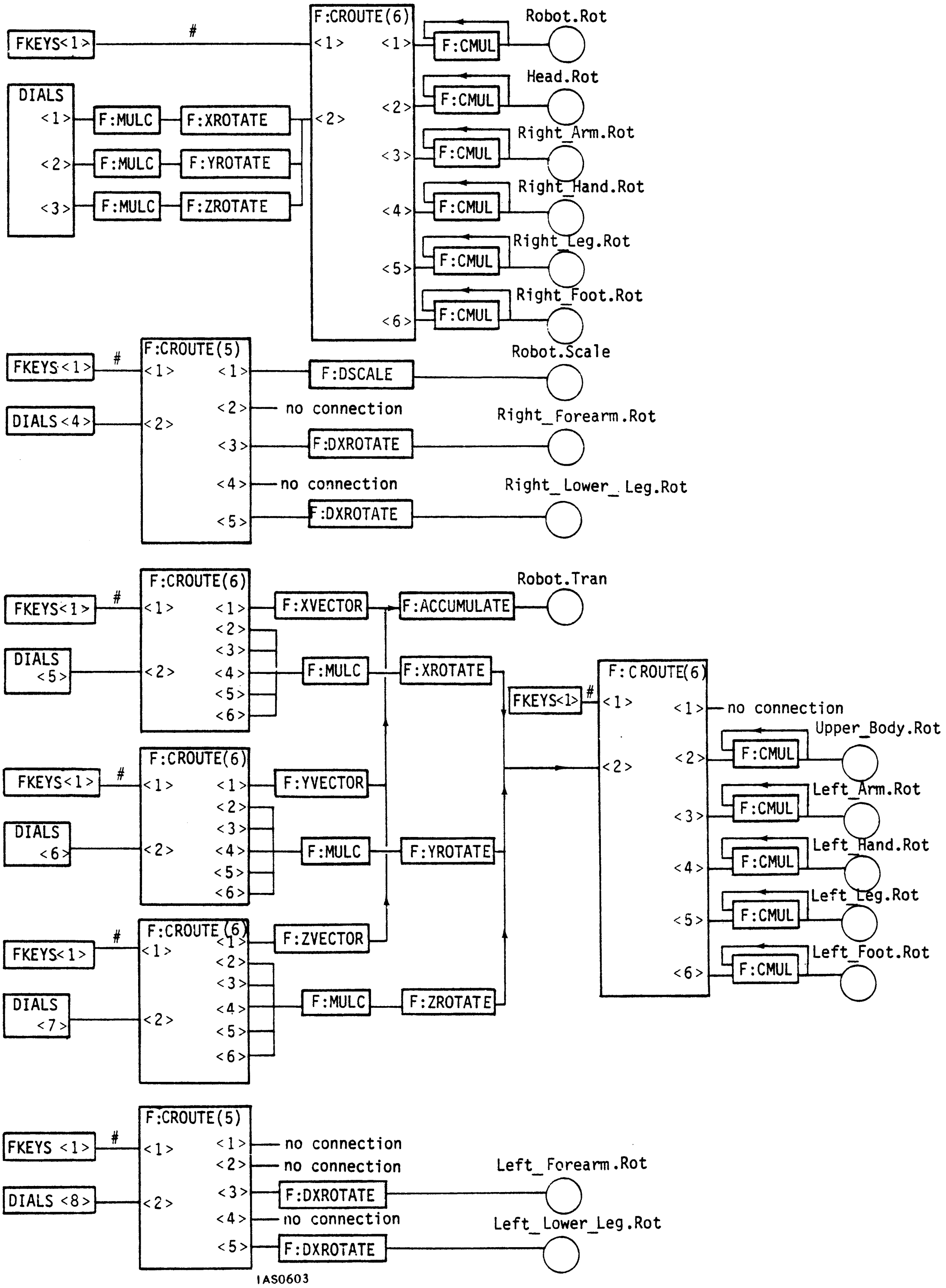


Figure 13. Final Function Network for Dials 1-8

The following lists the commands needed to code the function network. The code has been organized by dial, so that functions are instanced, connected, and primed for each dial, or group of dials, before preceding to the next dial. The names are suggestive of what each function instance does. Comment lines have been provided for clarification.

{CODE FOR DIALS 1-3}

```
X_Mul_D1 := F:MULC;                             {Instance MULC and}
Y_Mul_D2 := F:MULC;                             {ROT functions}
Z_Mul_D3 := F:MULC;

X_Rot_D1 := F:XROT;
Y_Rot_D2 := F:YROT;
Z_Rot_D3 := F:ZROT;

Switch1 := F:CROUTE(6);                         {Instance SWITCH and}
                                                {CMUL functions}

Acc_Rot_Robot := F:CMUL;
Acc_Rot_Head := F:CMUL;
Acc_Rt_Arm := F:CMUL;
Acc_Rt_Hand := F:CMUL;
Acc_Rt_Leg := F:CMUL;
Acc_Rt_Foot := F:CMUL;

CONNECT FKEYS<1>:<1>Switch1;                    {Connect FKEYS and}
                                                {DIALS}

CONNECT DIALS<1> : <1>X_Mul_D1;
CONNECT DIALS<2> : <1>Y_Mul_D2;
CONNECT DIALS<3> : <1>Z_Mul_D3;

CONNECT X_Mul_D1<1> : <1>X_Rot_D1;              {Connect rotation}
CONNECT Y_Mul_D2<1> : <1>Y_Rot_D2;              {accumulator to rotate}
CONNECT Z_Mul_D3<1> : <1>Z_Rot_D3;              {function}

CONNECT X_Rot_D1<1> : <2>Switch1;               {Connect rotate function}
CONNECT Y_Rot_D2<1> : <2>Switch1;               {to switch}
CONNECT Z_Rot_D3<1> : <2>Switch1;

CONNECT Switch1<1> : <2>Acc_Rot_Robot;          {Connect switch to}
CONNECT Switch1<2> : <2>Acc_Rot_Head;           {X,Y,Z accumulator}
CONNECT Switch1<3> : <2>Acc_Rt_Arm;
CONNECT Switch1<4> : <2>Acc_Rt_Hand;
CONNECT Switch1<5> : <2>Acc_Rt_Leg;
CONNECT Switch1<6> : <2>ACC_Rt_Foot;
CONNECT Acc_Rot_Robot<1> : <1> Acc_Rot_Robot;     {Connect X,Y,Z}
CONNECT Acc_Rot_Robot<1> : <1> Robot.Rot;         {accumulator back to}
                                                  {self and to display tree}
                                                  {node}

CONNECT Acc_Rot_Head<1> : <1> Acc_Rot_Head;
CONNECT Acc_Rot_Head<1> : <1> Head.Rot;

CONNECT Acc_Rt_Arm<1> : <1> Acc_Rt_Arm;
CONNECT Acc_Rt_Arm<1> : <1> Right_Arm.Rot;

CONNECT Acc_Rt_Hand<1> : <1> Acc_Rt_Hand;
CONNECT Acc_Rt_Hand<1> : <1> Right_Hand.Rot;

CONNECT Acc_Rt_Leg<1> : <1> Acc_Rt_Leg;
CONNECT Acc_Rt_Leg<1> : <1> Right_Leg.Rot;

CONNECT Acc_Rt_Foot<1> : <1>Acc_Rt_Foot;
CONNECT Acc_Rt_Foot<1> : <1>Right_Foot.Rot;

SEND 200 TO <2>X_Mul_D1;                          {Prime MULC function}
SEND 200 TO <2>Y_Mul_D2;
SEND 200 TO <2>Z_Mul_D3;

SEND M3d (1,0,0 0,1,0 0,0,1) TO <1>Acc_Rot_Robot; {Prime CMUL function}
SEND M3d (1,0,0 0,1,0 0,0,1) TO <1>Acc_Rot_Head;
SEND M3d (1,0,0 0,1,0 0,0,1) TO <1>Acc_Rt_Arm;
SEND M3d (1,0,0 0,1,0 0,0,1) TO <1>Acc_Rt_Hand;
SEND M3d (1,0,0 0,1,0 0,0,1) TO <1>Acc_Rt_Leg;
SEND M3d (1,0,0 0,1,0 0,0,1) TO <1>Acc_Rt_Foot;


                       {CODE FOR DIAL 4}


Switch2:= F:CROUTE(6);                            {Instance switch function}

Scale_Robot:= F:DSCALE;                           {Instance scale & rot}
Rot_Rt_Elbow := F:DXROTATE;                       {functions}
Rot_Rt_Knee := F:DXROTATE;


CONNECT FKEYS<1> : <1>Switch2;                    {Connect FKEYS and}
                                                  {DIALS}

CONNECT DIALS<4> : <2>Switch2;

CONNECT Switch2<1> : <1>Scale_Robot;              {Connect switch to scale}
CONNECT Switch2<3> : <1>Rot_Rt_Elbow;             {and rot functions}
CONNECT Switch2<5> : <1>Rot_Rt_Knee;

CONNECT Scale_Robot<1> : <1> Robot.Scale;         {Connect scale & rot}
CONNECT Rot_Rt_Elbow<1> : <1>Right_Forearm.Rot;   {functions to display tree}
CONNECT Rot_Rt_Knee<1> : <1> Right_Lower_Leg.Rot; {nodes}

SEND .075 TO <2>Scale_Robot;                      {Prime scale function}
SEND .02 TO <3>Scale_Robot;
SEND .1 TO <4>Scale_Robot;
SEND .025 TO <5>Scale_Robot;

SEND 0 TO <2>Rot_Rt_Elbow;                        {Prime rotation}
SEND 200 TO <3>Rot_Rt_Elbow;                      {functions}
SEND 0 TO <2>Rot_Rt_Knee;
SEND 200 TO <3> Rot_Rt_Knee;


                        {CODE FOR DIAL 5}


Switch3:= F:CROUTE(6);                            {Instance both switch}
Switch6:= F:CROUTE(6);                            {functions}

X_Vec_D5:= F:XVEC;                                {Instance X vector for}
X_Mul_D5:=F:MULC;                                 {translation}
X_Rot_D5 := F:XROT;                               {Instance MULC and}
                                                  {ROT functions}

Acc_Trans:= F:ACCUM;                              {Instance tran}
                                                  {accumulate function}

Acc_Rot_Trunk:= F:CMUL;                           {Instance CMUL}
Acc_Lt_Arm:=F:CMUL;                               {functions}
Acc_Lt_Hand:=F:CMUL;
Acc_Lt_Leg:= F:CMUL;
Acc_Lt_Foot:=F:CMUL;

CONNECT FKEYS<1> : <1>Switch3;                    {Connect FKEYS and}
CONNECT FKEYS<1> : <1>Switch6;                    {DIALS}
CONNECT DIALS<5> : <2>Switch3;

CONNECT Switch3<1> : <1>X_Vec_D5;                 {Finish connections for}
CONNECT X_Vec_D5<1> : <1>Acc_Trans;               {trans network}
CONNECT Acc_Trans<1> : <1>Robot.Tran;

CONNECT Switch3<2> : <1>X_Mul_D5;                 {Connect switch to}
CONNECT Switch3<3> : <1>X_Mul_D5;                 {MULC functions}
CONNECT Switch3<4> : <1>X_Mul_D5;
CONNECT Switch3<5> : <1>X_Mul_D5;
CONNECT Switch3<6> : <1>X_Mul_D5;
CONNECT X_Mul_D5<1> : <1>X_Rot_D5;            {Connect MULC to}
CONNECT X_Rot_D5<1> : <2>Switch6;             {rotation function}
                                              {Connect rotation}
                                              {function to other switch}

CONNECT Switch6<2> : <2>Acc_Rot_Trunk;        {Connect switch to}
CONNECT Switch6<3> : <2>Acc_Lt_Arm;           {CMUL functions}
CONNECT Switch6<4> : <2>Acc_Lt_Hand;
CONNECT Switch6<5> : <2>Acc_Lt_Leg;
CONNECT Switch6<6> : <2>Acc_Lt_Foot;

CONNECT Acc_Rot_Trunk<1> : <1>Acc_Rot_Trunk;  {Connect CMUL}
CONNECT Acc_Rot_Trunk<1> : <1>Upper_Body.Rot; {functions back to self}
                                              {and to display tree
                                              nodes}

CONNECT Acc_Lt_Arm<1> : <1>Acc_Lt_Arm;
CONNECT Acc_Lt_Arm<1> : <1>Left_Arm.Rot;

CONNECT Acc_Lt_Hand<1> : <1>Acc_Lt_Hand;
CONNECT Acc_Lt_Hand<1> : <1>Left_Hand.Rot;

CONNECT Acc_Lt_Leg<1> : <1>Acc_Lt_Leg;
CONNECT Acc_Lt_Leg<1> : <1>Left_Leg.Rot;

CONNECT Acc_Lt_Foot<1> : <1>Acc_Lt_Foot;
CONNECT Acc_Lt_Foot<1> : <1>Left_Foot.Rot;

SEND 200 TO <2>X_Mul_D5;                      {Prime MULC function}

SEND M3d (1,0,0 0,1,0 0,0,1) TO <1>Acc_Rot_Trunk; {Prime CMUL function}
SEND M3d (1,0,0 0,1,0 0,0,1) TO <1>Acc_Lt_Arm;
SEND M3d (1,0,0 0,1,0 0,0,1) TO <1>Acc_Lt_Hand;
SEND M3d (1,0,0 0,1,0 0,0,1) TO <1>Acc_Lt_Leg;
SEND M3d (1,0,0 0,1,0 0,0,1) TO <1>Acc_Lt_Foot;

SEND V3d (0,0,0) TO <2>Acc_Trans;             {Prime trans accumulate}
                                              {function}
SEND 0 TO <3>Acc_Trans;
SEND 1 TO <4>Acc_Trans;
SEND 10 TO <5>Acc_Trans;
SEND -10 TO <6>Acc_Trans;
```

```
                              {CODE FOR DIAL 6}

Switch4:= F:CROUTE(6);                          {Instance Switch}
                                                {function.  Note: 2nd}
                                                {Switch already}
                                                {instanced}

Y_Vec_D6:= F:YVEC;                              {Instance X vector for}
                                                {translation}

Y_Mul_D6:=F:MULC;                               {Instance MULC and}
                                                {ROT functions}

Y_Rot_D6:= F:YROT;

CONNECT FKEYS<1> : <1>Switch4;                  {Connect FKEYS and}
                                                {DIALS}
CONNECT DIALS<6> : <2>Switch4;

CONNECT Switch4<1> : <1>Y_Vec_D6;               {Finish connections for}
                                                {trans network}
CONNECT Y_Vec_D6<1> : <1>Acc_Trans;

CONNECT Switch4<2> : <1>Y_Mul_D6;               {Connect switch to}
                                                {MULC functions}
CONNECT Switch4<3> : <1>Y_Mul_D6;
CONNECT Switch4<4> : <1>Y_Mul_D6;
CONNECT Switch4<5> : <1>Y_Mul_D6;
CONNECT Switch4<6> : <1>Y_Mul_D6;

CONNECT Y_Mul_D6<1> : <1>Y_Rot_D6;              {Connect MULC to}
                                                {rotation function}

CONNECT Y_Rot_D6<1> : <2>Switch6;               {Connect rotation}
                                                {function to other switch}

SEND 200 TO <2>Y_Mul_D6;                        {Prime MULC function}

                              {CODE FOR DIAL 7}

Switch5:= F:CROUTE(6);                          {Instance Switch}
                                                {function.  Note: 2nd}
                                                {Switch already}
                                                {instanced}
```

```
Z_Vec_D7:= F:ZVEC;                            {Instance Z vector for}
                                              {translation}

Z_MUL_D7:=F:MULC;                             {Instance MULC and}
                                              {ROTfunctions}

Z_ROT_D7 := F:ZROT;

CONNECT FKEYS<1> : <1>Switch5;                {Connect FKEYS and}
                                              {DIALS}
CONNECT DIALS<7> : <2>Switch5;

CONNECT Switch5<1> : <1>Z_Vec_D7;             {Finish connections for}
                                              {trans network}
CONNECT Z_Vec_D7<1> : <1>Acc_Trans;

CONNECT Switch5<2> : <1>Z_Mul_D7;             {Connect switch to}
CONNECT Switch5<3> : <1>Z_Mul_D7;             {MULC functions}
CONNECT Switch5<4> : <1>Z_Mul_D7;
CONNECT Switch5<5> : <1>Z_Mul_D7;
CONNECT Switch5<6> : <1>Z_Mul_D7;

CONNECT Z_Mul_D7<1> : <1>Z_Rot_D7;            {Connect MULC to}
                                              {rotation function}

CONNECT Z_Rot_D7<1> : <2>Switch6;             {Connect rotation}
                                              {function to other}
                                              {switch}

SEND 200 TO <2>Z_Mul_D7;                      {Prime MULC function}
```

{CODE FOR DIAL 8}

```
Switch7 := F:CROUTE(6);                       {Instance switch}
                                              {function}

Rot_Lt_Elbow:= F:DXROTATE;                    {Instance rotate}
Rot_Lt_Knee:= F:DXROTATE;                     {functions}

CONNECT FKEYS<1> : <1>Switch7;                {Connect FKEYS and}
                                              {DIALS}

CONNECT DIALS<8> : <2>Switch7;

CONNECT Switch 7<3> : <1>Rot_Lt_Elbow         {Connect switch to}
                                              {rotate functions}
```

```
CONNECT Switch7<5> : <1>Rot_Lt_Knee;

CONNECT Rot_Lt_Elbow<1> : <1>Left_Forearm.Rot;      {Connect rotate}
CONNECT Rot_Lt_Knee<1> : <1>Left_Lower_Leg.Rot;     {function to display}
                                                    {tree node}

SEND 0 TO <2>Rot_Lt_Elbow;                          {Prime rotate functions}
SEND 0 TO <2>Rot_Lt_Knee;
SEND 200 TO <3>Rot_Lt_Elbow;
SEND 200 TO <3>Rot_Lt_Knee;
```

The above includes all the necessary code for a function network which will manipulate Robot. However, there is one other function you could add so that you can interactively reset Robot to its original position, before any transformations were applied, at any time. Connecting an F:XROTATE function to the F:CMUL (rotation accumulator) functions will do this (see Figure 14).

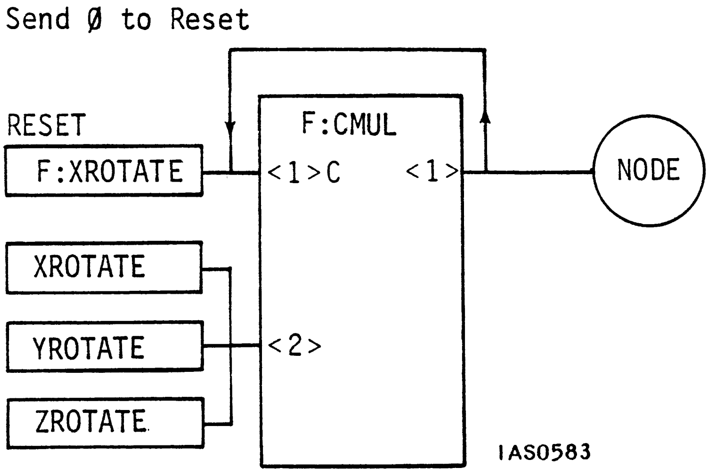


Figure 14. RESET Function Network

Add the following code:

```
Reset := F:XROTATE;

CONNECT Reset<1> : <1>Acc_Rot_Robot;
CONNECT Reset<1> : <1>Acc_Rot_Head;
CONNECT Reset<1> : <1>Acc_Rt_Arm;
CONNECT Reset<1> : <1>Acc_Rt_Hand;
CONNECT Reset<1> : <1>Acc_Rt_Leg;
CONNECT Reset<1> : <1>Acc_Rt_Foot;
CONNECT Reset<1> : <1>Acc_Rot_Trunk;
CONNECT Reset<1> : <1>Acc_Lt_Arm;
CONNECT Reset<1> : <1>Acc_Lt_Hand;
CONNECT Reset<1> : <1>Acc_Lt_Leg;
CONNECT Reset<1> : <1>Acc_Lt_Foot;
```

This will reset the network value but not the robot's display nodes. The nodes will be reset once the dials are moved again. To reset the display nodes at the same time as you reset the network, also connect this reset function to all of the rotation nodes in the display tree:

```
CONNECT Reset<1> : <1> Robot.Rot;
CONNECT Reset<1> : <1> Head.Rot;
CONNECT Reset<1> : <1> Upper_Body.Rot;
CONNECT Reset<1> : <1> Right_Arm.Rot;
CONNECT Reset<1> : <1> Left_Arm.Rot;
CONNECT Reset<1> : <1> Right_Hand.Rot;
CONNECT Reset<1> : <1> Left_Hand.Rot;
CONNECT Reset<1> : <1> Left_Leg.Rot;
CONNECT Reset<1> : <1> Right_Leg.Rot;
CONNECT Reset<1> : <1> Left_Foot.Rot;
CONNECT Reset<1> : <1> Right_Foot.Rot;
CONNECT Reset<1> : <1> Right_Forearm.Rot;
CONNECT Reset<1> : <1> Left_Forearm.Rot;
CONNECT Reset<1> : <1> Left_Lower_Leg.Rot;
CONNECT Reset<1> : <1> Right_Lower_Leg.Rot;
```

To RESET Robot, then, simply enter:

```
    SEND 0 TO <1>RESET;
```

LABELLING THE CONTROL DIALS

The function network that labels the dials also involves routing, except that the network's output will be routed to function instances associated with the control dial labels instead of into display tree nodes.

The "Function Summary" explains that there are eight DLABEL function instances, one for each dial, named DLABEL1...DLABEL8 (see Figure 15).

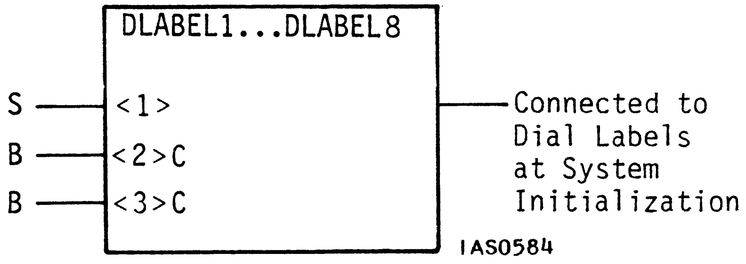


Figure 15. DLABEL Function

If you send the string of characters you want to appear in a dial's label to input <1> of a DLABEL function, the string will appear in the LEDs above the dial. (The second and third DLABEL inputs, not used in this example, allow you to blink the label or left-justify it. The default is non-blinking and centered in the available space above each dial.)

These character strings should be no more than 8 characters long. No connections need to be made <u>out</u> of DLABEL function instances; their "outputs" are the LEDs on the control dials box.

To build a function network using these functions, first determine what type of output the network needs to produce; that is, what sort of values a DLABEL function will accept. In this case, it is a <u>string</u> of characters. These strings need to be sent to the DLABEL functions. Each time you change modes, you will want a new set of LED labels to appear that correspond to the new operations handled by the dials.

Begin with the first mode. Here, seven dials control overall movements for the robot. Though the eighth dial is not labeled, a blank string is needed for the eighth label to erase any existing labels above Dial 8 which appear in other modes.

The following are suggested labels that might appear in the dial LEDs during Mode 1:

1--XRot_Bod
2--XRot_Hd
3--XR_Arm
4--XR_Hand
5--XR_Leg
6--XR_Foot

Once you identify labels to be sent to the LEDs, an efficient way to send them is to use an instance of F:INPUTS_CHOOSE(n) (Figure 16) for each DLABEL function.

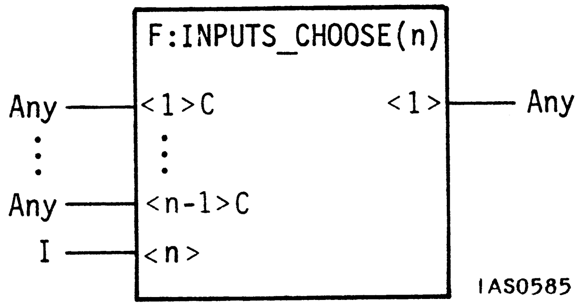


Figure 16. F:INPUTS_CHOOSE(n) Function

Make n one number larger than the number of modes you need. With six modes, use an instance of F:INPUTS_CHOOSE(7).

This function can house six different labels on its first six inputs, one for each mode. The seventh input is the "routing signal." An integer on input <7> indicates which of the labels to send out. Connect FKEYS to that input.

Now when you press a function key, FKEYS not only switches the dials into a different mode, it switches labels for the dials.

Figure 17 illustrates the network for Dial 1, with string outputs to DLABEL1 and integer inputs from FKEYS.

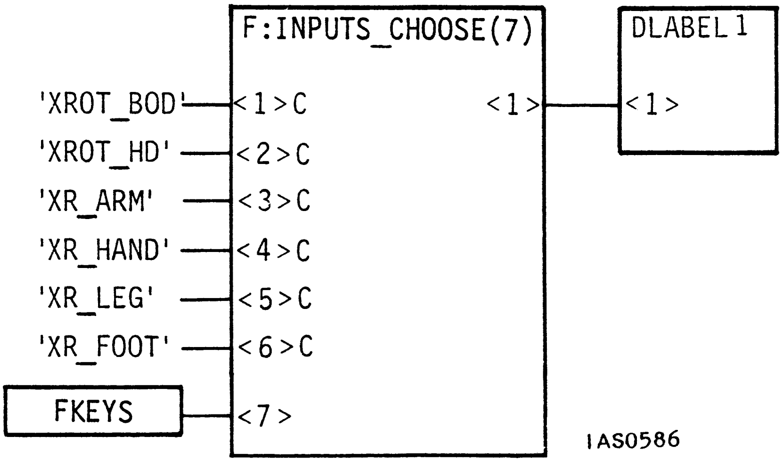


Figure 17. LED Labels for Dial 1

Exercise

The above diagram suggests how an instance of F:INPUTS_CHOOSE(7) can handle the labels for Dial 1 in all modes. Design a network with additional instances of F:INPUTS_CHOOSE that will handle the other DLABELS2 through DLABELS8. Design labels for the dials in each mode that use 8 or fewer characters to describe the dials' functions.

Figure 18 illustrates the rest of the function network needed to label LEDs. Following that is the code needed to implement the complete network.

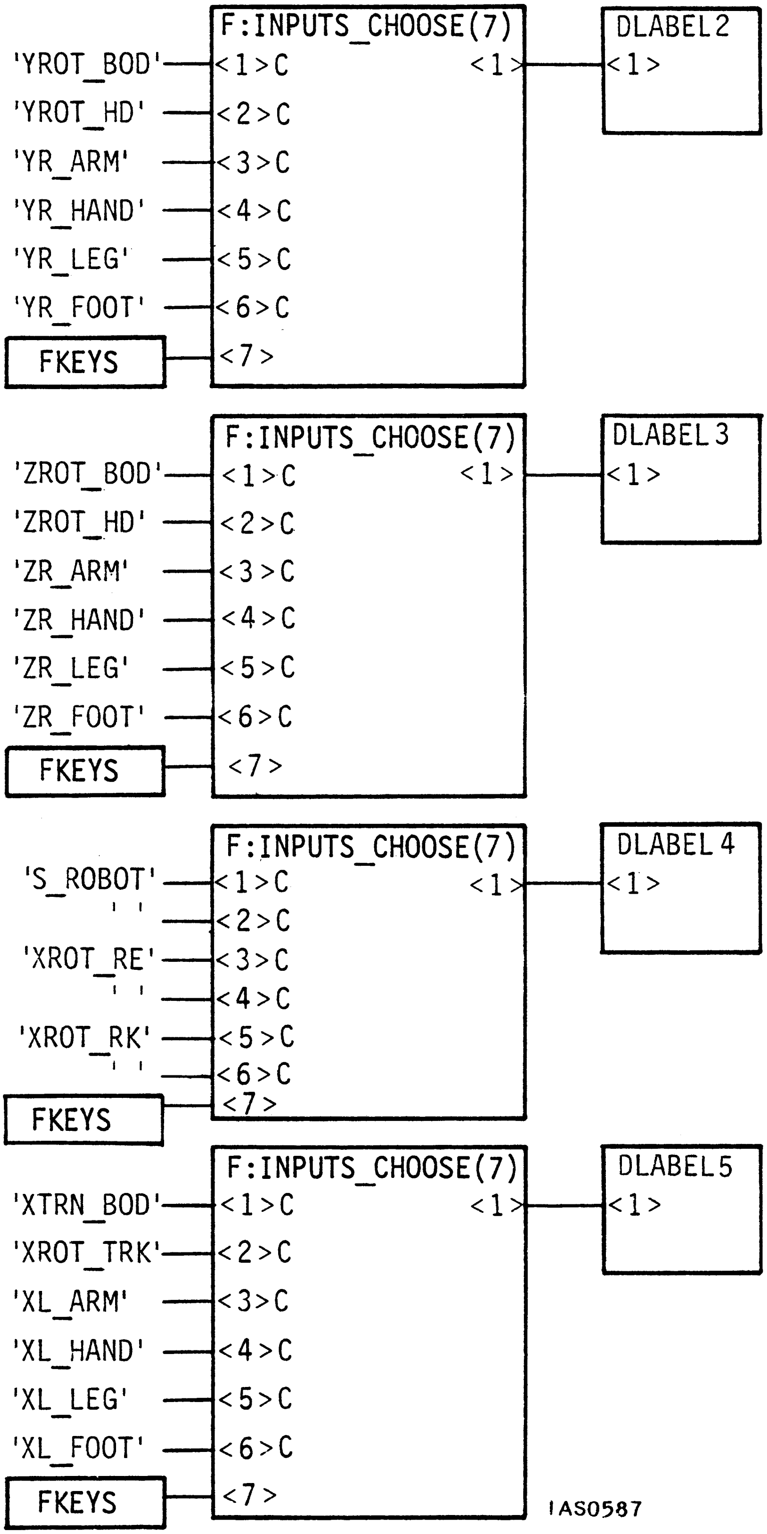


Figure 18. LED Labels for Dials 2-8

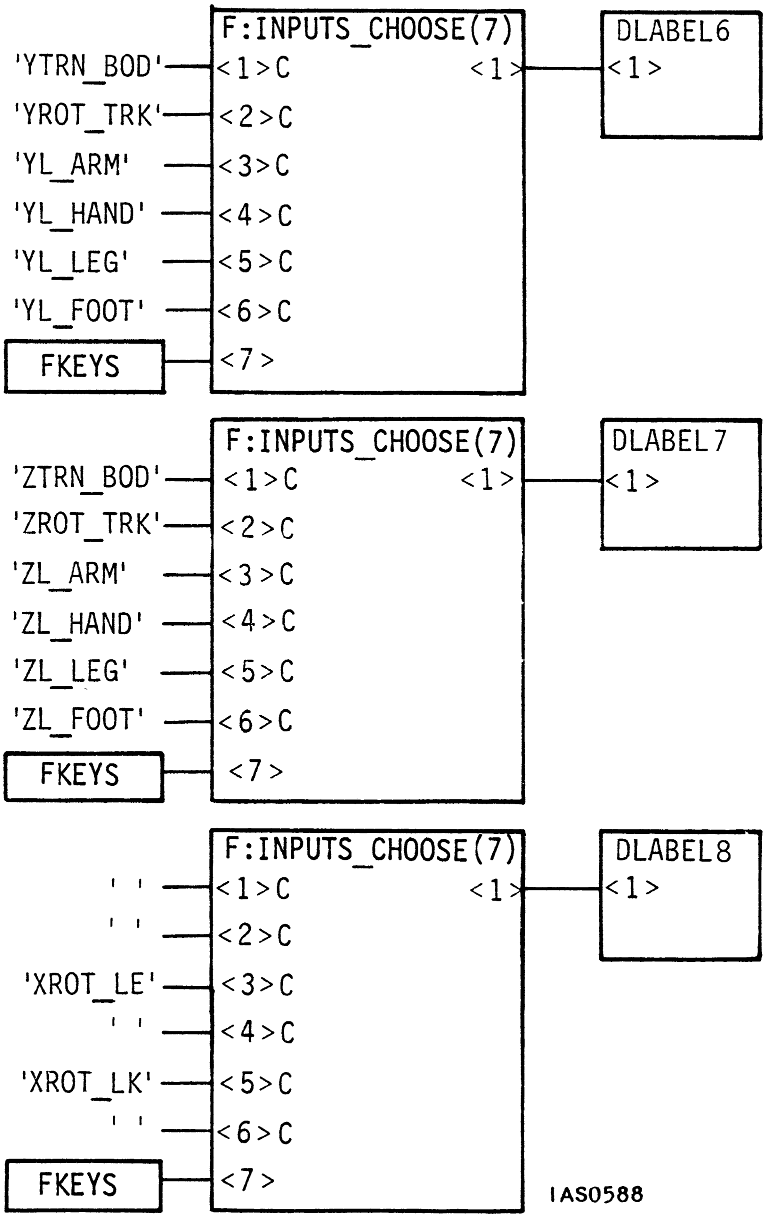


Figure 18. LED Labels for Dials 2-8 (continued)

The code follows for the eight labels in all six possible modes. Note that the DLABELS function does not have to be instanced by the user.

```
D1_Leds := F:INPUTS_CHOOSE(7);
D2_Leds := F:INPUTS_CHOOSE(7);
D3_Leds := F:INPUTS_CHOOSE(7);
D4_Leds := F:INPUTS_CHOOSE(7);            {Instance the switch function}
D5_Leds := F:INPUTS_CHOOSE(7);
D6_Leds := F:INPUTS_CHOOSE(7);
D7_Leds := F:INPUTS_CHOOSE(7);
D8_Leds := F:INPUTS_CHOOSE(7);

CONNECT FKEYS<1>:<7>D1_Leds;
CONNECT FKEYS<1>:<7>D2_Leds;
CONNECT FKEYS<1>:<7>D3_Leds;
CONNECT FKEYS<1>:<7>D4_Leds;              {Connect FKEYS to switch}
CONNECT FKEYS<1>:<7>D5_Leds;
CONNECT FKEYS<1>:<7>D6_Leds;
CONNECT FKEYS<1>:<7>D7_Leds;
CONNECT FKEYS<1>:<7>D8_Leds;

CONNECT D1_Leds<1>:<1>Dlabel1;
CONNECT D2_Leds<1>:<1>Dlabel2;
CONNECT D3_Leds<1>:<1>Dlabel3;
CONNECT D4_Leds<1>:<1>Dlabel4;            {Connect switch to LEDs}
CONNECT D5_Leds<1>:<1>Dlabel5;
CONNECT D6_Leds<1>:<1>Dlabel6;
CONNECT D7_Leds<1>:<1>Dlabel7;
CONNECT D8_Leds<1>:<1>Dlabel8;


SEND 'XRot_BOD' TO <1>D1_Leds;            {Send characters}
SEND 'XRot_HD' TO <2>D1_Leds;
SEND 'XR_ARM' TO <3>D1_Leds;
SEND 'XR_HAND' TO <4>D1_Leds;
SEND 'XR_LEG' TO <5>D1_Leds;
SEND 'XR_FOOT' TO <6>D1_Leds;

SEND 'YRot_BOD' TO <1>D2_Leds;
SEND 'YRot_HD' TO <2>D2_Leds;
SEND 'YR_ARM' TO <3>D2_Leds;
SEND 'YR_HAND' TO <4>D2_Leds;
SEND 'YR_LEG' TO <5>D2_Leds;
SEND 'YR_FOOT' TO <6>D2_Leds;
SEND 'ZRot_BOD' TO <1>D3_Leds;
SEND 'ZRot_HD' TO <2>D3_Leds;
SEND 'ZR_ARM' TO <3>D3_Leds;
SEND 'ZR_HAND' TO <4>D3_Leds;
SEND 'ZR_LEG' TO <5>D3_Leds;
SEND 'ZR_FOOT' TO <6>D3_Leds;

SEND 'S_Robot' TO <1>D4_Leds;
SEND ' ' TO <2>D4_Leds;
SEND 'XRot_RE' TO <3>D4_Leds;
SEND ' ' TO <4>D4_Leds;
SEND 'XRot_RK' TO <5>D4_Leds;
SEND ' ' TO <6>D4_Leds;

SEND 'XTRN_BOD' TO <1>D5_Leds;
SEND 'XRot_TRK' TO <2>D5_Leds;
SEND 'XL_ARM' TO <3>D5_Leds;
SEND 'XL_HAND' TO <4>D5_Leds;
SEND 'XL_LEG' TO <5>D5_Leds;
SEND 'XL_FOOT' TO <6>D5_Leds;

SEND 'YTRN_BOD' TO <1>D6_Leds;
SEND 'YRot_TRK' TO <2>D6_Leds;
SEND 'YL_ARM' TO <3>D6_Leds;
SEND 'YL_HAND' TO <4>D6_Leds;
SEND 'YL_LEG' TO <5>D6_Leds;
SEND 'YL_FOOT' TO <6>D6_Leds;

SEND 'ZTRN_BOD' TO <1>D7_Leds;
SEND 'ZRot_TRK' TO <2>D7_Leds;
SEND 'ZL_ARM' TO <3>D7_Leds;
SEND 'ZL_HAND' TO <4>D7_Leds;
SEND 'ZL_LEG' TO <5>D7_Leds;
SEND 'ZL_FOOT' TO <6>D7_Leds;

SEND ' ' TO <1>D8_Leds;
SEND ' ' TO <2>D8_Leds;
SEND 'XRot_LE' TO <3>D8_Leds;
SEND ' ' TO <4>D8_Leds;
SEND 'XRot_LK' TO <5>D8_Leds;
SEND ' ' TO <6>D8_Leds;
```

SETTING LIMITS ON THE MOTION OF A MODEL

As the robot model now operates, its movements are unbounded: it can continue bending its knees until they pass through its thigh and return to initial position. This section demonstrates how to set a limit on that motion, so that a model will more realistically imitate the movements of the object it represents.

The robot's knees provide a good illustration of how to do this. First, think of how a real leg bends (Figure 19).

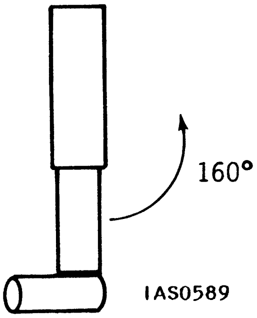


Figure 19. Realistic Limitations of Leg Movement

In a real leg, little or no forward bending is possible, but backward bending, through nearly 180 degrees is. If you set a limit at 160 degrees, it would be fairly realistic. Figure 20 shows how 160 degrees of "backward" movement in a real leg corresponds to the rotation values in the robot's knee.

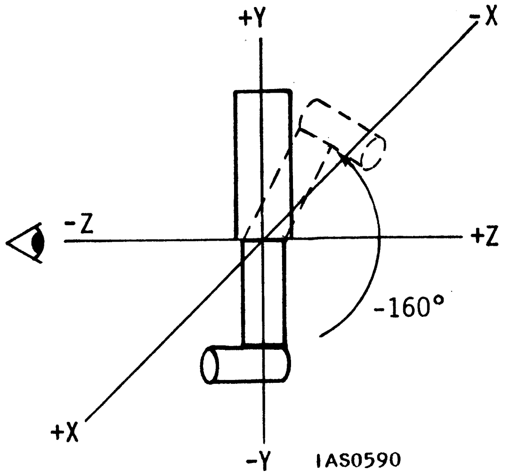


Figure 20. Limits for the Robot Leg

The rotations applied to it move it only around the X axis. Viewed from the positive X axis (the way it is in the diagram above), the "backward" rotation is counterclockwise. So the limits you want to impose are: no positive rotation in X at all, and only up to 160 in negative X.

You can modify the rotation network in the function network diagram for the robot. This requires the F:LIMIT function (see Figure 21). F:LIMIT will monitor values for degrees of rotation for the ROTATE functions and pass through only values between 0 and -160.

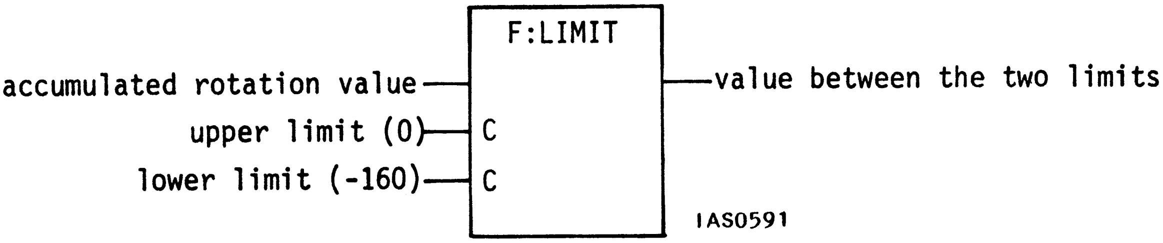
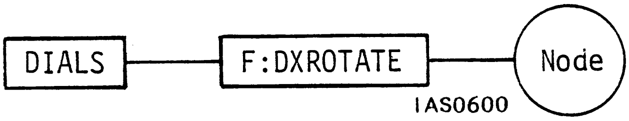


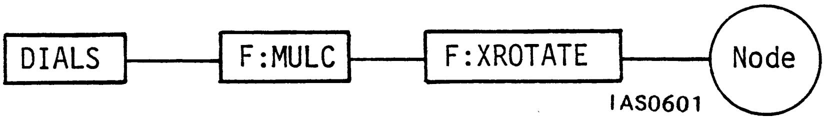
Figure 21. F:LIMIT Function

In this example, any value larger than 0 will cause F:LIMIT to send out a 0; anything less than -160 will output -160.

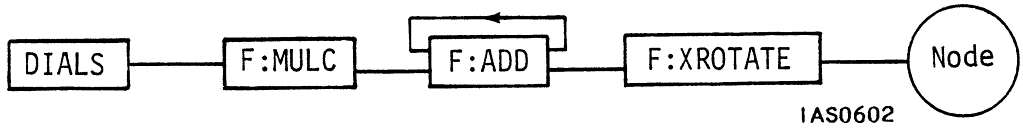
The network for robot's knees use F:DXROTATE functions because they require rotations only in X. However, the accumulator is built into F:DXROTATE, so you cannot tap into it for the input to F:LIMIT.



To use F:LIMIT, begin with an XROT network such as the one used in "Function Networks I":



Then modify it to accumulate rotation values with an add function:



Finally, add the F:LIMIT function. With this network, a stream of values from ADD (accumulated rotation values) can be output to F:LIMIT as shown in Figure 22.

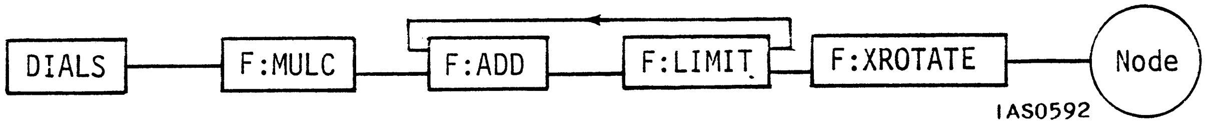


Figure 22. Function Network to Limit Movement

Though this network is bulkier (three functions now replace one), it allows you to limit the motion in the knee joint.

Exercise

Figure 23 illustrates two modified function networks that will limit rotations in both of the robot knees. Function instance names have been provided. Edit the existing code for Robot to incorporate these changes. Do not repeat any existing commands which create function instances; otherwise, all connections established by the original command are broken.

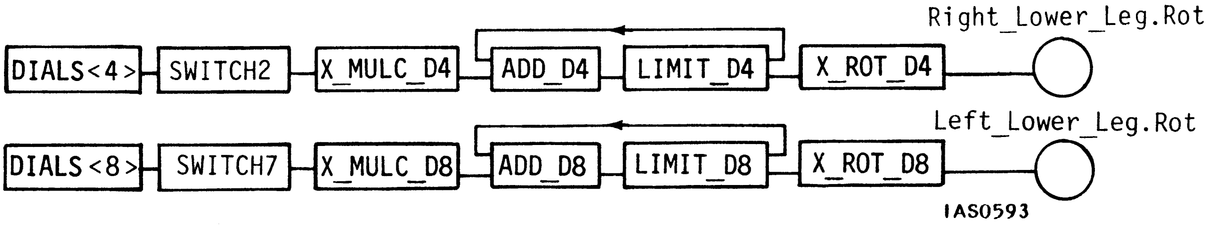


Figure 23. Function Networks to Limit the Robot Knee Movement

```
X_Mulc_D4 := F:MULC;
X_Mulc_D8 := F:MULC;
Add_D4 := F:ADD;
Add_D8 := F:ADD;                              {Instancing new functions}
Limit_D4 := F:LIMIT;
Limit_D8 := F:LIMIT;
X_Rot_D4 := F:XROTATE;
X_Rot_D8 := F:XROTATE;

DISCONNECT Switch2<5>:<1>Rot_Rt_Knee;
DISCONNECT Switch7<5>:<1>Rot_Lt_Knee;

CONNECT Switch2<5>:<1>X_Mulc_D4;
CONNECT Switch7<5>:<1>X_Mulc_D8;

CONNECT X_Mulc_D4<1>:<1>Add_D4;
CONNECT X_Mulc_D8<1>:<1>Add_D8;

CONNECT Add_D4<1>:<1>Limit_D4;                {Creating new network}
CONNECT Add_D8<1>:<1>Limit_D8;

CONNECT Limit_D4<1>:<2>Add_D4;
CONNECT Limit_D4<1>:<1>X_Rot_D4;
CONNECT Limit_D8<1>:<2>Add_D8;
CONNECT Limit_D8<1>:<1>X_Rot_D8;

CONNECT X_Rot_D4<1>:<1>Right_Lower_Leg.Rot;
CONNECT X_Rot_D8<1>:<1>Left_Lower_Leg.Rot;


SEND 200 TO <2>X_Mulc_D4;
SEND 200 TO <2>X_Mulc_D8;
SEND 0 TO <2>Limit_D4;                        {Priming functions}
SEND -160 TO <3>Limit_D4;
SEND 0 TO <2>Limit_D8;
SEND -160 TO <3>Limit_D8;

SEND 0 to <2>Add_D4;
SEND 0 to <2>Add_D8;
```

The next logical step would be to limit rotations in ALL of the robot's joints. However, this is no trivial matter. The other rotate nodes accept three-dimensional rotations which are all accumulated using matrices. Matrices cannot go through an F:LIMIT function. This problem is not insurmountable, but solutions can be complex. (For example, you could have three rotation nodes, each limiting movement using the F:LIMIT function.)

USING VARIABLES TO STORE VALUES

One difference between programming with PS 300 function networks and programming a conventional language such as FORTRAN is that you almost never need to use variables. In a conventional program, you may represent two values to be added together as variables X and Y. In a function network, you would add these using an ADD function. The "variables" are the function's two inputs.

Sometimes, though, you may want to use a variable value in a function network in a more conventional way. Often, this can be done using a F:CONSTANT function (see Figure 24).

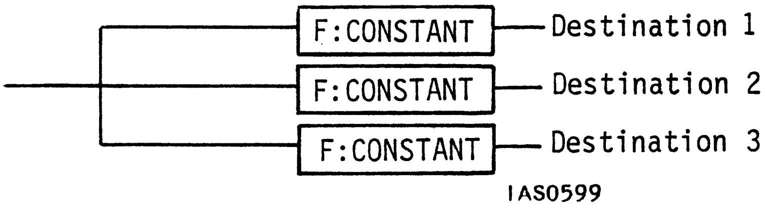
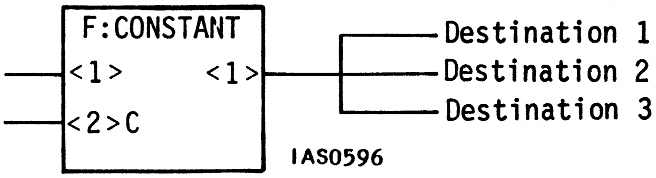


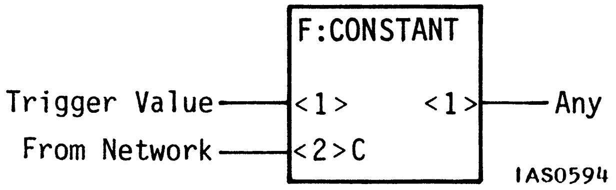
Figure 24. F:CONSTANT Function

In this setup, the value you want to save is sent to the constant input of the function. If you send a stream of values, each one will over-write the preceding one, so the value on the constant input will always be current (the latest one sent). When you need the variable somewhere else in the network, send any value to trigger F:CONSTANT's input <1> and the value will fire out to wherever you connect the output.

It may be the case, however, that several areas in a network need to access the variable in an F:CONSTANT function. You might think that can be done by making numerous output connections to all the destinations that may use the variable.



However, this presents a problem of routing and selection. To send the variable value to destination 1, you must trigger F:CONSTANT, which sends out values to all destinations. One solution to this problem could be to use more instances of F:CONSTANT.



A more efficient solution is to use the VARIABLE command in conjunction with the command STORE and the function F:FETCH. This section discusses how to do that.

The VARIABLE command creates a "holding tank" for a single value, much the same way the constant input of F:CONSTANT does. Look at the following command:

```
VARIABLE This, That, The_Other;
```

This command creates three variables named This, That, and The_Other. Variables have only one input and no outputs. Function networks can be connected to them or they can receive values by means of the SEND command:

```
CONNECT Spinner<1>:<1>This;

SEND 4.5 TO <1>This;
```

If a network is connected to a variable, it can receive a stream of values and will retain the last one sent.

An alternate way to send a value to a variable is to use the STORE command. The following commands both do the same thing:

```
STORE 4.5 IN This;

SEND 4.5 TO <1>This;
```

There are two ways to retrieve a value stored in a variable: using the SEND VALUE command or using a function network with F:FETCH. For example, if you want to send a value from the variable "This" to the third input of a function named ROT_X, you could enter:

```
SEND VALUE(This) TO <3>Rot_X;
```

Even more convenient is using F:FETCH (Figure 25).

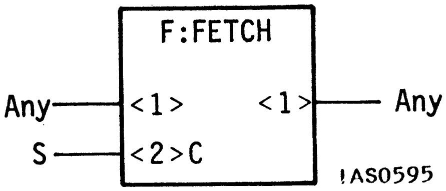


Figure 25. F:FETCH Function

F:FETCH accepts the name of the variable on its constant input (input 2). When any value arrives on input 1, the function is triggered. It fetches the latest value from that variable and sends it out.

For example, in Figure 26 below, values for the variable "This" are routed to the host using the F:FETCH function. (User-assigned names are written above the function box.)

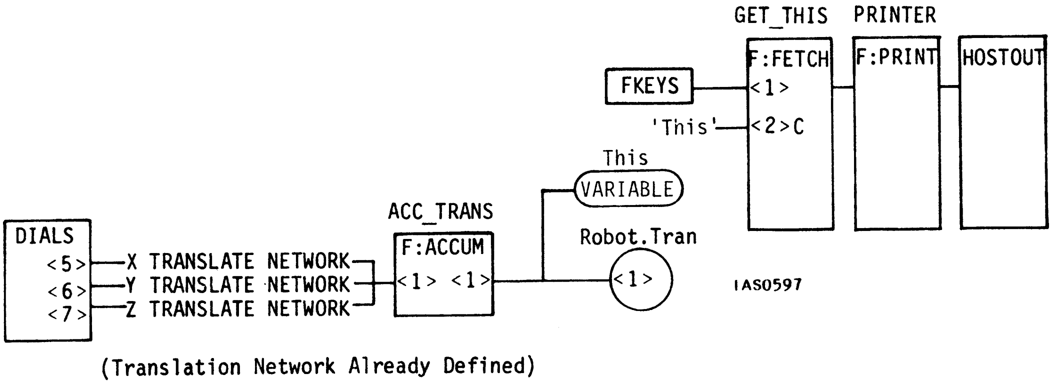


Figure 26. Routing Values From THIS Variable to the Host

The variable This holds a 2D vector that indicates the accumulated translation values sent out from ACC_TRANS in Mode 1. (The translation network has already been defined and coded in the Robot code.)

HOSTOUT has one input, which accepts a string and routes it to the host. HOSTOUT is preceded by a function that turns PS 300 values into strings, F:PRINT. (If the GSRs are being used, HOST_MESSAGE should be used in lieu of HOSTOUT.)

The additional code needed for this network is:

```
VARIABLE This;

Get_This := F:FETCH;
Printer := F:PRINT;

CONNECT Acc_Trans<1>:<1>This;
CONNECT FKEYS<1>:<1>Get_This;
CONNECT Get_This<1>: <1>Printer;
CONNECT Printer<1>:<1>Hostout

SEND 'This' to <2> Get_This;
```

Exercise

Using Figure 26 as a pattern, create a function network that uses a variable named MATRIX which holds the most current rotation matrix from F:CMUL for the robot's left arm (ACC_LT_ARM) in Mode 3. Retrieve this value and send it to HOSTOUT using an instance of F:FETCH named Retrieve. Specify any additional code needed (the rotation network for Robot has already been done).

Figure 27 illustrates the function network which retrieves values from the variable MATRIX.

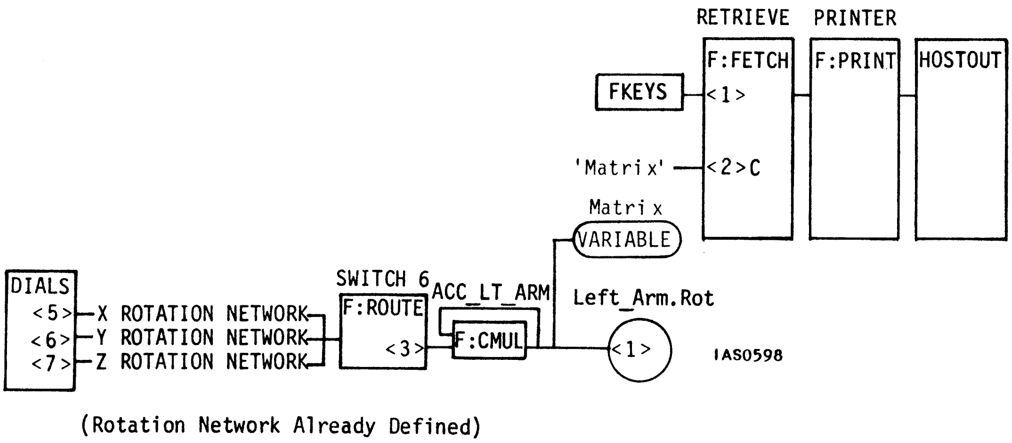


Figure 27. Routing Values From MATRIX Variable to the Host

The additional code needed for this network is:

```
VARIABLE Matrix;

Retrieve := F:FETCH;
Printer := F:PRINT;

CONNECT Acc_Lt_Arm<1>:<1>Matrix;
CONNECT FKEYS<1>:<1>RETRIEVE;
CONNECT Retrieve<1>: <1>Printer;
CONNECT Printer<1>:<1>Hostout;

SEND 'Matrix' to <2> RETRIEVE;
```

SUMMARY

This module illustrates how to expand a function network so that a single dial can manipulate several movements of a model. This entails determining the number of dials needed for interactions in the model and assigning each dial several destinations (in this module, interactive nodes in the model's display tree or LED labels).

Function keys and instances of F:CROUTE(n) are used to switch values from the dials to their various destinations. This prevents dial values from being routed to all function network destinations at once.

Specifically, the initial function instance FKEYS is connected to input<1> of the switching function F:CROUTE(n). Incoming values from the dials are connected to input <2>. The outputs of F:CROUTE(n) are connected to the various destinations.

LEDs above the dials are labeled in each mode of operation. Specifically, labels in every mode for that dial are sent to the constant inputs of F:INPUTS_CHOOSE. FKEYS is connected to the last input of this function. The output of F:INPUTS_CHOOSE is connected to the DLABEL function associated with that dial. When the function key is pressed, to switch modes, the correct label for the dial in that mode is routed to DLABEL, which outputs to the LEDs.

Functions can serve more than one purpose. For example, in addition to controlling X rotations, the F:XROTATE function can be used to reset the model back to its original position before any transformations were applied.

The F:LIMIT function can be inserted into a network to set limits on a model's movement. F:LIMIT requires that you establish upper and lower limits for transformation values. It then passes through only those values which lie within this range.

Finally, the VARIABLE command and F:FETCH functions allow you to store and retrieve a variable value in a function network.

TEXT MODELING AND STRING HANDLING

CONTENTS

TEXT MODELING AND STRING HANDLING

ILLUSTRATIONS

Text is handled by the PS 300 in the same way as any other graphical item. Characters are defined as data nodes consisting of a single string (a CHARACTERS node) or a block of several strings or *labels* (a LABELS node). Just like other graphical items, characters can be transformed through matrices. Because they are affected by 3X3 matrices, they can be transformed along with any three-dimensional object which includes them in its definition. Characters can also be rotated and scaled using commands that create 2X2 transformation matrices. These matrices transform text while leaving other 2D and 3D graphical data unaffected.

Strings can be created and manipulated with commands. They can also be manipulated interactively using function networks and interactive devices.

A standard character font comes with the PS 300. Commands exist which allow you to design and use an unlimited number of alternate character fonts. A graphical character font editor program, MAKEFONT, is also available for designing and modifying character fonts. Refer to Volume 4 for information about this program.

Text and text-handling nodes are included in display trees. Text strings are data nodes and text transformations are operation nodes. The current character font is an attribute node which points to a look-up table for the vectors which comprise the font in current use.

OBJECTIVES

In this module you will learn how to:

- Use commands to create character strings.
- Use commands to manipulate character strings.
- Use functions to manipulate characters and strings.
- Update characters and labels nodes.
- Create and use different character fonts.

PREREQUISITES

Be at a PS 300 and have access to the Tutorial Demonstration programs. Be familiar with the concepts covered in "Graphics Principles" and in the "Modeling" and "PS 300 Command Language" modules. Also have at hand the ***Command Summary*** and ***Function Summary*** in Volume 3A for reference to the commands and functions you will be using.

Be sure that you have read ***User Operation and Communications*** in Volume 1 so that you know how to put the PS 300 into and out of Command mode.

USING COMMANDS TO CREATE CHARACTER STRINGS

Two PS 300 commands create character strings: the CHARACTERS command and the LABELS command.

The CHARACTERS Command

The CHARACTERS command lets you create a single string of up to 240 characters and specify the location of that string in the world coordinate system.

The simplest form of the command lets you create a string which starts at the origin (the default location). The following command assigns the name String to a character string. Put the PS 300 in Command Mode by pressing the CONTROL and LINE LOCAL keys, and enter this command.

```
String := CHARACTERS 'The quality of mercy...';
```

Now DISPLAY String. All you can see at the moment is a large "T" in the top-right quadrant and the vertical stroke of the "h". This is because each character is defined in a square which, by default, is one unit on each side. The default starting point for any string is the origin. Since the default window is from -1 to 1 in X and Y, only the first letter is within the window. Figure 1 illustrates this.

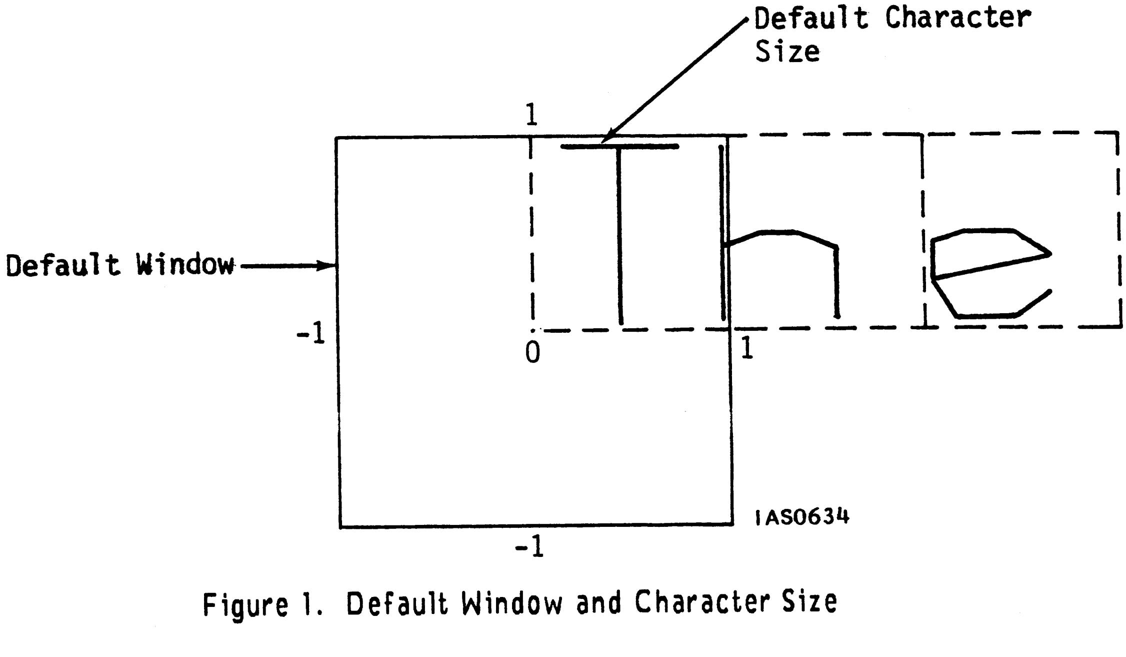


Figure 1. Default Window and Character Size

To scale the characters to fit the default window and display the string at its new size, enter the following commands.

```
Scale_String := SCALE BY .04 APPLIED TO String;
REMOVE String;
DISPLAY Scale_String;
```

The string should now appear in much smaller letters beginning at the center of the screen. Notice that the characters which form the string in the CHARACTERS command are enclosed in single quotation marks; however, when String is displayed, only the characters appear. If you want quotation marks in the text string, you must use three single quotation marks at the start and at the end of the string. Redefine String by entering the following command.

```
String := CHARACTERS '''The quality of mercy...''';
```

The character string should now appear in single quotation marks.

To get a single quote to appear in a string (as an apostrophe, for example) you must enter two single quotes. Redefine String with the following command.

```
String := CHARACTERS 'Love''s not time''s fool';
```

The string should appear with the contraction ***Love's*** and the possessive ***time's***.

Changing Starting Position and Spacing

When the PS 300 displays a character string, the string is positioned by default with the lower-left corner of the unit square enclosing the first character at the origin of the world coordinate system. Characters are regularly spaced and follow each other horizontally. Optional parameters in the command let you specify the beginning coordinates of the string and change the horizontal and vertical spacing of the characters to create vertical and diagonal text strings.

Enter the following command to redefine String as a new line of text positioned off the origin.

```
String := CHARACTERS 0,5,0 'Up a little';
```

This string starts at 0 on the X axis and 0 on the Z axis but 5 on the Y axis. The X,Y,Z coordinate of the starting point can always be specified in this way. The Z coordinate is optional and, if not supplied, defaults to zero.

The spacing between characters can be changed with a STEP clause. This clause lets you specify the spacing between characters in X and Y as a value from -1 to 1. The default spacing is 1,0 or one unit in X and zero in Y for regular horizontal spacing.

The vertical spacing can be changed by specifying the Y component of the STEP clause as a value other than zero. Enter the following command to create a string which descends diagonally from the origin to the right.

```
String := CHARACTERS STEP 1,-1 'Stepping down';
```

Now redefine the string as a diagonal which ascends from the origin to the upper-right.

```
String := CHARACTERS STEP 1,1 'Stepping up';
```

Exercise

Try different combinations of X and Y values to produce strings which descend and ascend vertically from the origin.

The LABELS Command

The LABELS command, like CHARACTERS, defines character strings for display. Whereas CHARACTERS defines a single string, LABELS combines any number of character strings into a single block. Each character string in the block is called a *label*.

The command is quite straightforward to use. The following example combines some of the text strings created earlier in this module into a single label block.

```
String := LABELS   0,0  'The quality of mercy...'
                  -1,2  '''The quality of mercy...'''
                   4,5  'Up a little'
                   2,-5 'Love''s not time''s fool';
```

Diagonal and vertical strings could not be included in the block, however, because they specify different horizontal and vertical spacing between characters. The LABELS command is not able to accommodate this. The only clause in the command is the X,Y,Z coordinate of each label in the block.

When to Use CHARACTERS and LABELS

Both the CHARACTERS and the LABELS commands create data nodes in a display tree. Whenever several character strings are defined as a single LABELS node rather than as separate CHARACTERS nodes, there is a gain in display capacity. If you are displaying a lot of text, it is best defined using the LABELS command.

Character strings defined with the CHARACTERS command, however, are more versatile. In deciding which command to use, keep the following in mind.

- The CHARACTERS command lets you change the horizontal and vertical spacing between characters. The LABELS command does not.

- If text is created using CHARACTERS, you can manipulate any character in the text string. If the LABELS command is used, the smallest entity you can manipulate is a single text string.

USING COMMANDS TO MANIPULATE CHARACTER STRINGS

The CHARACTERS and LABELS commands create data nodes containing text. Like any other primitive data, text can be transformed by having a matrix applied to it. Text can be rotated and scaled using the ROTATE and SCALE commands which transform any two-dimensional or three-dimensional structures. In addition, characters can be transformed with their own rotate and scale commands: CHARACTER ROTATE, CHARACTER SCALE, and TEXT SIZE. These commands create 2X2 transformation matrices which only operate on text.

Character Rotations

The CHARACTER ROTATE command rotates a character string or label block around the Z axis. When you look in the positive direction of the axis, the rotation is counterclockwise.

To see the effect of this command, initialize the display, then rotate and display the scaled labels block.

```
    INITIALIZE DISPLAY;
    Rot_Text := CHARACTER ROTATE 90 APPLIED TO Scale_String;
    DISPLAY Rot_Text;
```

Each string in the block should be rotated 90 degrees to the left. Notice that each label in the block is rotated around its own starting location. There is no single point in a labels block around which the whole block rotates.

A character rotate node can be updated interactively by any 2X2 matrix. The functions F:MATRIX2 and F:CROTATE (where C stands for character) are often used to supply the new matrix to the node.

Character Scales

Characters can be scaled like any other primitive data by a three-dimensional scale matrix using the SCALE command. There is also a CHARACTER SCALE command which creates a 2X2 scale matrix for transforming text only.

There are two forms of the CHARACTER SCALE command, one for uniform scaling and one for non-uniform scaling. Enter the following commands to initialize the display and to uniformly scale by .05 and then display the characters in the labels block.

```
INITIALIZE DISPLAY;
Char_Scale := CHARACTER SCALE .75 APPLIED TO Scale_String;
DISPLAY Char_Scale;
```

The scale factor is applied in both X and Y to the characters that compose scale-string. A non-uniform scale can be applied by specifying separate scale factors in X and Y. Enter the following command to redefine Char_Scale and make tall characters.

```
Char_Scale := CHARACTER SCALE .5,3 APPLIED TO Scale_String;
```

Characters in the strings are made tall and thin with this command.

When several CHARACTER SCALE commands are used, each is concatenated with the next and a cumulative scaling matrix is applied to the characters. To see this effect, initialize the display and create and display a text string called Text.

```
INITIALIZE DISPLAY;
Text := CHARACTERS 'See Spot run.';
DISPLAY Text;
```

Since the characters are at the default size, only the capital 'S' and one line of the first lowercase 'e' are visible in the top-right quadrant of the screen. Now scale the string by prefixing it with a CHARACTER SCALE node.

```
PREFIX Text WITH CHARACTER SCALE .5;
```

The characters should now change to half their previous size, and the 'S', first 'e', and one line of the second 'e' should be visible. The PREFIX command inserts a new node above the existing node and assigns the existing node's name to the new node. Figure 2 shows the effect of the PREFIX command on the display tree.

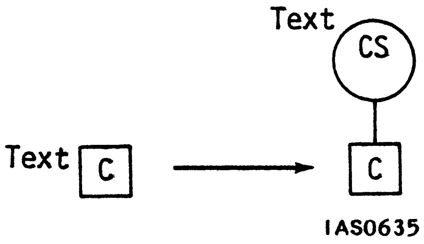


Figure 2. The Effect of the PREFIX Command

Use the PREFIX command again to create another scale node above the last one.

```
PREFIX Text WITH CHARACTER SCALE .1;
```

Notice that the size of the characters is now one tenth of what it was before, not one tenth of the original default size. The actual size of the text is .5 times .1, which is .05 of the default size. The new display tree is as shown in Figure 3.

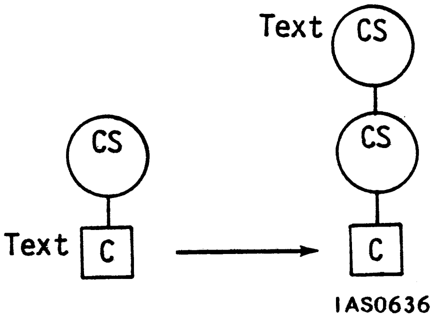


Figure 3. New Node Added With the PREFIX Command

The two character scales are concatenated and the combined scaling matrix is applied to the characters.

The TEXT SIZE Command

Character sizes can also be changed with the TEXT SIZE command. This command creates a text size which replaces the default size of 1. Text sizes are expressed as multiples or fractions of the default size.

Like the CHARACTER SCALE command, TEXT SIZE creates a 2X2 scaling matrix. However, this matrix is not concatenated with any other matrix. This means that the command creates a node which overrides any 2X2 matrix nodes above it in the same branch of the display tree.

To see the effect of the command, first remove the two CHARACTER SCALE prefixes of the string called Text, then prefix Text with a TEXT SIZE node.

```
REMOVE PREFIX OF Text;
REMOVE PREFIX OF Text;
PREFIX Text WITH TEXT SIZE .5;
```

As you remove the prefixes, the characters being displayed should get larger until they are back to the default size, and only the capital S is visible in the top-right quadrant. Prefixing with the TEXT SIZE command should make the letters half of the default size. The display tree for this structure is as shown in Figure 4.

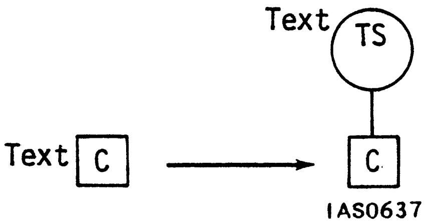


Figure 4. Display Tree With TEXT SIZE Node

Now prefix Text with a CHARACTER SCALE node to scale the characters by half again.

```
PREFIX Text WITH CHARACTER SCALE .5;
```

The text size does not change. This is because the effect of the CHARACTER SCALE node is overridden by the TEXT SIZE node below it in the structure. The display tree for the structure is shown in Figure 5.

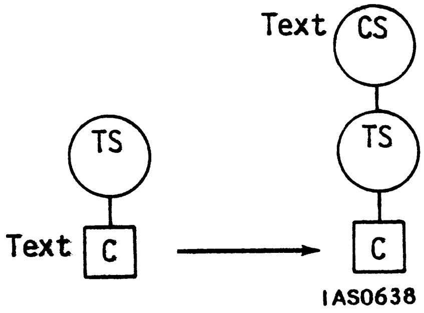


Figure 5. TEXT SIZE Node Prefixed With CHARACTER SCALE Node

Now prefix the CHARACTER SCALE node with a character rotation node.

```
PREFIX Text WITH CHARACTER ROTATE 90;
```

Again, nothing happens. The TEXT SIZE node overrides <u>all</u> 2X2 matrices above it. Since a CHARACTER ROTATE node is a 2X2 matrix node, it too is cancelled out like the character scale. You should take this into account when structuring data.

Exercise

The TEXT SIZE node has no effect on 3X3 matrices, however. Try replacing the CHARACTER ROTATE node with a ROTATE node, and the rotation will be applied.

Character Orientation

If a transformation is applied to an object or part of an object which contains text in its structure, the default condition is that the text will be transformed too. Consider the display tree in Figure 6.

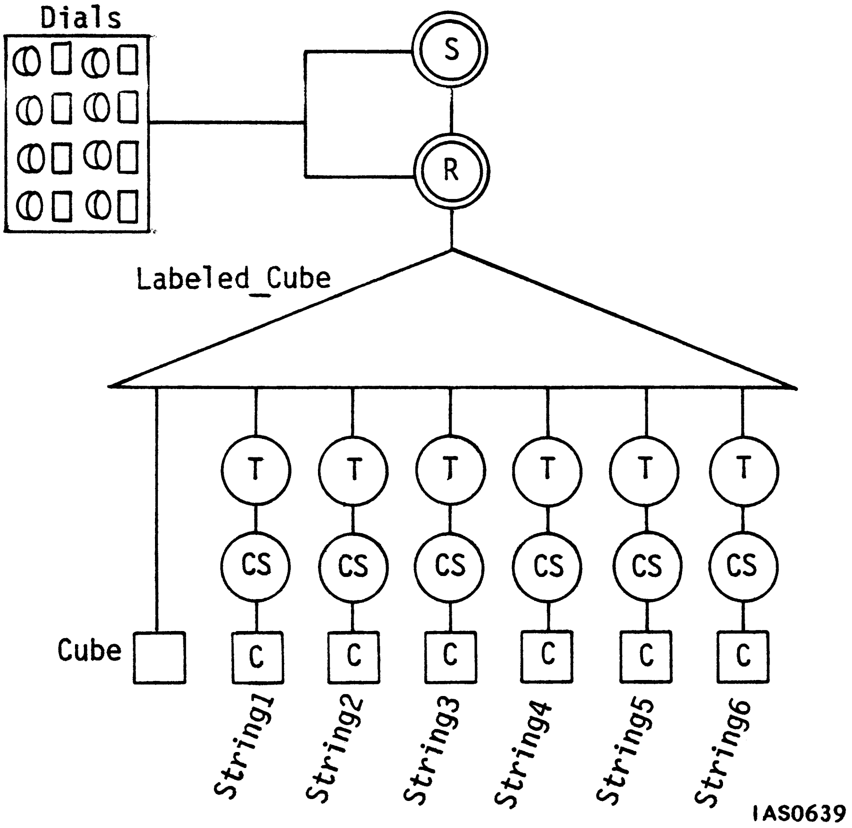


Figure 6. Display Tree for a Labeled Cube

An instance node called Labeled_Cube groups a vector list defining a cube and character strings which are scaled and positioned on each face to label the ***FRONT***, ***BACK***, ***TOP***, ***BOTTOM***, ***LEFT***, and ***RIGHT***. A rotation node connected to three dials through a function network allows Labeled_Cube to be rotated interactively. A scale node is also connected to a dial to allow interactive scaling. Any rotation or scale that is applied to the cube is also applied to the character strings.

To display the cube represented by the display tree in Figure 6, go to the Tutorial Demonstration Menu and select the program called CHARACTERS.

The cube with its faces labeled will be displayed in three viewports. The rotation node is connected to Dials 1, 2 and 3 for rotations in X, Y, and Z. Dial 4 is connected to the scale node. Use the dials to manipulate the cube.

Notice that as you rotate and scale the cube, the character strings on the faces of the cube in viewport 1 are rotated and scaled also. Depth-cueing is performed on the characters as well as on the lines that make up the cube.

As you manipulate the cube in viewport 1, the character strings which label its faces are unreadable much of the time. They may be backwards, upside-down, and too small to read. Notice that this is not the case with the characters in viewports 2 and 3. These characters are unaffected by rotations and scales while the object is being transformed. This is achieved by using the SET CHARACTERS command. This command determines the orientation of characters which are part of a model. It has an "orientation" clause with three options: WORLD_ORIENTED, SCREEN_ORIENTED, and SCREEN_ORIENTED/FIXED.

World-Oriented Characters

World-oriented characters are what you are seeing with the cube in viewport 1. The characters are transformed along with the object just like any other part of it. When an object is rotated, translated, or scaled, the characters undergo the same transformations. This is the default condition for any character string or label block you create.

The syntax for this command is as follows.

```
Name := SET CHARACTERS WORLD_ORIENTED APPLIED TO Name1;
```

Screen-Oriented Characters

Screen-oriented characters are unaffected by ROTATE and SCALE nodes. The SET CHARACTERS command can be used with the SCREEN_ORIENTED clause to maintain a readable orientation for character strings when an object is transformed. The cube in viewport 2 has a SET CHARACTERS SCREEN_ORIENTED node added. When this cube rotates, the names on the cube's faces stay readable. They rotate around the three axes but they stay parallel to the XY plane. When the cube is scaled, the character size remains unchanged.

The syntax for this form of the command is as follows.

```
Name := SET CHARACTERS SCREEN_ORIENTED APPLIED TO Name1;
```

Screen-Oriented/Fixed Characters

Notice that with the screen-oriented characters in viewport 2, the intensity of the characters varies with depth. If the cube were being displayed in perspective projection, the size of the characters would vary too. In the cube's initial position, the characters ***BACK*** on the back face of the cube would appear smaller and dimmer than the characters ***FRONT***. You can use the SCREEN_ORIENTED/FIXED option of SET CHARACTERS to fix the size and intensity at which characters are displayed.

The cube in viewport 3 has a SET CHARACTERS node with the SCREEN_ORIENTED/FIXED option. Notice that when you rotate this cube, depth-cueing is not performed on the characters, so they remain at full intensity.

The syntax for this form of the command follows.

```
Name := SET CHARACTERS SCREEN_ORIENTED/FIXED
        APPLIED TO Name1;
```

USING FUNCTIONS TO MANIPULATE CHARACTERS AND STRINGS

There are several functions which are used for manipulating characters and strings. These functions convert characters and strings to other types of data, format and reformat strings, transform characters, and perform other miscellaneous character and string-handling operations.

Complete information on these functions is contained in the *Function Summary* in Volume 3A. The following sections summarize the functions and give a few examples of their use.

Character and String Conversion Functions

F:CHARCONVERT

Converts characters to integers. The function accepts a string and converts each byte of the string (i.e., each character) to an integer. For example, the string 'AB' will be converted to 65 66, the ASCII decimal equivalent of A and B.

F:CHARMASK

Masks each character in a string by ANDing each byte with a constant integer. This is useful for converting one character or a string of characters to another, for example, from upper to lower case or from a non-printable to a printable character.

F:PRINT

Converts any data type to a string. For example, a Boolean input will generate the string 'TRUE' or 'FALSE'; a 3D vector will generate a string such as '5,2,1' and so on.

F:TRANS_STRING

Translates one string into an output string using another string as a translation table. For example, prime the function by sending 'ABCDEFGHIJKLMONPQRS TUVWXYZ' as the translation table to input <3> of the function, and 97 (the ASCII decimal equivalent of 'a') to input <2>. If a string of lowercase letters of the alphabet is now sent to input <1>, the letters will be converted to uppercase on output <1>.

F:STRING_TO_NUM

Converts a string to a real number or an integer.

F:GATHER_STRING

Collects strings until a terminator arrives. It then packages them into one string which may or may not include the terminator.

String Formatting and Reformatting Functions

F:CONCATENATE

Concatenates strings. The string on input <2> of the function is appended to the string on input <1>.

F:SPLIT

Compares two strings and splits them depending on the match. If a match occurs, characters in the string on input <1> that precede the match are output on output <1>. Matching characters are output on output <2>. Characters following the matching characters are output on output <3>. And a Boolean TRUE is output on output <4>. If no match is found, nothing is output on outputs <1>, <2>, and <3>, and a Boolean FALSE is output on output <4>.

F:PUT_STRING

Replaces characters in the string on input <1> with the string on input <3>, starting at the position specified by the integer on input <2>.

F:TAKE_STRING

Outputs a string consisting of the number of characters specified on input <3> taken from the string on input <1>, starting at the position given on input <2>.

F:LINEEDITOR

Accepts a stream of characters and simple editing commands, accumulates the characters in an internal line buffer, applies the commands to the contents of the line buffer as they are received, and outputs the edited line when a specified delimiter character is recognized.

F:LABEL

Creates a label to send to a LABELS node. A vector on input <1> of the function indicates the location of the label in the coordinate system. A string on input <2> is the text of the label. A Boolean value on input <3> indicates whether the label is to be displayed or not. The data type output by this function can only be used as input to a LABELS node.

Miscellaneous String-Handling Functions

F:LENGTH_STRING

Accepts a string and outputs its length.

F:FIND_STRING

Determines whether the string on input <2> is a substring of the string on input <1>. Outputs the starting location of the substring if it is found.

F:COMP_STRING

Compares two strings to determine if the string on input <1> is greater than, less than, or equal to the string on input <2>.

F:LBL_EXTRACT

Extracts information about a label in a LABELS node. An integer on input <1> is an index into the LABELS block. A string on input <2> is the name of the node. The function outputs the text of the label, its location in the coordinate system, and a TRUE or FALSE to indicate if the label is displayed or not.

Character Transformation Functions

F:CROTATE

Uses an integer on input <1> which represents degrees of rotation to create a 2X2 Z-axis rotation matrix. This matrix can be used to update a CHARACTER ROTATE node.

F:CSCALE

Uses a real number or a two-dimensional vector to create a uniform or non-uniform 2X2 scaling matrix. The matrix can be used to update a CHARACTER ROTATE node.

F:MATRIX2

Accepts two-dimensional vectors on inputs <1> and <2> and creates a 2X2 matrix. This matrix can be used to update a CHARACTER SCALE or CHARACTER ROTATE node.

UPDATING CHARACTERS AND LABELS NODES

Both CHARACTERS and LABELS nodes can have their contents updated using commands and functions.

Updating With Commands

The COPY and SEND commands can be used to change the contents of a CHARACTERS or LABELS node.

The COPY Command

Labels can be copied from one labels node to another using the COPY command. Note, however, that this command does not work with a CHARACTERS node. The command has the following format:

```
Name := COPY Name1 [START=] i [,] [COUNT=] n;
```

The parameters for this command are:

Name - The name of the labels node you are creating and copying into.

Name1 - The name of the labels node you are copying from.

i - The number of the first label to be copied.

n - A count of the number of labels to be copied.

The command can be used as follows. First create a labels node called Limerick.

```
Limerick := LABELS -1,.75   'What''s wrong with this PS 300?'
                   -1,.5    'The frustrated programmer thundered'
                   -1,.25   'I''ve entered commands'
                   -1,0     'With the carefulest of hands'
                   -1,-.25  'But somehow I seem to have blundered!';
```

To see the limerick, scale the labels block by .05 and display it.

```
Scale_Block := CHARACTER SCALE .05 THEN Limerick;
DISPLAY Limerick;
```

Now create a new labels block which starts at the third label and is three labels long.

```
New_Block := COPY Limerick START = 3, COUNT = 3;
```

The words START and COUNT and the equals signs are optional, so you could have typed "COPY Limerick 3,3;" instead. If one word is used, however, both must be used.

Now redefine Scale_Block so that is refers to New_Block.

```
Scale_Block := CHARACTER SCALE .05 THEN New_Block;
```

The last three lines of the Limerick should now be displayed on the screen.

The SEND Command

Several forms of the SEND command can be used to update a LABELS or CHARACTERS node. Both nodes have similar input queues. Figure 7 shows inputs to a CHARACTERS node and Figure 8 shows inputs to a LABELS node.

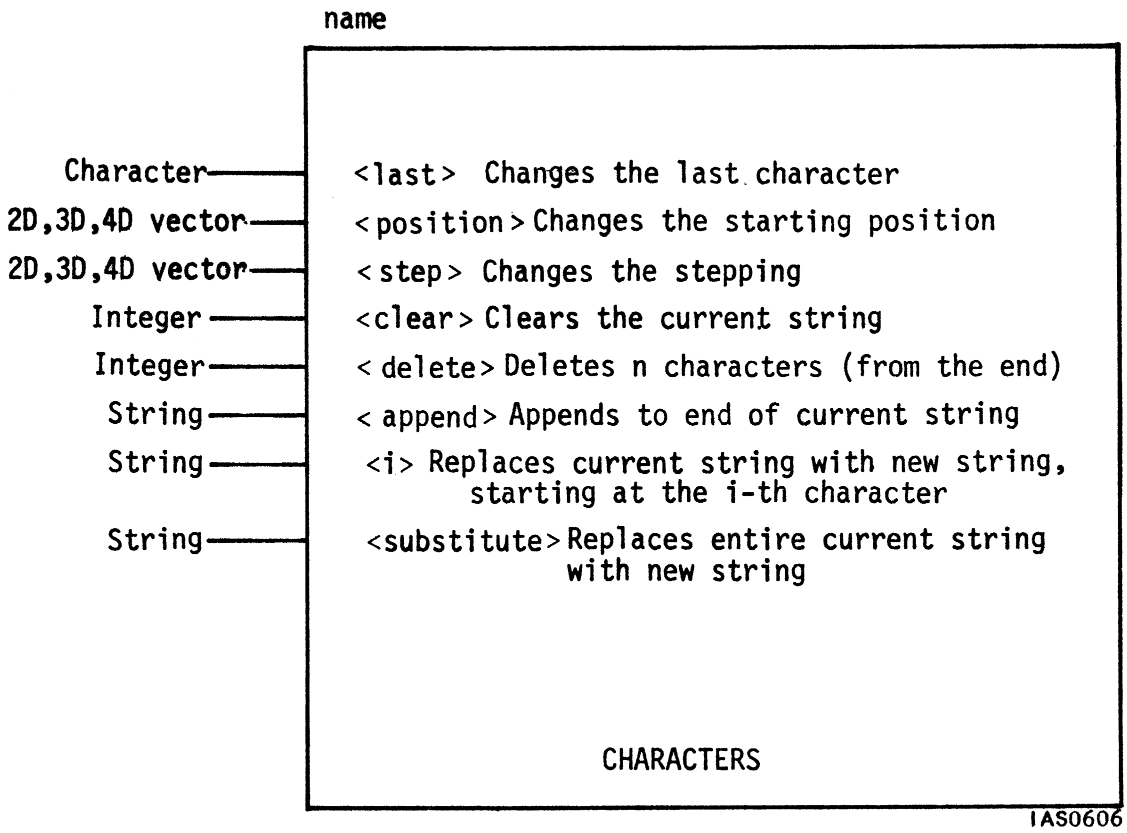

name
Character
<last> Changes the last character
2D,3D,4D vector
<position> Changes the starting position
2D,3D,4D vector
<step> Changes the stepping
Integer
<clear> Clears the current string
Integer
< delete> Deletes n characters (from the end)
String
< append> Appends to end of current string
String
<i> Replaces current string with new string, starting at the i-th character
String
<substitute> Replaces entire current string with new string
CHARACTERS
IAS0606

Figure 7. Inputs to a CHARACTERS Node

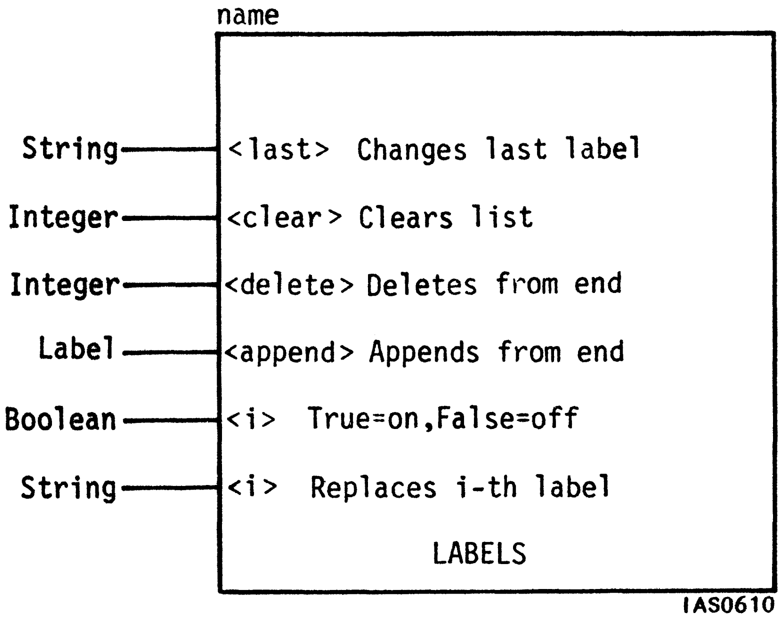


Figure 8. Inputs to a LABELS Node

Unlike most other nodes, these nodes have inputs with names as well as numbers. All data sent to these nodes are sent to a named input or to a numeric input which indicates the position of a character within a string or a label within a block.

The simplest form of the SEND command has the following format:

SEND *option* TO <*n*>name1;

The parameters in this command are as follows:

option - For sending to a LABELS node, this is a string enclosed in single quotes. For sending to a CHARACTERS node, the format is ***CHAR(number)***, where ***number*** is the ASCII decimal equivalent of a single character.

n - The name or number of the input to the LABELS or CHARACTERS node.

name1 - The name of the destination LABELS or CHARACTERS node.

You can use the command, for example, to send a new string to replace an existing one. Create a string called Quote.

```
Quote := CHARACTERS -1,0 'If we had world enough and time';
```

Now scale the string by .05 so it will fit the default window.

```
Scale_Quote := CHARACTER SCALE .05 THEN Quote;
```

Remove anything you are displaying and display Scale_Quote. Now use the SEND command to replace this string with the second line of John Donne's poem to his reluctant mistress.

```
SEND 'This coyness, mistress, were no crime' TO <substitute>Quote;
```

Exercise

Try SENDing to some of the other inputs of CHARACTERS and LABELS nodes. For more information, refer to the ***Command Summary*** in Volume 3A.

Two other forms of the SEND command can be used with LABELS but not with CHARACTERS: they are ***SEND VL*** and ***SEND number*mode***. The ***SEND VL*** form allows you to overwrite or append labels in a labels block. The ***SEND number*mode*** form allows you to send a P or L identifier to a label to indicate if a label is off (P) or on (L). Refer to the ***Command Summary*** for more details.

Updating With Functions

You can create function networks to update a CHARACTERS or LABELS node. Only four data types are accepted by the inputs to these nodes: an integer, a vector (2D or 3D), a character string, and a Boolean value. Any function which outputs one of these data types can be used to feed new values to a node containing text. In particular, the output of the string handling functions mentioned earlier can be used as input to a text node.

The function F:LABEL is designed specifically for updating a LABELS node. The data type output by this function is the only type accepted by input <append> of a LABELS node.

CREATING AND USING DIFFERENT CHARACTER FONTS

A *character font* is a complete set of characters in the same size and type face. The PS 300 has a standard font consisting of the 128-character ASCII set. This is the default font for all textual displays. There are two commands which let you create and use alternate character fonts: the BEGIN_FONT ... END_FONT command and the CHARACTER FONT command.

Creating an Alternate Font

Alternate fonts are created as a sequence of itemized, two-dimensional vector lists defining each character in the font. Up to 128 ASCII character codes can be defined for each font.

Each character in the font is defined as follows.

```
C[i]: N=n  vectors;
```

The parameters are:

[*i*] - The decimal ASCII code to be defined, i.e. a number from 0 to 128.

n - The number of vectors in the 2D vector list.

vectors - The vectors which make up the character.

The vectors which comprise a character must be itemized 2D vectors. Itemized vectors are each preceded by P or L identifiers to indicate whether a vector is a position or a line vector. The following is the definition of a capital 'A' in a font called Simplex_Roman.

```
C[65]: N= 6
P 0.5455, 0.9545  L 0.1818, 0.0000
P 0.5455, 0.9545  L 0.9091, 0.0000
P 0.3182, 0.3182  L 0.7727, 0.3182;
```

The Simplex_Roman letter 'A' is compared to an 'A' in the standard font in Figure 9.

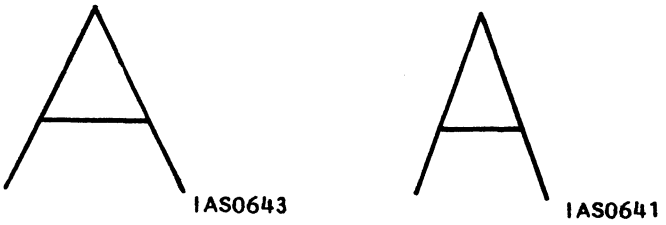


Figure 9. Standard 'A' and Simplex Roman 'A'

In an Old English font, the definition of the same letter is much more complex.

```
C[65]: N=49
P 0.2727, 0.8182  L 0.3636, 0.9091  L 0.4545, 0.9545  L 0.5455, 0.9545
L 0.5909, 0.9091  L 0.9091, 0.1818  L 0.9545, 0.1364  L 1.0455, 0.1364
P 0.5000, 0.9091  L 0.5455, 0.8636  L 0.8636, 0.1364  L 0.9091, 0.0455
L 0.9545, 0.0909  L 0.8636, 0.1364  P 0.3636, 0.9091  L 0.4545, 0.9091
L 0.5000, 0.8636  L 0.8182, 0.1364  L 0.8636, 0.0455  L 0.9091, 0.0000
L 0.9545, 0.0000  L 1.0455, 0.1364  P 0.2727, 0.6364  L 0.3182, 0.6818
L 0.4091, 0.7273  L 0.4545, 0.7273  L 0.5000, 0.6818  P 0.4545, 0.6818
L 0.4545, 0.6364  P 0.3182, 0.6818  L 0.4091, 0.6818  L 0.4545, 0.5909
P 0.0455, 0.0000  L 0.1364, 0.0909  L 0.2273, 0.1364  L 0.3636, 0.1364
L 0.4545, 0.0909  P 0.1818, 0.0909  L 0.3636, 0.0909  L 0.4091, 0.0455
P 0.0455, 0.0000  L 0.1818, 0.0455  L 0.3182, 0.0455  L 0.3636, 0.0000
L 0.4545, 0.0909  P 0.5455, 0.7727  L 0.2727, 0.1364  P 0.3636, 0.3636
L 0.7273, 0.3636;
```

This letter 'A' is compared to the standard font 'A' in Figure 10.

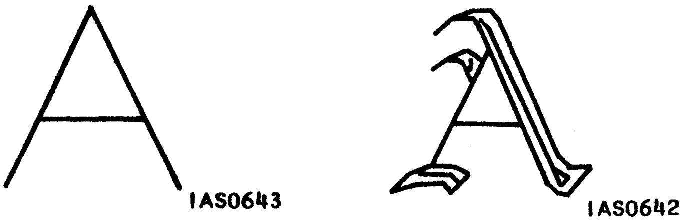


Figure 10. Standard 'A' and Old English 'A'

A complete set of character definitions is enclosed in a BEGIN_FONT ... END_FONT structure with the following format.

```
New_Font :=  BEGIN_FONT

             C[0]: N=n P, L, L, ... L;

             C[n]: N=n P, L, L, ... L;

             C[127]: N=n P, L, L, ... L;

             END FONT;
```

Notice that in the sample 2D vector lists given, the range of the vectors in X and Y is between 0 and 1. There is no limit on the range of the vectors you use, but you should keep within the range of 0 and 1 for the correct spacing and orientation of adjacent characters.

Using an Alternate Font

The BEGIN_FONT ... END_FONT command does not create a data node in a display tree but a look-up table of alternate character definitions. To switch to an alternate font in a structure, the CHARACTER FONT command is used to create an attribute node which indicates the font look-up table that must be read for the character definitions.

An alternate font called Old_English is included on the Tutorial Demonstration Tape. To use this font in a structure, you must create a node which points to the Old_English font and apply it to the text you want to display.

Create, scale, and display a character string.

```
Text := CHARACTERS -.5,0 'To be, or not to be';
Scale_Text := CHARACTER SCALE .05 APPLIED TO Text;
DISPLAY Scale_Text;
```

Now apply a CHARACTER FONT command to the scaled string to display it in the Old_English font.

```
New_Font := CHARACTER FONT Old_English APPLIED TO Scale_Text;
REMOVE Scale_Text;
DISPLAY New_Font;
```

Hamlet's question should now be displayed in the Old_English font. If it is displayed in the standard font instead, this means that the Old_English font was not available.

The display tree for New_Font is shown in Figure 11.

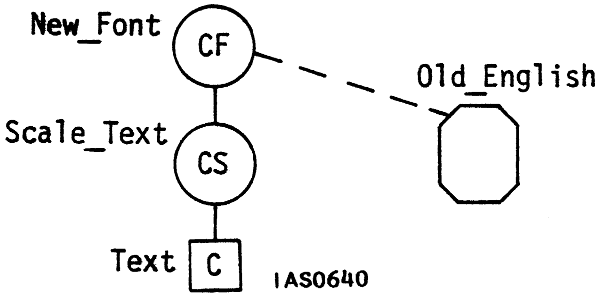


Figure 11. Display Tree With CHARACTER FONT Node

The Old_English font is shown as a look-up table which is not part of the actual structure. The CHARACTER FONT node New_Font points to this table as well as to the CHARACTER SCALE and CHARACTERS node.

The Character Font Editor Program

Another way to create alternate character fonts is to use the program MAKEFONT which is distributed on magnetic tape and is documented in Volume 4. MAKEFONT is a menu-driven, graphical character font editing program which allows you to create a font from scratch by drawing each of the characters, or to make changes to existing alternate fonts. Refer to the MAKEFONT user's guide in Volume 4 for details.

SUMMARY

Two commands create data nodes containing text: CHARACTERS and LABELS.

Creating Text Nodes

The CHARACTERS command creates a single text string of up to 240 characters. Optional parameters allow you to specify the starting location of the string and the horizontal and vertical spacing between characters. The syntax of the command is as follows.

```
Name := CHARACTERS [x,y[,z]][STEP dx,dy] 'string';
```

The LABELS command creates a block of character strings or labels. Each label can be given its own starting location. The syntax of the command is as follows.

```
Name := LABELS  x,y [,z] 'string'
                    .
                    .
                [xi,yi [,zi] 'string'];
```

Manipulating Text With Commands

Text nodes, just like any other data nodes, are affected by transformations. They can be rotated and scaled by 3X3 transformation matrices (created by the ROTATE and SCALE commands) or by exclusive 2X2 character transformation matrices.

Transforming Text

The commands which create these matrices are CHARACTER ROTATE, CHARACTER SCALE, and TEXT SIZE. The matrices which these commands create have no effect on three-dimensional data or non-textual two-dimensional data.

The CHARACTER ROTATE command creates a Z-rotation matrix from an angle of rotation which is entered as parameter. The syntax of the command is as follows.

```
Name := CHARACTER ROTATE angle [APPLIED TO Name1];
```

The CHARACTER SCALE command creates a uniform or non-uniform scaling matrix from the scale factor entered with the command. For non-uniform scaling an X and Y scale factor is given. The syntax of the command is as follows.

```
Name := CHARACTER SCALE s [APPLIED TO Name1];
Name := CHARACTER SCALE sx,sy [APPLIED TO Name1];
```

The TEXT SIZE command creates a 2X2 matrix node which overrides any 2X2 matrix settings above it in the display tree. Any character scales or character rotations are superseded by this command. The command establishes a character size for text which is a multiple or fraction of the default character size of 1. The syntax of the command is as follows.

```
Name := TEXT SIZE x [APPLied to Name1];
```

Setting Character Orientation

When text forms part of an object that is being displayed and manipulated, the characters can be transformed with the object or they can remain unaffected by object transformations. The SET CHARACTERS command lets you determine the orientation of the text. The format of the command is as follows.

```
Name := SET CHARACTERS orientation [APPLIED TO Name1];
```

Three types of orientation may be set:

WORLD_ORIENTED - Characters are transformed just like any part of the object containing them.

SCREEN_ORIENTED - Characters are not affected by ROTATE or SCALE transformations. Intensity and size of characters still vary with depth (Z-position).

SCREEN_ORIENTED/FIXED - Characters are not affected by ROTATE or SCALE transformations. They are always displayed with full size and intensity.

Manipulating Text With Functions

Several functions are available for manipulating text and strings. These functions are listed below.

Character and String Conversion

F:CHARCONVERT
F:CHARMASK
F:GATHER_STRING
F:PRINT
F:STRING_TO_NUM
F:TRANS_STRING

String Formatting and Reformatting

F:CONCATENATE
F:LABEL
F:LINEEDITOR
F:PUT_STRING
F:SPLIT
F:TAKE_STRING

Miscellaneous String Handling Functions

F:COMP_STRING
F:FIND_STRING
F:LBL_EXTRACT
F:LENGTH_STRING

Character Transformation Functions

F:CROTATE
F:CSCALE
F:MATRIX2

Text Nodes

The CHARACTERS and LABELS commands create data nodes containing text. Both nodes have inputs which accept vectors, strings, integers, or Boolean values to update the contents of the node.

Updating Nodes

CHARACTERS and LABELS nodes can be updated using commands or the functions listed earlier. The following commands are most frequently used to update these nodes.

```
COPY
SEND
SEND VL
SEND number*mode
```

Alternate Character Fonts

Character fonts other than the standard font can be created using the BEGIN_FONT ... END_FONT command. The syntax for this command is as follows.

```
    Name := BEGIN_FONT
                    [C[0]: N=n {itemized 2D vectors};]
                        .
                        .
                        .
                    [C[i]: N=n {itemized 2D vectors};]
                        .
                        .
                        .
                    [C[127]: N=n {itemized 2D vectors};]
               END_FONT;
```

Each character in the font is defined as a vector list consisting of itemized 2D vectors. The clause ***C[i]*:** identifies the ASCII character being defined; for example, ***C[65]*:** indicates that the character is a capital 'A'. Up to 128 characters can be defined in an alternate font.

Alternate fonts are used by including CHARACTER FONT nodes in a display tree. The syntax of the CHARACTER FONT command is as follows.

```
Name := CHARACTER FONT font_name APPLIED TO Name1;
```

The parameter ***font_name*** is the name of an alternate font defined with the BEGIN_FONT ... END_FONT command.

PICKING

SELECTING DISPLAYED OBJECTS

CONTENTS

ILLUSTRATIONS

Picking allows you to retrieve information about a selection or ***pick*** made on displayed data. This information contains details about the structure that makes up the displayed data. Details can include the name of the data node that the picked portion of the object is associated with, names of nodes along the branch of the display structure that was selected by a pick, an index into the vector list, character string or label that was picked, and the coordinate values of the location where the pick took place. The information is available in a special format called the ***picklist***.

Normally, picking is done by using the data tablet and the stylus to select any part of a displayed object designed to allow for picking. The selection is made by moving the stylus across the surface of the data tablet; this positions the cursor on the screen. (The cursor is an X.) Picking is usually activated by pressing the tip of the stylus down when the cursor is positioned over the appropriate line, dot, or text character. The information that is returned when a pick takes place, the picklist, can be displayed, used to drive a function network, or sent to the host. The amount and kind of information received on the location of a pick is user-defineable.

An obvious use of picking is to make selections from a menu, where the cursor is positioned over a line or the piece of text in the menu that is to be selected. By pressing the stylus down, that item on the menu is "picked", and the appropriate function can be performed (i.e., move to another menu, exit the menu, bring up a displayable structure, etc.)

Central to the picking process is the initial function instance, PICK. PICK is enabled by sending any message to input <1> of PICK. (Normally this message is the X,Y location of the pick sent to PICK when the tipswitch of the stylus is depressed.) PICK feeds this trigger message to the display processor, asking for any pick information within the data structure being traversed to be sent back to PICK. If this information is found (a pick occurs if there is data) the data associated with the pick, the picklist, is placed on the queue of output <1> of PICK. The main responsibility of PICK is to signal the display processor that picking has been enabled and to output the picklist that contains information about the location of the pick.

Before picking can take place, the data structure that you want to be able to pick from must contain certain nodes and pieces of information. Polygonal objects, because of their construction, cannot be picked.

This module will define the various elements involved in picking: picking attribute nodes and the commands that create them, and the picking functions.

This module will teach how to place and set the appropriate attribute nodes used in picking and how to design a function network to use the information that is generated when a pick occurs.

OBJECTIVES

In this module you will learn how to:

- Use Picking Attribute Nodes
- Use Initial Picking Functions
- Use the Picking Functions in a Function Network

PREREQUISITES

Before using this module, you should be familiar with the following:

Designing display trees
Creating function networks
Using the PS 300 command language

USING PICKING ATTRIBUTE NODES

Before an object can be picked, the display tree of the object must contain certain nodes and the object must be displayed. These nodes provide for picking capabilities such as:

- Turning picking on and off
- Determining the portions of the object (or branches of the object's display tree) that can be picked
- Selecting the name of the ***pick identifier*** that will be returned as part of the picklist

Set Picking ON and OFF

The first picking attribute node that must appear in the display tree is the SET PICKING ON/OFF node. This node must be above the parts of the display tree where picking will take place. This node is turned on and off by Boolean values; a TRUE will enable picking in the data structure below the node, a FALSE will disable it.

The command that creates the SET PICKING ON/OFF node is:

```
Name := SET PICKING OFF APPLIED TO Name1;
```

The SET PICKING ON/OFF node is usually placed in the display tree in an "off" condition and activated when the Boolean value TRUE is sent to input <1> of the named node. As an example, the following two commands first create an instance of a SET PICKING ON/OFF node, and then activate that node.

```
Pick_Car := SET PICKING OFF APPLIED TO Car;
```

where Car is the name of the data structure, or the part of a data structure that you want to be able to pick from,

```
SEND TRUE TO <1>Pick_Car;
```

activates picking for Car. (The Boolean value is normally sent by a network connected to the node.)

In designing a pickable display tree, the placement of the SET PICKING ON/OFF nodes is very important. As with any other attribute node, this node controls only its descendants. In the structure in Figure 1, picking can be enabled and disabled for each branch individually because of the placement of the SET PICKING ON/OFF nodes. In Figure 2, picking is established for the whole structure, but not for the individual branches.

This placement can be important in complicated display structures, where there are close or overlapping data structures simultaneously displayed on the screen. In molecular modeling graphics applications, it can be useful to disable picking for specific parts of the molecule. This same principle holds for architectural or engineering applications, where only specific parts of the entire display will be used as pickable structures.

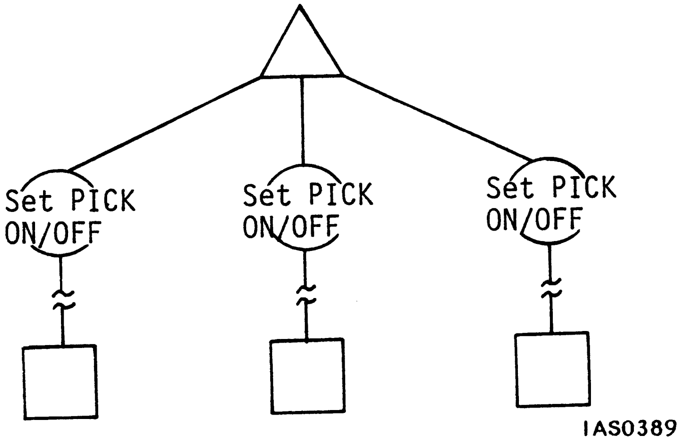


Figure 1. Picking Selectable by Branch

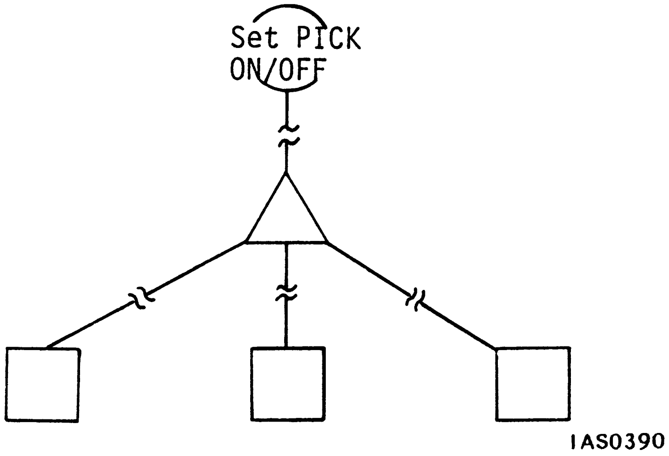


Figure 2. Picking an Entire Structure

Using Picking Identifiers

The other attribute node that must be placed in the display tree for picking is the SET PICKING IDENTIFIER node. This pick identifier node determines how detailed the information you get back in your picklist (output from the PICK function) will be.

A picked object is identified by two types of names in the picklist (pick information output from PICK). The first type of name is the picking identifier or the pick ID. The second name is the name of the data node that contains the picked vector or character (in the command shown above, "Car" would be the name of the node that contains the picked vector).

The command to create a set picking identifier node is:

```
Name := SET PICKING IDENTIFIER = id_name APPLIED TO Name1;
```

This command assigns id_name to be the picking identifier (the reported character string) to be output by PICK in the picklist if any part of Name1 is picked. Id_name can be the name of the data node, but in many cases, several branches of a display structure terminate at the same data node. The name(s) of the pick identifiers in the picklist in such cases show which branch was traversed to get to the common data node.

Example

```
Wheelpick1 := SET PICKING IDENTIFIER = Wheel1 APPLIED TO Wheel;
```

In this example, it is assumed that the display tree includes a car with four tires. There are five branches, four of which include an instance of the vector list for "Wheel". Each branch contains the appropriate translate and rotate operation nodes required to position the tires. To determine which instance of "Wheel" was picked, each branch must also contain a set pick identifier node with a unique name. This is illustrated in Figure 3.

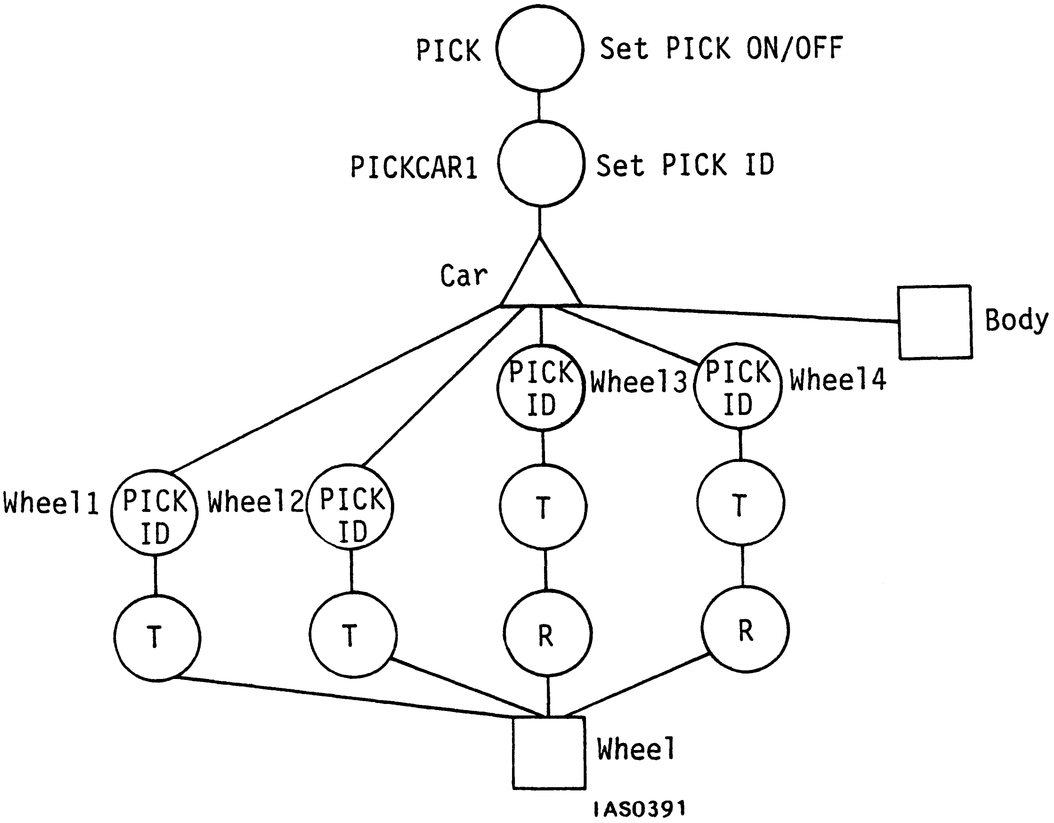


Figure 3. Display Tree With Car and Four Tires

Assuming the right-front tire is Wheell, then the picklist generated when a pick was made on the right-front tire would be:

‹index› Wheell,Pickcarl Wheel

If there had been only one set picking identifier node directly below the SET PICKING ON/OFF node in Figure 3, when you picked from any part of the displayed object below the instance node, you would only get back the pick identifier for the whole data structure:

‹index› Pickcarl Wheel (or body)

The information in a picklist includes the names of ALL the set pick ID nodes down the branch of the display structure that has been enabled for picking. The picklist will also include the name of the picked data node. The picklist can be reported as a character string with pick IDs on that branch separated by commas. This list always starts with the name of the set pick ID node closest to the picked vector or character.

The amount of detail about the display tree contained in information returned in the picklist is determined by the location and number of the set pick IDs. In the code below, the picklist is will contain only one pick identifier (Pickcar1).

```
Display Car;

Car :=   BEGIN_STRUCTURE
Pick :=  SET PICKING OFF;
         SET PICKING IDENTIFIER = Pickcar1;
         INSTANCE OF Body, Wheel1,Wheel2,Wheel3,Wheel4;
         END_STRUCTURE;
```

Setting up the display tree to enable picking follows one simple rule:

For picking to take place, there must a SET PICKING ON/OFF node placed in the display structure, followed by at least one SET PICK IDENTIFIER node down each pickable path. However, one structure can contain multiple SET PICKING ON/OFF nodes, and each SET PICKING ON/OFF node can be followed by multiple SET PICKING IDENTIFIER nodes.

USING INITIAL PICKING FUNCTIONS

The initial system function PICK was briefly described in the introduction to the module. The initial function network that should be built to make use of picking is shown in Figure 4.

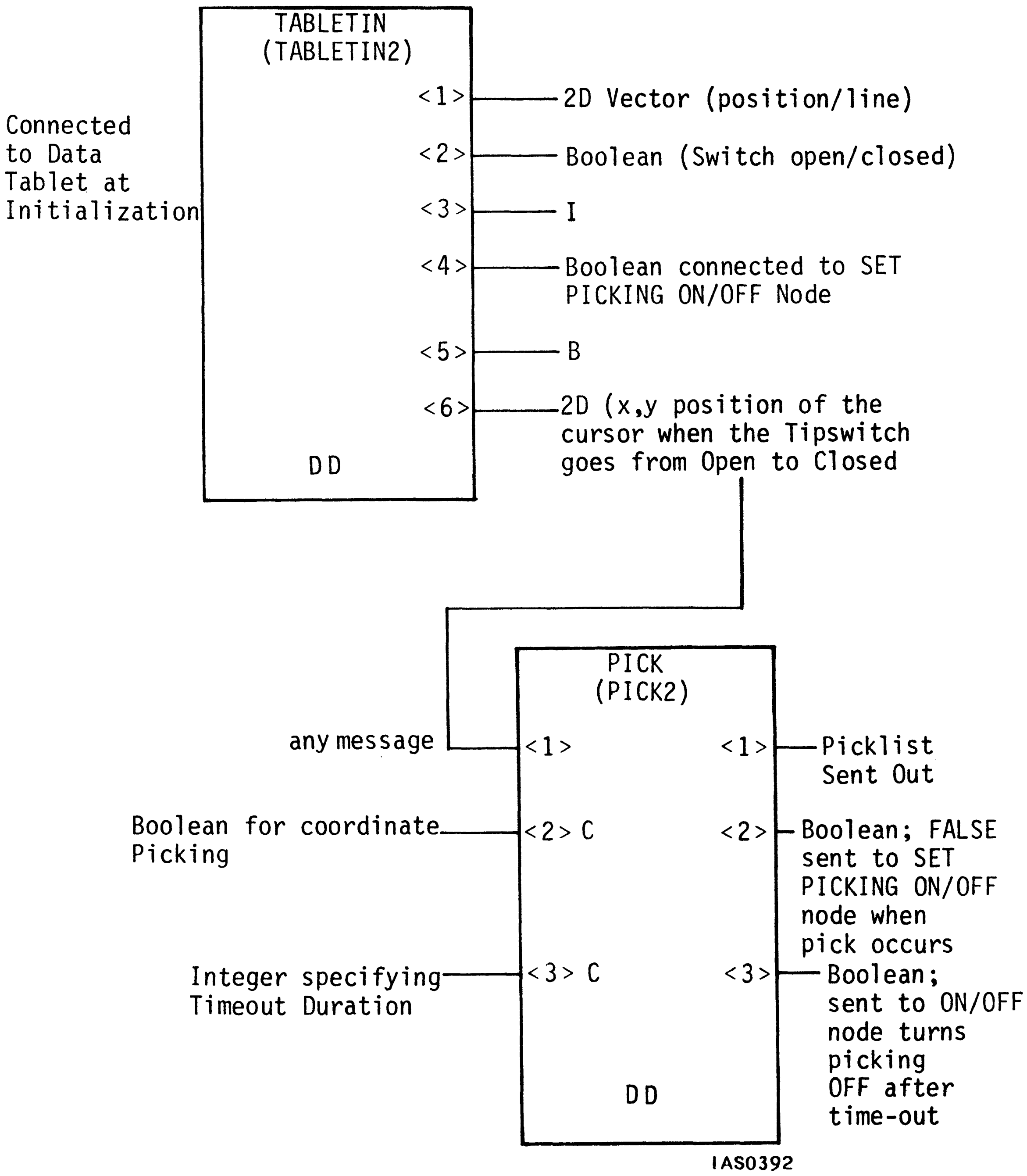

TABLETIN
(TABLETIN2)
Connected to Data Tablet at Initialization
<1>
2D Vector (position/line)
<2>
Boolean (Switch open/closed)
<3>
I
<4>
Boolean connected to SET PICKING ON/OFF Node
<5>
B
<6>
2D (x,y position of the cursor when the Tipswitch goes from Open to Closed
DD
PICK
(PICK2)
any message
<1>
<1>
Picklist Sent Out
Boolean for coordinate Picking
<2> C
<2>
Boolean; FALSE sent to SET PICKING ON/OFF node when pick occurs
Integer specifying Timeout Duration
<3> C
<3>
Boolean; sent to ON/OFF node turns picking OFF after time-out
DD
IAS0392

Figure 4. Diagram of TABLETIN and PICK

The system has provided for picking with one other initial function, TABLETIN. TABLETIN accepts the X,Y vectors that identify the position of the picking location (and the center of the cursor cross) as the stylus moves across the data tablet and uses them to position the cursor on the screen. TABLETIN identifies the X,Y coordinates of the picking location that are output when the tipswitch on the stylus is pressed. These coordinates are used to to determine if a pick has occurred and if it has, the location of the pick is made available.

Output <4> of TABLETIN is typically connected to the SET PICKING ON/OFF nodes in the display strucuture and is used to send Boolean values to the nodes. When the tipswitch on the stylus is pressed, a TRUE is sent to the node, enabling picking.

Input <1> of PICK accepts any message. Typically, this queue is connected to TABLETIN<6> which supplies the 2D coordinates of the pick location when the tipswitch is pressed. This arms the function, as the other two inputs to PICK are constants. Output <2> of PICK should be connected to the same SET PICKING ON/OFF nodes that are connected to output <4> of TABLETIN. This output sends a FALSE whenever a pick occurs which turns picking off until the tipswitch is again pressed and a TRUE is sent from TABLETIN to the ON/OFF node. (This false is sent to disable picking so that the picking process ceases until a pick location is asked for.)

Input <2> of PICK accepts a Boolean value that allows you to select the kind of picklist that will be sent out of output <1>. A FALSE sent to <2>PICK indicates that the output picklist will be the pick ID names, the data node name, and an index into the vector list or character string (the data node). A TRUE sent to <2>PICK indicates that the picklist will include the pick ID names, the data node name, an index to the data node, and the picked coordinates and the dimension (2D or 3D) of the picked vector.

The format for the picklist then, with FALSE sent to <2>PICK is:

    <index> pickid1,pickid2, _name of data node

where <index> is a pointer into the picked data node.

The chart below shows the data node types and the definition of the <index> that is returned when the value of the <index> is the integer 3.

Data Node Type	Index of 3 Definition
Vector list	The third vector in the list was picked.
Character String	The third character in the string was picked.
Label	The third character string in the label was picked.
Polynomial or Rational polynomial curve	The value of the parameter (t) where the curve was picked

The format for the picklist with TRUE sent to <2>PICK (coordinate picking) is:

```
<index> [x,y,z] pickid1,pickid2, _name of data node
```

where x,y,z are the coordinate points of the picked vector.

- Performing coordinate picking on a character string returns an index into the string, not its picked coordinates.
- Performing coordinate picking on a label block returns an index into the label, not its picked coordinates.
- Coordinate picking cannot be performed on a vector over 500 units long.

The integer on <3>PICK is used to set a timeout interval for the PICK function in refresh frames. Timing starts when the PICK function receives any message on input <1>. This timing interval is used to determine if a pick occurs in the specified period of time. The allowable integers on input <3> are from 4 through 60. This is a safeguard feature: it deactivates PICK if no pick occurs within the timeout period.

Once the PICK function is armed (by receiving input on <1>PICK), if no pick occurs within the specified time, PICK outputs a FALSE on PICK<3>. This output should be connected to the ON/OFF nodes to disable picking when a timeout occurs. Picking is enabled when the stylus is again pressed.

One other feature that is initialized by the system is the ***picking location.*** This is by default the center of the cursor. The picking location must be defined within the current viewport and can be modified with the following command:

```
name := SET PICKING LOCATION x,y sizex,sizey APPLIED TO Name1;
```

where:

the 2D vector X,Y specifies the center of the picking location and the 2D vector sizex,sizey specifies the size in X and Y from the center to the edge of the picking location. Name1 is the current and applicable viewport name.

The pick location, then, specifies a region within a screen. If the pick-sensitive object (line, dot, or character) is within the pick location, it can be reported as having been picked.

The pick location can be moved within the viewport by sending the 2D vector that represents the coordinate location of the new set pick location to input <1> of the set picking location node. In effect, picking can take place by positioning the picking location over a displayed object (containing the appropriate picking attribute nodes) and sending a TRUE to <1>PICK.

The following diagram is a typical arrangement of the TABLETIN and PICK functions and their connections to the display structure.

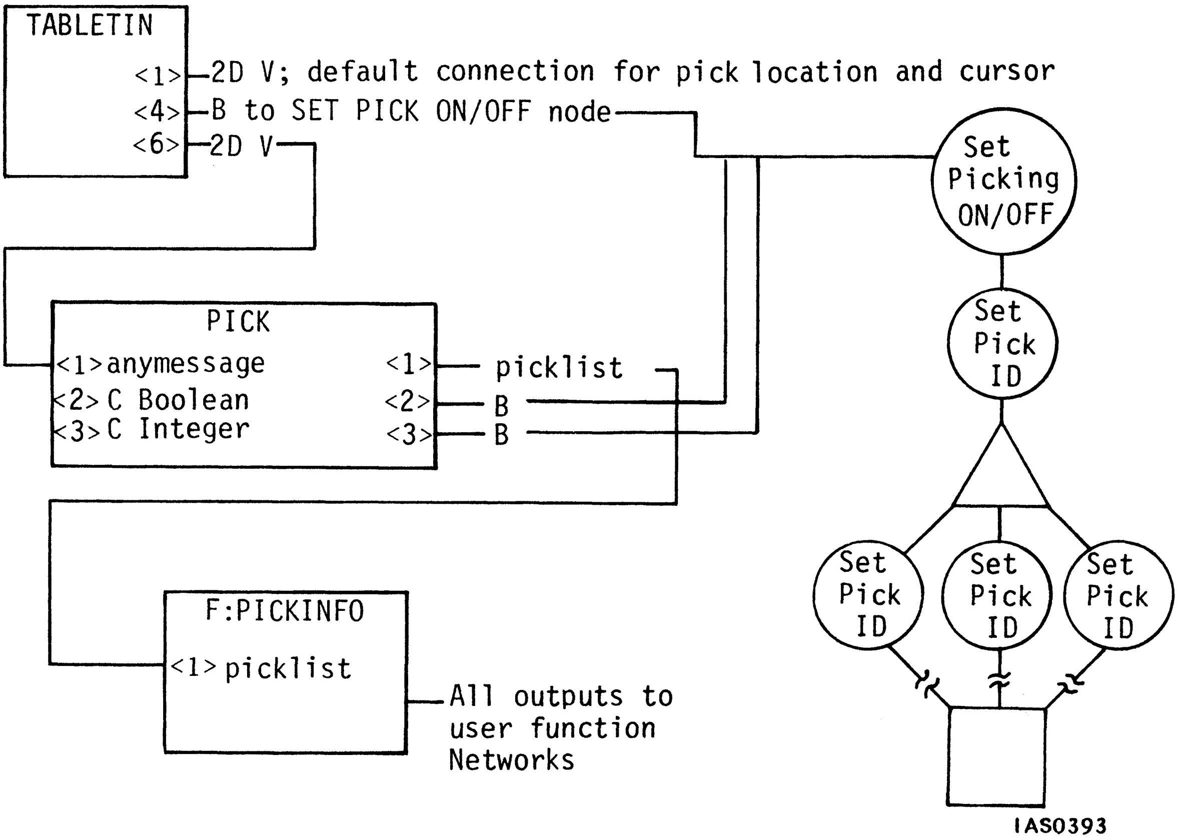


Figure 5. Typical TABLETIN and Pick Arrangement

USING THE PICKING FUNCTIONS IN A FUNCTION NETWORK

A function associated with picking is F:PICKINFO. This function converts the picklist data type into character strings that are acceptable by other functions. There is only one active input to F:PICKINFO, <1>, and it should be connected to output <1> of PICK.

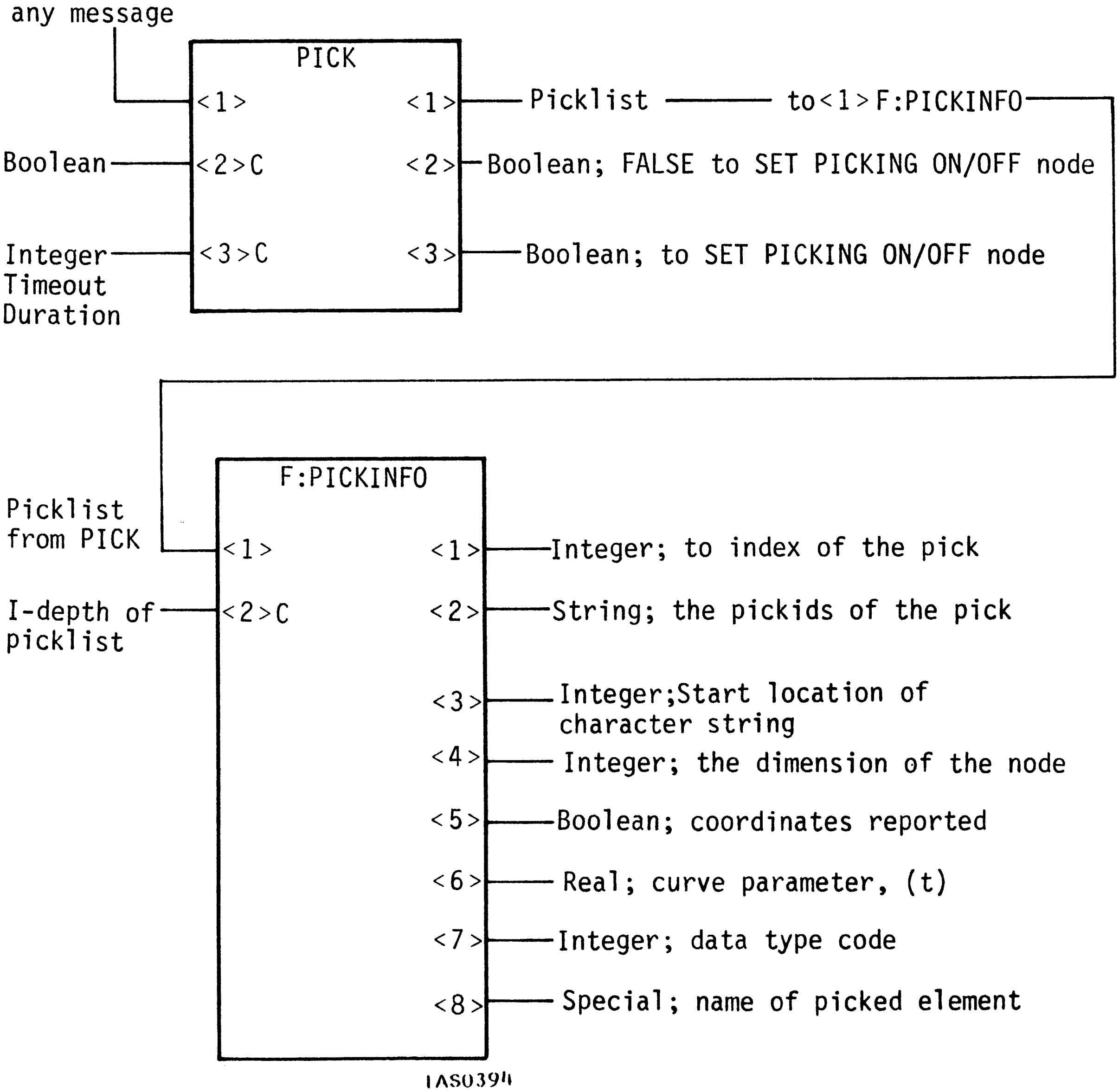

any message
PICK
<1>
<1>
Picklist
to<1>F:PICKINFO
Boolean
<2>C
<2>
Boolean; FALSE to SET PICKING ON/OFF node
Integer Timeout Duration
<3>C
<3>
Boolean; to SET PICKING ON/OFF node
F:PICKINFO
Picklist from PICK
<1>
<1>
Integer; to index of the pick
I-depth of picklist
<2>C
<2>
String; the pickids of the pick
<3>
Integer;Start location of character string
<4>
Integer; the dimension of the node
<5>
Boolean; coordinates reported
<6>
Real; curve parameter, (t)
<7>
Integer; data type code
<8>
Special; name of picked element
IAS0394

Figure 6. F:PICKINFO (Connected to PICK)

The picklist output from PICK<1> can be connected to an instance of F:PICKINFO to convert the picklist into a logically useful format. The picklist can also be printed out or displayed by connecting PICK<1> to F:PRINT. F:PRINT converts the picklist code to printable characters.

The constant input <2> of F:PICKINFO accepts an integer that specifies the depth of the pick identifiers that will be output. Since the picklist contains all of the set pick IDs in a picked branch of a display tree, this input allows you to select the depth. For example, if there were four pick IDs active when a pick occured and the integer 2 was input to <2>F:PICKINFO, then the two pick IDs closest to the data node and the name of the data node itself would be output as the string on F:PICKINFO<2>.

The output information from F:PICKINFO varies with the type of picklist supplied on input <1>. If the PICK function has a TRUE on input <2>, then it supplies a detailed coordinate picklist and most of F:PICKINFO outputs are activated. If the PICK function has a FALSE on input <2>, a less detailed picklist is supplied, and only outputs <1>, <2>, and <5> are active. Refer to the ***Function Summary*** in Volume 3 for a complete description of the outputs of F:PICKINFO.

The best use of picking is when the picklist is sent to an instance of F:PICKINFO. Then information generated by the function can be used to drive function networks that can be triggered by typical data types. Examples of what this data can be used for are described in the next section.

Examples of Picking

The following example demonstrates how picking can be used to trigger a switching network for an object designed to have parts with independent motion. The control dials are normally used to rotate, translate, and scale objects in three dimensions. If the designed object requires more than eight elements of freedom (the maximum number that can be provided by one set of control dials), a picking network can be set up to access a bank of switching functions that control the output of the dials. This network will allow you to point at the part that you want to manipulate and the picking information will drive the function network that routes the dial outputs to various networks.

In this example, the display tree that defines a robot figure includes set pick IDs in each branch of the figure networked for motion through a switch function to DIALS. This is the same robot that was built in the "PS 300 Command Language" module and it is connected to the function networks that were designed in the Function Networking module. The function network provides for several modes for the control dials. These modes provide the triggers to animate each part of the robot that requires independent movement; i.e., rotation of each shoulder joint, knee joints, torso, head, etc.

The picking network will use the data tablet to trigger the mode of the dials. In the "Function Networks I" module, the Function Keys were used for dial mode switching. If you examine the design of the robot, you will notice that there are 'n' degrees of freedom designed into the structure. This will require 'n modes' of the dials. As the picking network will be used to trigger the dials mode, only 'n' set pick ids need to be coded into the structure.

The picking network to switch the modes for dials that are connected to the robot display structure works in the following manner. When the cursor is positioned over a part of the robot with independent motion controlled by a dial (like the shoulder) and the tipswitch of the stylus is pressed, the name of the pickid in the shoulder branch of the display tree is sent from PICK to an instance of F:PICKINFO. This instance of F:PICKINFO<2> is connected to an instance of F:CHARCONVERT. F:CHARCONVERT converts the bytes of the string it receives on input <1> into a stream of integers. If the pick id sent to F:PICKINFO is an 'A', F:CHARCONVERT will translate 'A' to the ASCII 65. If this is then sent to an instance of F:SUBC, it can subtract 64 and output the integer 1 that can be used to trigger the appropriate bank of switches for the dials.

Figure 7 illustrates the function network described above.

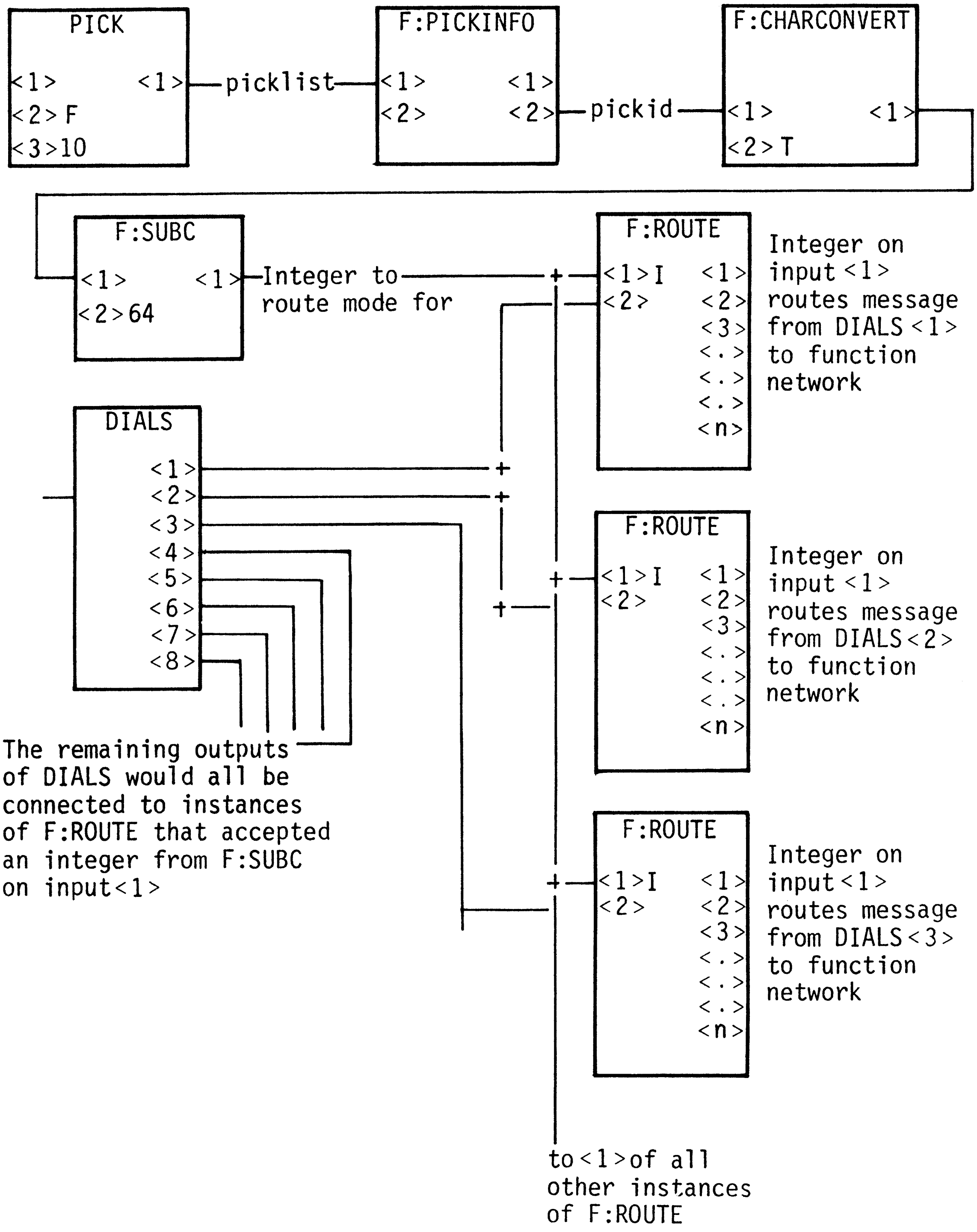

PICK
<1>
<2> F
<3>10
<1>
picklist
F:PICKINFO
<1>
<2>
<1>
<2>
pickid
F:CHARCONVERT
<1>
<2> T
<1>
F:SUBC
<1>
<2>64
<1>
Integer to route mode for
F:ROUTE
<1>I
<2>
<1>
<2>
<3>
<.>
<.>
<.>
<n>
Integer on input <1> routes message from DIALS <1> to function network
DIALS
<1>
<2>
<3>
<4>
<5>
<6>
<7>
<8>
F:ROUTE
<1>I
<2>
<1>
<2>
<3>
<.>
<.>
<.>
<n>
Integer on input <1> routes message from DIALS <2> to function network
The remaining outputs of DIALS would all be connected to instances of F:ROUTE that accepted an integer from F:SUBC on input<1>
F:ROUTE
<1>I
<2>
<1>
<2>
<3>
<.>
<.>
<.>
<n>
Integer on input <1> routes message from DIALS <3> to function network
to <1> of all other instances of F:ROUTE
IAS0395

Figure 7. Diagram of PICK Through F:SUBC Feeding a Bank of F:ROUTE(n) Instances

To implement the previous example of picking as an exercise demonstrating the placement of the picking attribute nodes and the connections that should be made for the picking network, use the source code supplied for the robot in the "PS 300 Command Language" module. Picking attribute nodes can be set into the display structure and then connected to the picking function network that is used in the picking demonstration available on the Tutorial Demonstration tape.

Exercise

Design a pickable display structure with several instances of a primitive.

Design a function network that outputs the picklist to the screen. Use F:PRINT and a character data node. Code your display structure and function network. Display and pick each primitive.

SUMMARY

Picking allows you to retrieve information about a selection or "pick" made on displayed data. The information is available in a special format called the picklist. Before picking can take place, the data structure that you want to be able to pick from must contain certain nodes and pieces of information.

Picking Attribute Nodes

The first picking attribute node that must appear in the display tree is the SET PICKING ON/OFF node. This node must be above the parts of the display tree where picking will take place. This node is turned on and off by Boolean values; a TRUE will enable picking in the data structure below the node, a FALSE will disable it.

The command that creates the SET PICKING ON/OFF node is:

```
Name := SET PICKING OFF APPLIED TO Name1;
```

The other attribute node that must be placed in the display tree for picking is the SET PICKING IDENTIFIER node. This pick identifier node determines how detailed the information you get back in your picklist (output from the PICK function) will be.

A picked object is identified by two types of names in the picklist (pick information output from PICK). The first type of name is the picking identifier or the pick ID. The second name is the name of the data node that contains the picked vector or character.

The command to create a set picking identifier node is:

```
Name := SET PICKING IDENTIFIER = id_name APPLIED TO Name1;
```

For picking to take place, there must a SET PICKING ON/OFF node placed in the display structure, followed by at least one SET PICK IDENTIFIER node down each pickable path. However, one structure can contain multiple SET PICKING ON/OFF nodes, and each SET PICKING ON/OFF node can be followed by multiple SET PICKING IDENTIFIER nodes.

Picking Functions

An initial system function used for picking is PICK. Input <1> of PICK (usually connected to TABLETIN<6>) accepts any message type as a trigger message to activate picking. The data associated with the pick, the picklist, is placed on the queue of output <1> of PICK. The main responsibility of PICK is to signal the display processor that picking has been enabled and to output the picklist that contains information about the location of the pick.

Another function associated with picking is F:PICKINFO. This function converts the picklist data type into character strings that are acceptable by other functions. There is only one active input to F:PICKINFO, <1>, and it should be connected to output <1> of PICK.

TRANSFORMED DATA

RETRIEVING TRANSFORMATION INFORMATION

CONTENTS

The PS 300 provides a means to retrieve ***transformed data.*** Transformed data is the matrix or vector-list representation of transformation operations in a display tree.

After an object has been transformed on the PS 300, the transformed accummulated data for that object can be:

(1) Established as a separate data or operation node in the display tree.

(2) Retrieved as ASCII information for transmission to the host computer.

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 300 VECTOR_LIST ITEMIZED command can be generated. If a transformation matrix is requested, an operation node can be created and an ASCII PS 300 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 (CHAR ROTATE, etc.) are excluded from both forms of retrieved transformed data.

OBJECTIVES

This discussion of transformed-data retrieval covers the following topics:

- The XFORM command and the F:XFORMDATA and F:LIST functions.
- A note on excluding perspective and window transformations from transformed vector lists.
- A suggested function network to prevent successive transformed-data requests from overlapping.

- A note on restricting transformed-data retrieval to a specified range of vectors within a list.
- A program example.

PREREQUISITES

Before reading this module, you need to know the basics of data structures and function networks. These topics are covered in Volume 2A, in the "Modeling," "PS 300 Command Language," and "Function Networks I" modules, and in this volume under "Function Networks II."

RELEVANT PS 300 COMMANDS AND FUNCTIONS

To retrieve transformed data for a given data node (or set of data nodes):

- Mark the data node by applying an 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 300 command string using an instance of F:LIST and send this ASCII information to the host computer via HOST_MESSAGE.

The following paragraphs discuss these topics.

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. VECTOR may be abbreviated VEC.

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 data_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 mode.

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 CHAR ROTATE, CHAR SIZE, TEXT SIZE, CHAR SCALE, and MATRIX_2X2.

- WINDOW, EYE, FIELD_OF_VIEW, MATRIX_4X3, MATRIX_4X4, and LOOK 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.

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 a subsequent section of this module.)

- 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 an 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).

  If XFORM MATRIX is used, specifying a name at input <3> creates an operation node (4X4 matrix) with that name.

- Output <1> contains the transformed data. If ASCII PS 300 command information is desired for the host, connect this output directly to F:LIST (below). 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.)

The F:LIST Function

F:LIST converts the output of F:XFORMDATA into an ASCII PS 300 command string suitable for storage on the host computer (and for retransmission back to the PS 300). 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 F:XFORMDATA<1>.

- Output <1> contains the transformed data, reformatted as an ASCII PS 300 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 F:XFORMDATA<3> at the time of the request.

- Output <2> is a Boolean TRUE completion indicator. Refer to the last section of this module for a sample application of this completion indicator.

The ASCII command string from F:LIST may be sent to a host computer via HOST_MESSAGE. For details on HOST_MESSAGE, refer to the ***Function Summary*** in Volume 3A.

EXCLUDING CERTAIN VIEWING TRANSFORMATIONS

If WINDOW, EYE, FIELD_OF_VIEW, MATRIX_4X3, MATRIX_4X4, or LOOK 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.

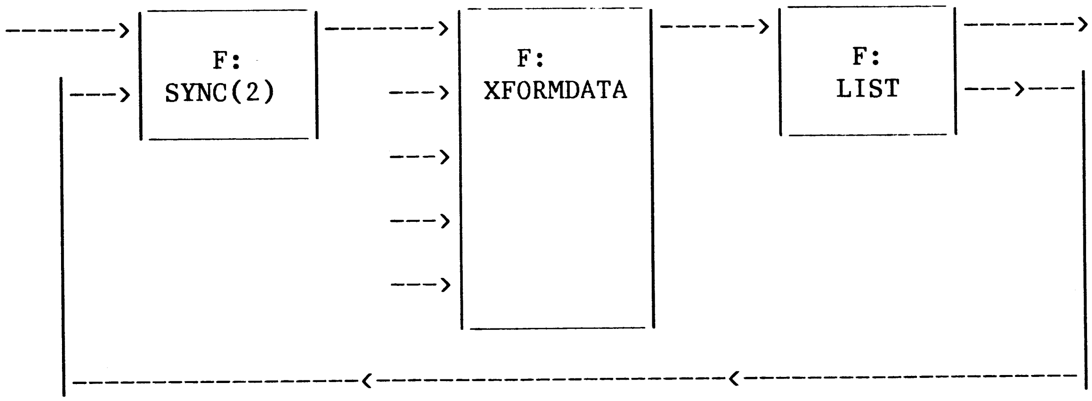
Since the default window transformation matrix is not an identity matrix, this practice is recommended even when no nodes for the above six transformations have been included explicitly in the display tree.

The example at the end of this module illustrates one way to exclude these viewing transformations while leaving others in effect during a transformed-data request.

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 <2>F:SYNC(2).

The use of this network is illustrated in the example at the end of this module.

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 VEC THEN Z;
Z := INSTANCE A,B,C,D;
A := VEC N=5 ... ;
B := VEC N=6 ... ;
C := VEC N=10 ... ;
D := VEC 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;
```

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,
                  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;
     CONN SYNC2<1>:<1>XFORMDATA;
     CONN XFORMDATA<1>:<1>LIST;
     CONN LIST<1>:<1>HOST_MESSAGE;  {Send transformed data to host}
     CONN 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 F:LIST's completion indicator to this input) to redisplay the object and continue designing).

SUMMARY

Transformed data can be retrieved from a given data node and then established as a separate data or operation node in the display tree. The transformed data can also be converted to an ASCII PS 300 command string for transmission to the host. To retrieve transformed data you must:

- Mark the data node by applying an XFORM VECTOR or XFORM MATRIX node in the display tree. 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 300 command string with an instance of the F:LIST function and send the data to the host via HOST_MESSAGE.

USING THE PS 340

RENDERING OPERATIONS FOR SURFACES AND SOLIDS

CONTENTS

RENDERING OPERATIONS

ILLUSTRATIONS

This module explains how to use the POLYGON command to define objects eligible for rendering and how to perform rendering operations on these objects. It is intended both as an introduction to rendering concepts and as a detailed statement of the rules for using the PS 340.

Objects composed of polygons defined by the POLYGON command are the only objects that are eligible for rendering operations. Other data-definition commands, such as VECTOR_LIST, CHARACTERS, LABELS, POLYNOMIAL, RATIONAL POLYNOMIAL, BSPLINE, and RATIONAL BSPLINE, do not establish objects which can be rendered. Their ordinary use, aside from rendering, is not affected.

There are two types of rendering operations: those applied to objects displayed on the calligraphic screen and those applied to objects displayed on the raster screen. Once an object has been correctly defined with the POLYGON command, it can be displayed on either the calligraphic or the raster screen without any modification to the data definition.

Calligraphic Renderings

Rendering operations on the calligraphic display can remove hidden line segments from an object, perform backface removal, section an object relative to a sectioning plane, and obtain a cross section.

Following are brief descriptions and examples of the rendering operations for the calligraphic display.

Hidden-Line Removal

Hidden–Line removal generates a view in which only the unobstructed portions of an object are displayed.

(Before Hidden–Line Removal)

(After Hidden–Line Removal)

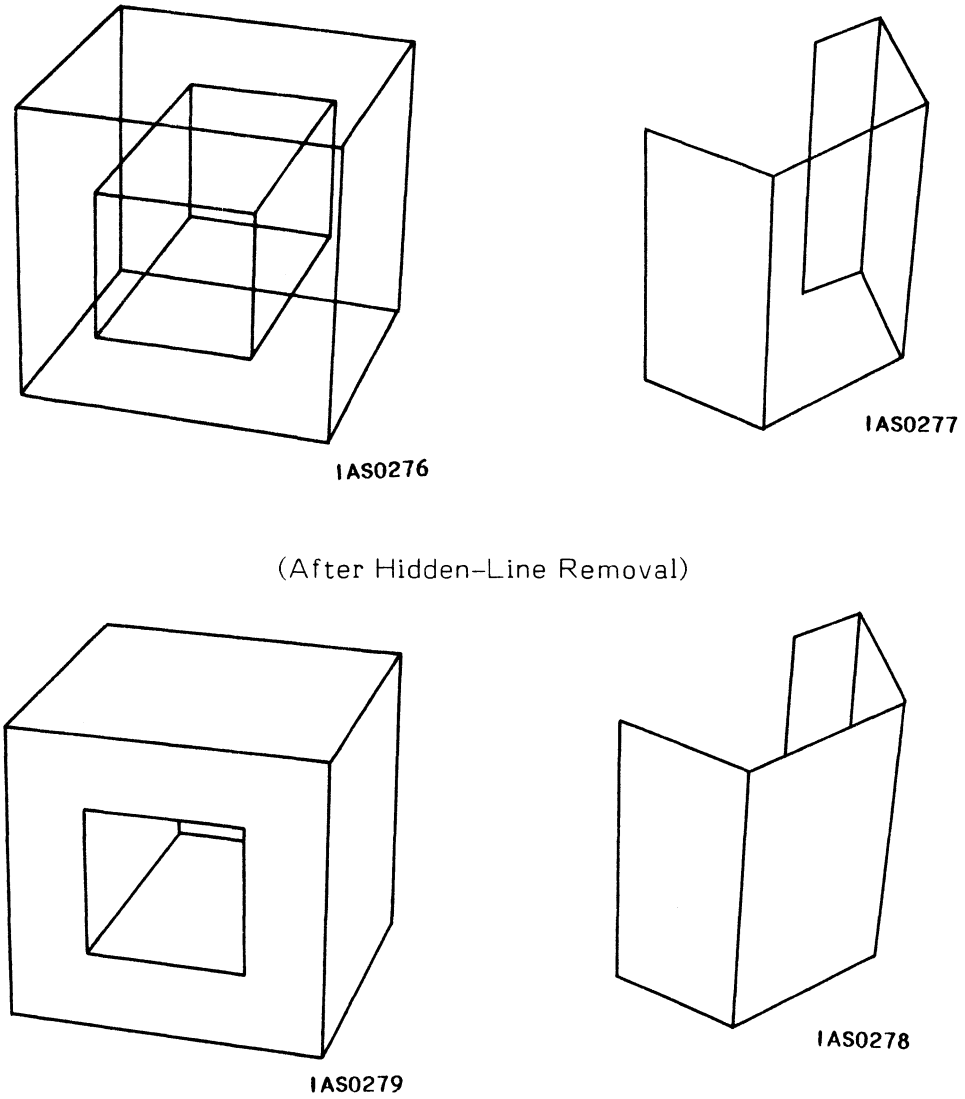
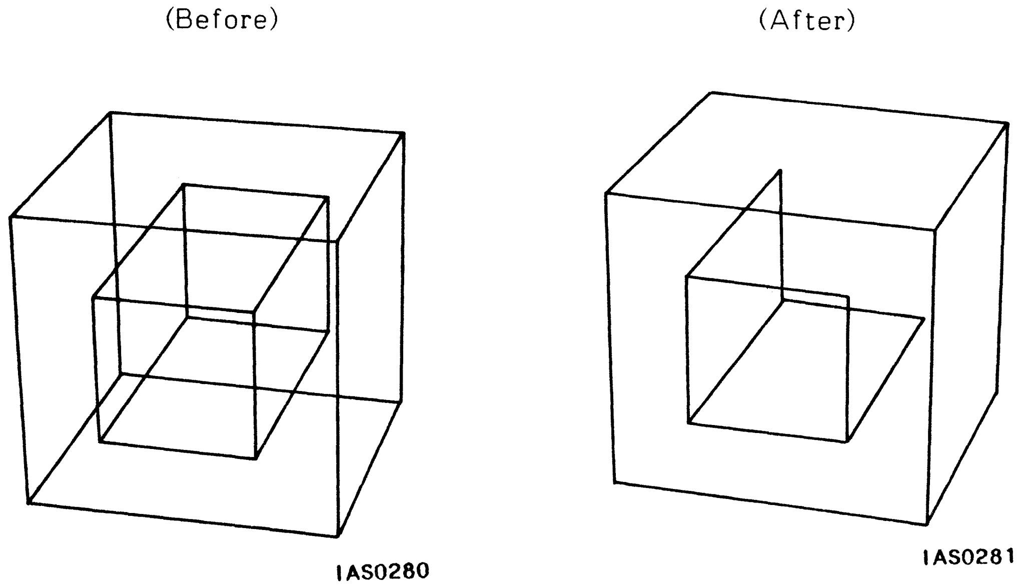


Figure 1. Object Before and After Hidden-Line Removal

Backface Removal

Backface removal is an intermediate step in hidden-line removal in which all polygons facing away from the viewer are removed.

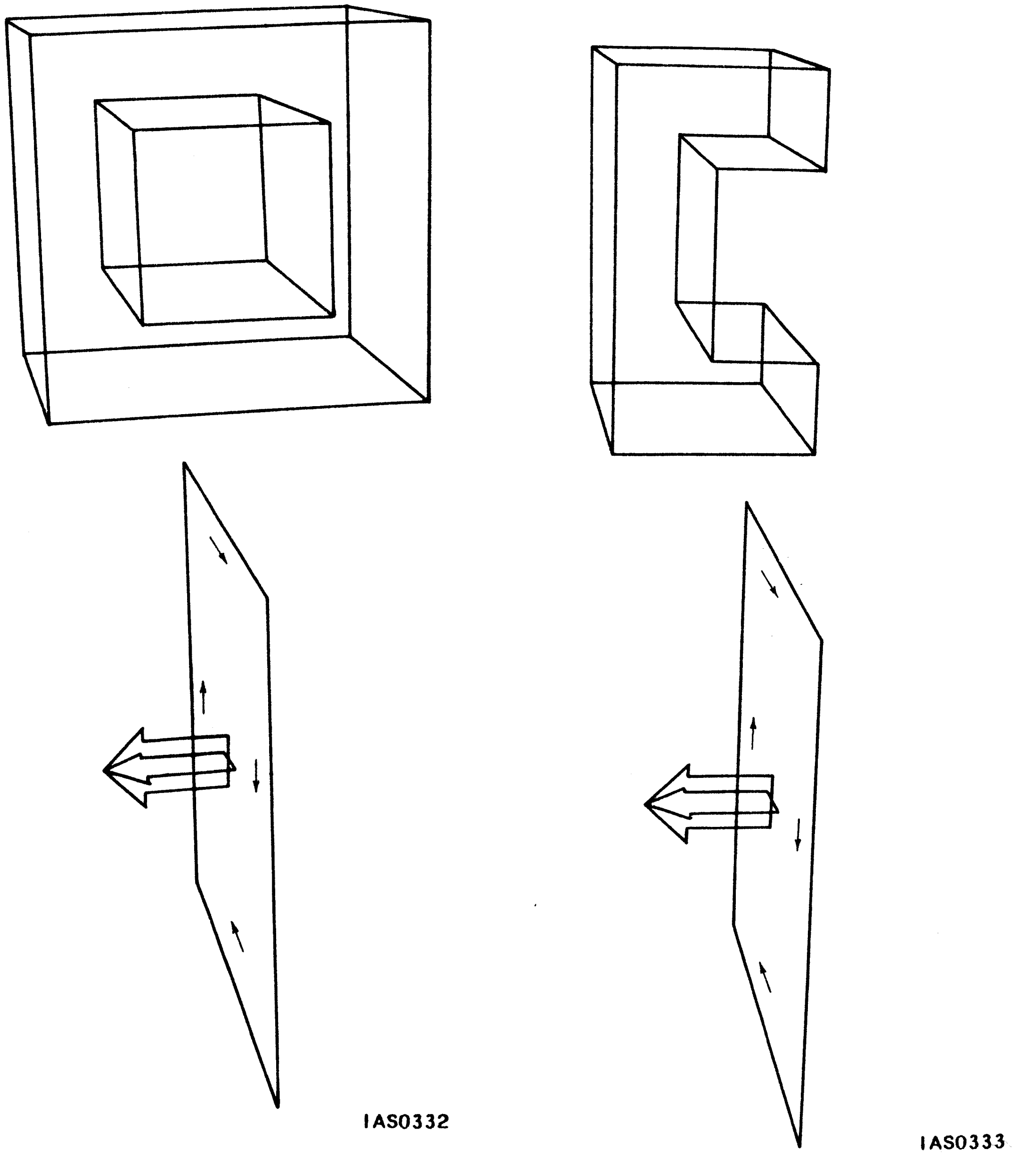

(Before)

(After)

Figure 2. Object Before and After Backface Removal

Sectioning

This operation makes use of a sectioning plane passing through the object which divides the object into two pieces. Upon sectioning, one piece is removed while the other is remains displayed. For solids, capping polygons are generated to maintain the integrity of the solid.


(Before)

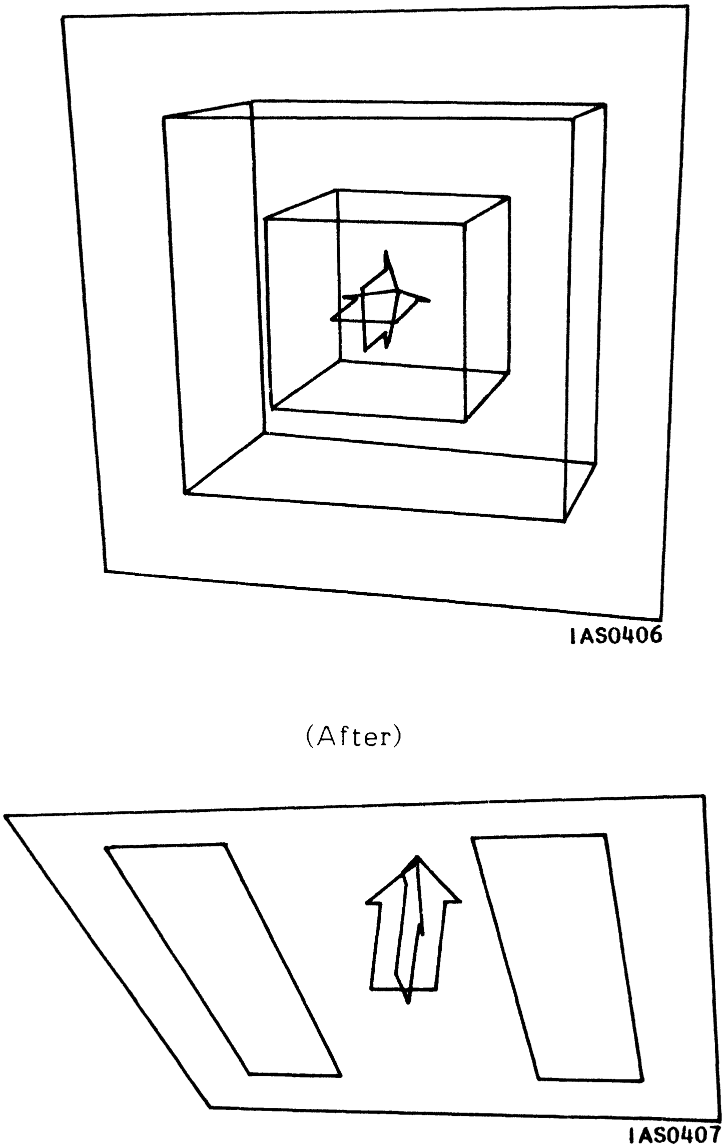
(After)

Figure 3. Object Before and After Sectioning

Cross Sectioning

The cross sectioning operation makes use of a defined sectioning plane to create a cross section of an object. When this operation is used, both sides of the object are discarded and only the slice defined by the sectioning plane remains.

(Before)



IAS0406

(After)

IAS0407

Figure 4. Object Before and After Cross Sectioning

Raster Renderings

Rendering operations that apply to objects on the raster screen are wash shading, flat shading, and smooth shading.

Wash shading produces an object with area-filled colored polygons ignoring normals, light sources, all lighting parameters, and all depth cueing parameters. This operation does not product objects that simulate a curved surface.

The ***flat shading*** process considers color, one light source and depth cueing to shade the polygons in the object accordingly. Flat shading can produce objects that simulate a faceted surface.

Smooth shading is the most complex process. The color of a polygon is varied across its surface, considering the normals at the polygon's vertices, the direction and color of various active light sources, the polygon's attributes (both color and highlights), and depth cueing. Objects that simulate a curved surface can be produced with smooth shading.

OBJECTIVES

After reading this module, you should be able to:

- Define a polygonal object with the POLYGON command using all the command options (COPLANAR, NORMALS, S, OUTLINE, WITH ATTRIBUTES).
- Establish a workspace in memory.
- Mark an object as a solid or a surface for rendering.
- Create a rendering.
- Save and compound a rendering.
- Display a shaded object on the raster screen and change the shading environment in which the object is displayed.

For those already familiar with the PS 340, a reference summary at the end of this module lists important rules and guidelines.

PREREQUISITES

Before reading this module, you should be familiar with programming the PS 300. It is helpful to have an understanding of the representation of polygonal objects in graphics applications. It is assumed that you have some method, such as an application program, to automatically generate polygonal data structures. If you will be using the Shading Firmware for the Raster System, it is assumed that you have some knowledge of the parameters used in shading objects for display on a raster screen.

DEFINING POLYGONAL OBJECTS

The first step in defining a polygonal object is to determine what it looks like. The next step is to determine the correct geometry to define that object in the world coordinate space. This would typically be done by an application program since determining the vertices of all the polygons of an object is too complex to do manually.

The polygons that make up an object to be rendered must be defined in the POLYGON command according to certain rules. If these rules are not followed, the results of a rendering operation applied to that object are unpredictable and usually incorrect, even though the object may appear correct when displayed.

A ***polygon*** is defined by the coordinates of its ***vertices.*** The ***edges*** of the polygon are defined by lines that connect those vertices. In the PS 340, a polygon must have at least three vertices and no more than 250, all of which must lie in the same plane. Ensuring that the vertices in a polygon are coplanar is the responsibility of the user.

Concave polygons are acceptable. ***Degenerate*** polygons (less than three vertices) and ***Interpenetrating*** polygons (intersecting themselves or others) are not acceptable. Polygons are not pickable and polygon data nodes have no inputs to allow them to be modified by function networks.

Using the Polygon Command

A ***polygon clause***, part of the POLYGON command, defines an individual polygon or face of an object by specifying the coordinates of its vertices. Since an object has many faces, several polygon clauses are used to define the entire object.

The syntax for the polygon clause is the word POLYGON and a set of x,y,z coordinates. The number of polygon clauses in the POLYGON command is equal to the number of polygons in the object. Each polygon in the object must be defined with a polygon clause.

A named group of one or more polygon clauses, with a semicolon at the end, constitutes a POLYGON data-definition command (or POLYGON command for short). This command defines the data node in the data structure of that object. There is no syntactical limit on the number of POLYGON clauses in the group. POLYGON may be abbreviated POLYG.

An option of the POLYGON command declares polygons to be coplanar, providing the capability to create objects with holes or protrusions. Other options allow you to define the color of the edges of polygons and to declare edges "soft" to simulate a curved surface on a calligraphic display.

There are additional POLYGON command options that associate characteristics or "attributes" with polygons for use in creating shaded images on a color raster screen. These options include color and the concentration of specular highlights. Normals can be specified for the vertices of an object to create a smooth-shaded image that simulates a curved surface. These options are shown below and explained briefly; complete details are discussed throughout this module.

Given,

```
    <vertex>  := [ S ] x,y,z [ N x,y,z ]
    <polygon> := [WITH [ATTRIBUTES name2] [OUTLINE h]]
                    POLYGON [COPLANAR] <vertex> ... <vertex>
```

The polygon command is:

```
[ Name := ] <Polygon> <Polygon> . . . <Polygon> ;
```

where:

- A vertex definition has the form [S] x,y,z [N x,y,z]

  where

  - S indicates that the edge drawn between the previous vertex and this one represents a soft edge of the polygon (discussed in detail later in this module). If the S specifier is used for the first vertex in a polygon definition, the edge connecting the last vertex with the first is soft.

  - N indicates a normal to the surface with each vertex of the polygon. Normals are used only in smooth-shaded renderings. Normals must be specified for all vertices of a polygon or for none of them. If no normals are given for a polygon, they are defaulted to the same as the plane equation for the polygon.

  - x, y, and z are coordinates in a left-handed Cartesian system.

- WITH ATTRIBUTES is an option that assigns the attributes defined by name2 for all polygons until superseded by another WITH ATTRIBUTES clause.

- WITH OUTLINE is an option that specifies the color of the edges of a polygon on the color CSM display, or their intensity on a black and white display.
- COPLANAR declares that the specified polygon and the one immediately preceding it have the same plane equation.

Constructing Surfaces and Solids

The PS 340 command language allows you to define two classes of polygons: ***surfaces*** and ***solids***. Solids enclose a volume of space, while surfaces do not.

Surfaces can have edges that belong to just one polygon. For example, in Figure 5, edge CD is a part of polygon 3 but not of any other polygon.

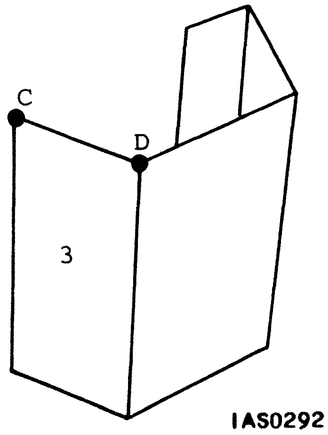


Figure 5. Surface Object

In a solid, each edge of each polygon must coincide with the edge of an adjacent polygon. For example, edge AB in Figure 6, is defined as part of polygon 1 and as part of polygon 2, and each edge of each polygon is similarly repeated in different polygons.

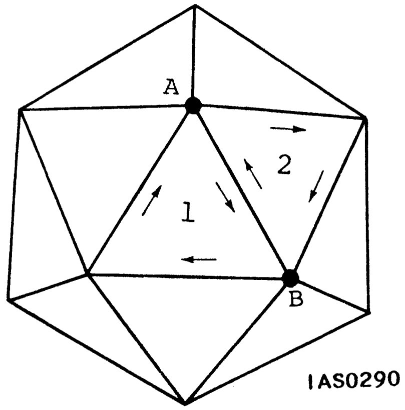


Figure 6. Solid Object

A solid cannot contain three or more polygons which have a single edge in common, although surfaces like the one in Figure 7 can:

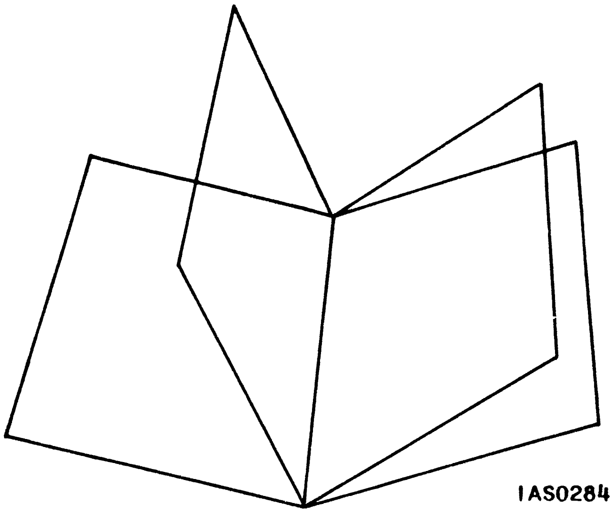


Figure 7. Surface With Three Common Edges

The nature of a polygonal object, representing a surface or a solid, is determined not only by the construction but by placing it beneath a rendering node in the PS 340 data structure created by the SOLID_RENDERING and SURFACE_RENDERING commands. These commands are discussed in detail in the section titled, "Marking Objects For Rendering."

Specifying Vertices

By definition, polygons are closed implicitly, so the first vertex is not repeated when defining a polygon. The system connects the last vertex given to the first vertex.

In solids, the direction in which the vertices are ordered within each polygon clause has important consequences for rendering operations. The vertices should be listed so that if you start at any vertex and move to the next vertex (as indicated by the order in the polygon clause), you are traveling around the edges of the polygon in a clockwise direction.

There are no similar restrictions for surfaces. The vertices of a surface can be listed in either a clockwise or a counterclockwise direction.

For example, let A (0,0,0), B (.5,.87,0) and C (1,0,0) be the vertices of one triangular face of an icosahedron as shown below.

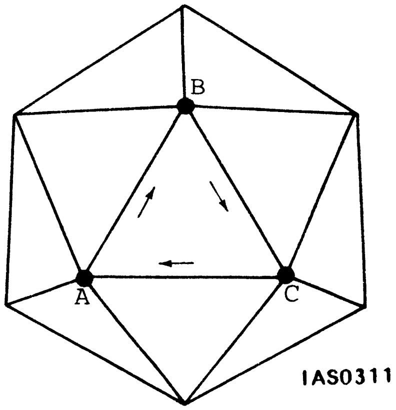


Figure 8. Icosahedron With Correct Vertex Ordering

Since the points A, B, and C have the arrangement indicated by the arrows when the triangular face is viewed from the outside of the icosahedron, that triangle could be defined correctly by any one of the following clauses, all of which specify the vertices in clockwise order:

```
... POLYGON 0,0,0 .5,.87,0 1,0,0 ...

... POLYGON .5,.87,0 1,0,0 0,0,0 ...

... POLYGON 1,0,0 0,0,0 .5,.87,0 ...
```

However, the following definition is incorrect for this polygonal face because it specifies the vertices in counterclockwise order:

```
... POLYGON 0,0,0 1,0,0 .5,.87,0 ...
```

Another method to determine the order of vertices is to use the right hand rule. The right hand rule states that if you point the thumb of your right hand towards the center of the object and rotate your fingers towards your wrist, the direction that your fingers move indicate the order in which the vertices of that polygon should be listed.

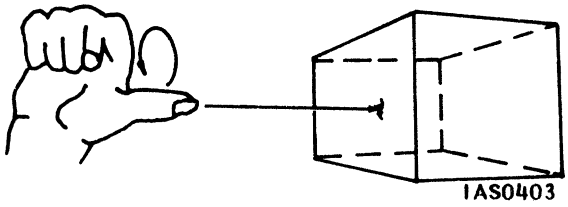


Figure 9. Right Hand Rule

Using arrows to show the vertex order of each polygon, a correctly constructed icosahedron looks like this:

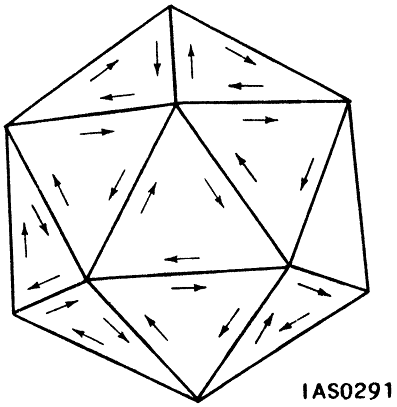


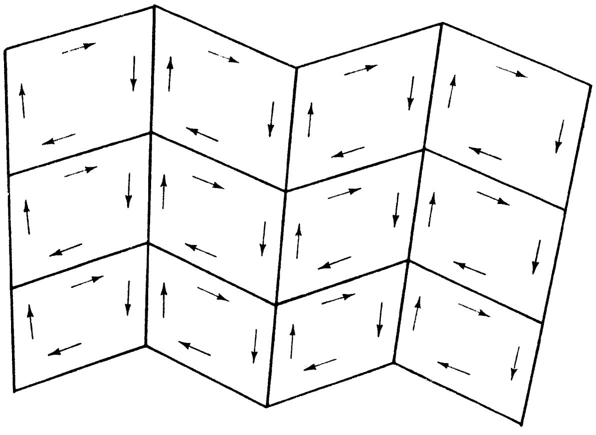
Figure 10. Correctly-Constructed Icosahedron

In all correctly defined solids, each edge is repeated in two different polygons. For each pair of adjacent polygons, their common edges run in opposite directions. Each edge is associated with a pair of opposing arrows in Figure 10. This is true for any edge of any correctly-defined solid, even if it contains inner contours. For solids, all vertices must run clockwise and all common edges of adjacent polygons must run in opposite directions.

For surfaces the vertex-ordering rule is less stringent. Vertices in surfaces do not have to be ordered in a clockwise fashion but they should be ordered so that common edges of adjacent polygons run in opposite directions.

For example, the edges should be ordered like this

(Correct)



not like this

(Incorrect)

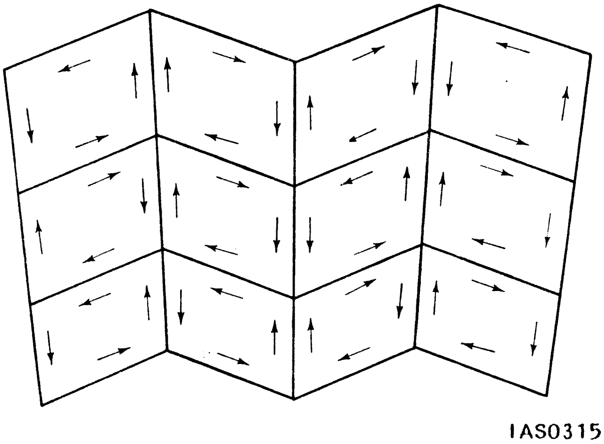


Figure 11. Correct and Incorrect Vertex Ordering for Surfaces

Although for surfaces it is not required that vertices run clockwise, it is a good idea to follow this rule when convenient since it allows surfaces to be easily "upgraded" to solids (especially if the surface has what could be called an interior). Assuming that polygon data are equally available in either form, it is better to have a surface's vertices in a clockwise order.

Given the following object (cube):

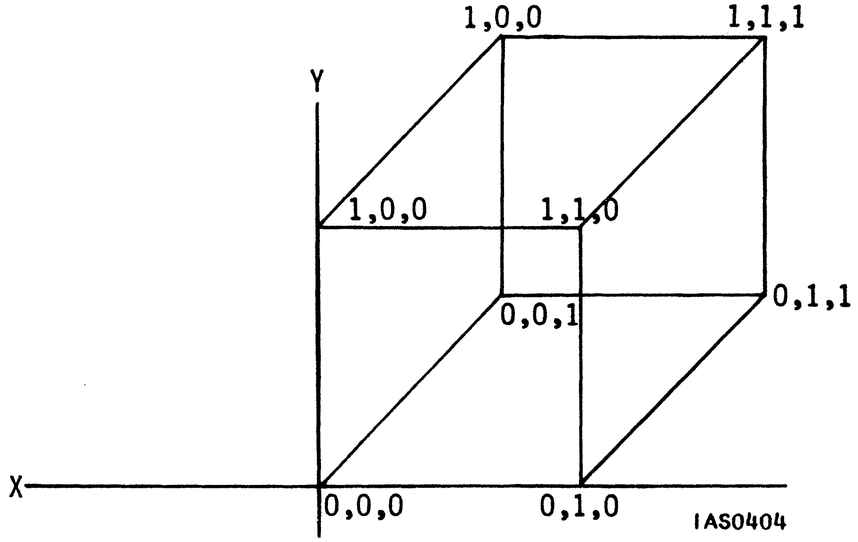


Figure 12. Cube

A correct syntax to define this object is as follows:

```
Cube :=    POLYGON 0,0,0  1,0,0  1,1,0  0,1,0
           POLYGON 0,1,0  1,1,0  1,1,1  0,1,1
           POLYGON 1,1,1  1,0,1  0,0,1  0,1,1
           POLYGON 1,0,1  0,0,1  0,0,0  1,0,0
           POLYGON 1,0,0  1,0,1  1,1,1  1,1,0
           POLYGON 0,1,0  0,1,1  0,0,1  0,0,0;
```

Associating Outer and Inner Contours With Coplanar

A polygon that represents a face of an object is called an *outer contour.* Some polygons, known as *inner contours* represent cavities, holes, or protrusion sites in an object.

For the PS 340 to interpret inner contours properly, two things must be done. One is to observe the vertex-ordering convention for inner and outer contours. The other is to use the *coplanar* option in the POLYGON clause to associate inner and outer contours.

The vertex ordering rule for inner and outer contours is as follows: vertices of inner contours must run in the opposite sense to the corresponding outer contour. For a solid this implies that the vertices of an inner contour run counterclockwise while outer contours run clockwise when viewed.

The vertices of the following triangular polygon face (outer contour) with a hole in it (inner contour) are ordered as follows.

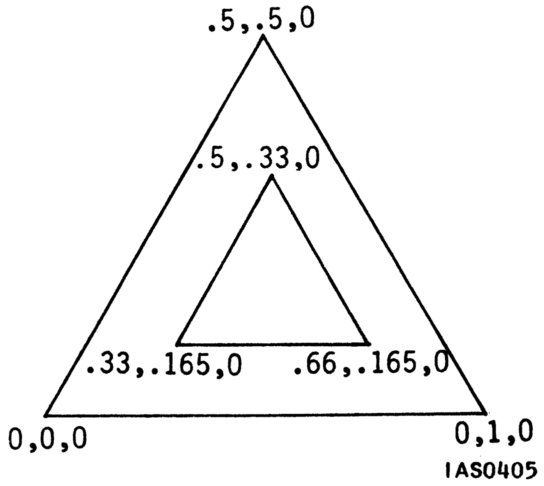


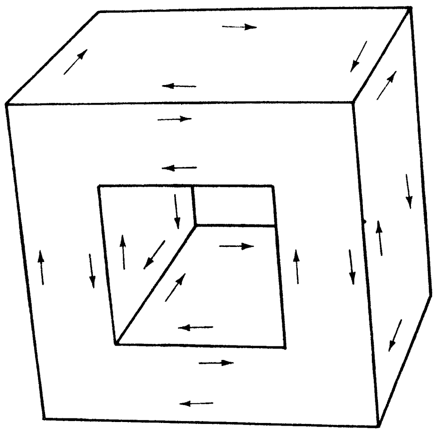
Figure 13. Surface With Inner/Outer Contours

A POLYGON command syntax for this object is :

```
Object :=  POLYGON  0,0,0  .5,.5,0  1,0,0 {outer contour}
           POLYGON COPLANAR .5,.33,0  .33,.165,0  .66,.165,0;
           {inner contour}
```

Note that the vertices for the inner contour in the above example are listed in the opposite order of those of the outer contour.

The following solid illustrates the rule that inner contours must run opposite to outer contours.



IAS0313

Figure 14. Solid With Correct Vertex Ordering

Rules for Using the Coplanar Option

An inner contour is always coplanar with some surrounding outer contour. To define an inner contour, you must associate it with the appropriate outer contour by declaring an inner contour to be coplanar with the outer contour. The COPLANAR specifier makes this declaration. COPLANAR is an option of the polygon clause which declares that the specified polygon and the one immediately preceding it have the same plane equation.

A polygon declared to be COPLANAR must lie in the same plane as the previous polygon if correct renderings are to be obtained. The system does not check for this condition.

All members of a set of consecutive COPLANAR polygons are taken to have the same plane equation. The polygon without a COPLANAR specifier immediately preceding the consecutive COPLANAR polygons is also taken to be in the set.

Polygons that are coplanar can be included in the polygon list without the COPLANAR specifier, but when outer and inner contours are being associated the COPLANAR clause is required.

If COPLANAR is specified for the first polygon in a polygon list, it has no effect.

It is legal to define two coplanar polygons without specifying COPLANAR, as long as the polygons are not to be associated as an outer/inner pair.

In the following example the second polygon is coplanar with the first polygon. The third polygon is not coplanar with either of the two preceding polygons.

```
Object :=  POLYGON -.6,-.6, -.6  -.6,.6,-.6  .6,.6,-.6  .6,-.6,-.6        {1}
           POLYGON COPLANAR  -.3,-.3,-.6  .3,-.3,-.6  .3,.3,-.6           {2}
           -.3,.3,-.6
           POLYGON  -.6,-.6,-.6  .6,-.6,-.6  .6,-.6,.6  -.6,-.6,.6        {3}
```

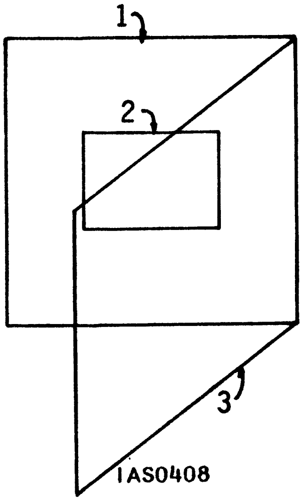


Figure 15. Object With Coplanar Polygon

In the next example, the first four polygons are coplanar with each other. The fifth polygon is not coplanar with any of the preceding polygons.

```
Object :=
  {outer}  POLYGON  1,1,0  1,0,0  -1,0,0  -1,1,0                          {1}
  {inner}  POLYGON COPLANAR .4,.8,0  -.4,.8,0  -.4,.4,0  .4,.4,0          {2}
  {inner}  POLYGON COPLANAR 1,0,0  1,-1,0  -1,-1,0  -1,0,0                {3}
  {inner}  POLYGON COPLANAR .4,-.4,0  -.4,-.4,0  -.4,-.8,0  .4,-.8,0      {4}
  {inner}  POLYGON 1,1,0  1,-1,0  1,-1,1  1,1-1                           {5}
```

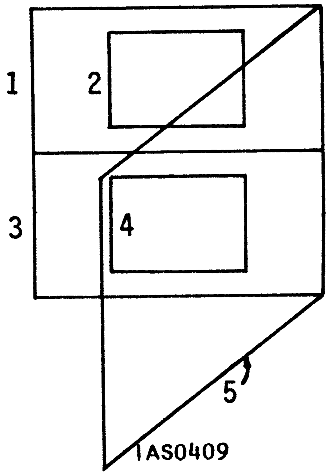


Figure 16. Object With Inner/Outer Contours

A solid object with a cavity usually includes an inner contour. In the following object, one triangle is an inner contour and all other polygons are outer contours, including the walls and back of the cavity. The back wall of the triangle is not an inner contour.

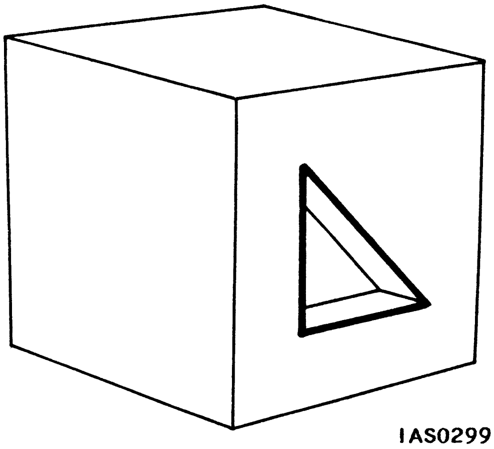


Figure 17. Solid With a Cavity

The POLYGON command syntax for this object follows. Notice that there is only one polygon declared COPLANAR, for the one inner contour on the object. The polygon declared coplanar (inner contour) comes after the polygon clause for the front face of the cube (outer contour).

```
Object :=
  {Cube faces}
  POLYGON -.6,-.6,.6  .6,-.6,.6  .6,.6,.6  -.6,.6,.6 {back}
  POLYGON -.6,-.6,-.6  .6,-.6,-.6  .6,-.6,.6  -.6,-.6,.6 {bottom}
  POLYGON .6,-.6,-.6  .6,.6,-.6  .6,.6,.6  .6,-.6,.6 {right}
  POLYGON .6,.6,-.6  -.6,.6,-.6  -.6,.6,.6  .6,.6,.6 {top}
  POLYGON -.6,.6,-.6  -.6,-.6,-.6  -.6,-.6,.6  -.6,.6,.6 {left}
  {Cube face containing cavity}
  POLYGON .6,.6,-.6 .6,-.6,-.6 -.6,-.6,-.6 -.6,.6,-.6
  {Cavity openings}
  POLYGON COPLANAR .6,.3,-.3  .6,-.3,-.3  .6,-.3,.3
  {Cavity side walls}
  POLYGON .6,.3,-.3  .6,-.3,-.3  .4,-.3,-.3  .4,.3,-.3
  POLYGON .6,-.3,-.3  .6,-.3,.3  .4,-.3,.3  .4,-.3,-.3
  POLYGON .6,-.3,.3  .6,.3,-.3  .4,.3,-.3  .4,-.3,.3
  {Cavity rear wall}
  POLYGON .4,.3,-.3  .4,-.3,.3  .4,-.3,-.3;
```

A polygon should not be defined as an inner contour unless it is coplanar with a surrounding contour. Tunnels, protrusions and holes do not need inner contours unless this coplanar arrangement is present. For example, in Figure 18 neither of the objects contains inner contours.

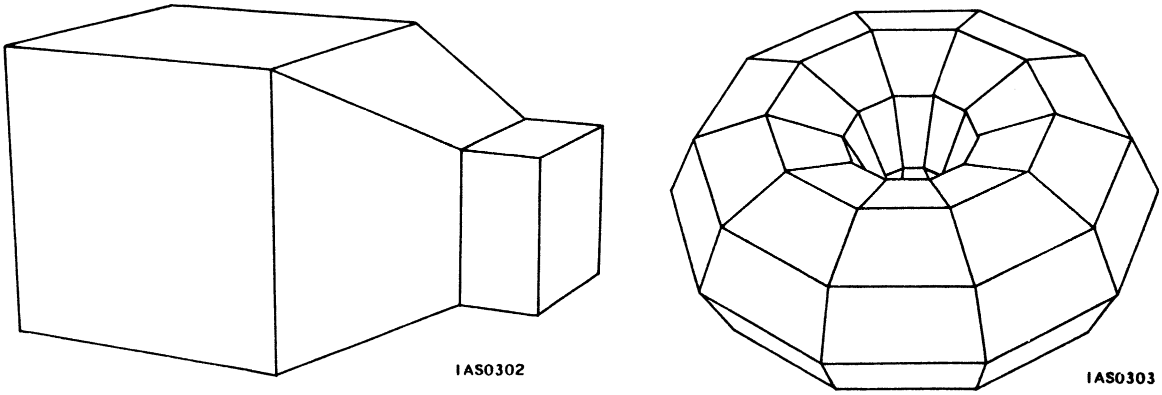
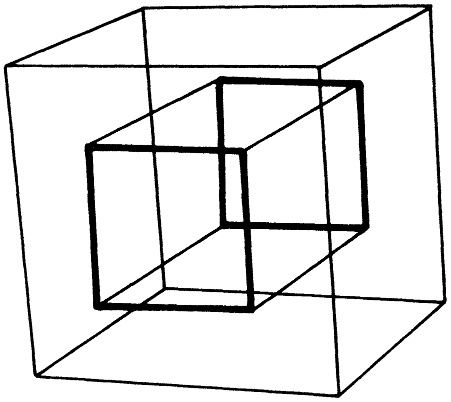


Figure 18. Solid Without Inner Contours

A cube with a tunnel running through it has two inner contours in its polygon definition, one for each opening of the tunnel.



IAS0300

Figure 19. Cube With a Tunnel

A POLYGON command syntax for this object is:

```
Object :=
POLYGON  -.6,-.6,.6  .6,-.6,.6  .6,.6,.6  -.6,.6,.6
POLYGON COPLANAR  -.3,-.3,.6  -.3,.3,.6  .3,.3,.6  .3,-.3,.6
POLYGON  -.6,-.6,-.6  .6,-.6,-.6  .6,-.6,.6  -.6,-.6,.6
POLYGON  .6,-.6,-.6  .6,.6,-.6  .6,.6,.6  .6,-.6,.6
POLYGON  .6,.6,-.6  -.6,.6,-.6  -.6,.6,.6  .6,.6,.6
POLYGON  -.6,.6,-.6  -.6,-.6,-.6  -.6,-.6,.6  -.6,.6,.6
POLYGON .6,.6,-.6 .6,-.6,-.6 -.6,-.6,-.6 -.6,.6,-.6
POLYGON COPLANAR  -.3,.3,-.6  -.3,-.3,-.6  .3,-.3,-.6  .3,.3,-.6
POLYGON  -.3,-.3,-.6  -.3,-.3,.6  .3,-.3,.6  .3,-.3,-.6
POLYGON  .3,.3,-.6  .3,-.3,-.6  .3,-.3,.6  .3,.3,.6
POLYGON  .3,.3,-.6  .3,.3,.6  -.3,.3,.6  -.3,.3,-.6
POLYGON  -.3,.3,-.6  -.3,.3,.6  -.3,-.3,.6  -.3,-.3,-.6;
```

Objects with inner contours can often be defined without inner contours. For example, the outer/inner contour pair in this object could be replaced by four coplanar outer contours.

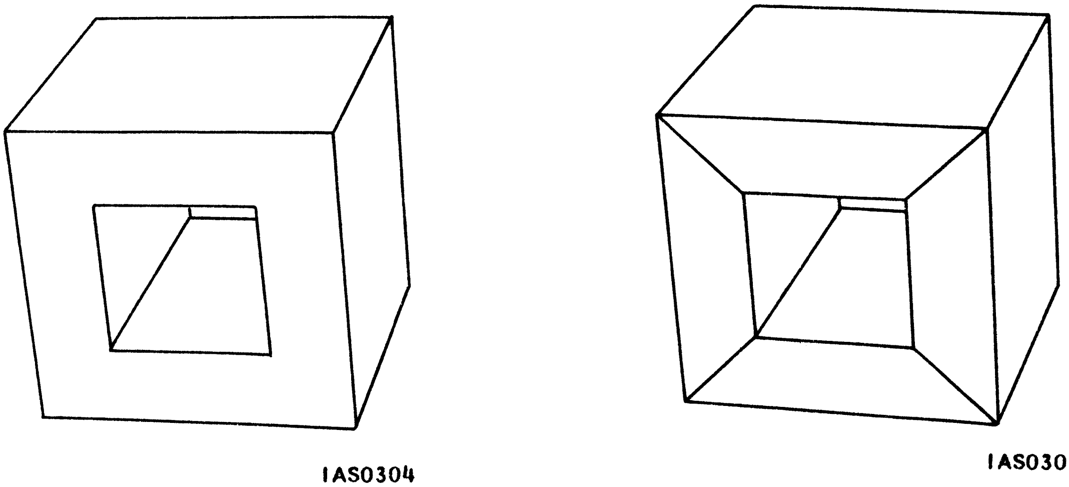


Figure 20. Objects With Coplanar Outer Contours

Both objects are admissible and can be rendered correctly. However, all other things being equal, *the object with declared inner contours is processed more efficiently.*

For correct renderings, polygons may not intersect other polygons. (This prohibition extends to polygons which just coincide, since numerical precision constraints may result in the polygons intersecting.) For example, consider the following solid, which contains a protrusion:

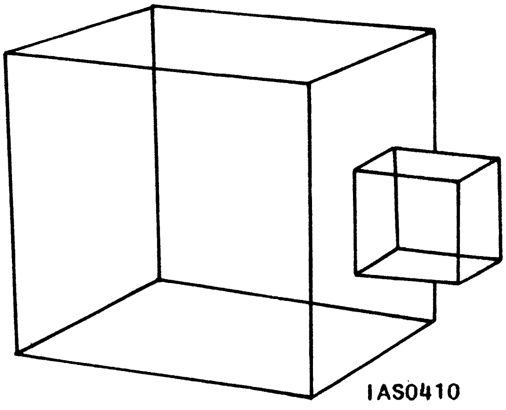


Figure 21. Solid With Protrusion

It seems that this solid can be constructed by putting two cubes together.

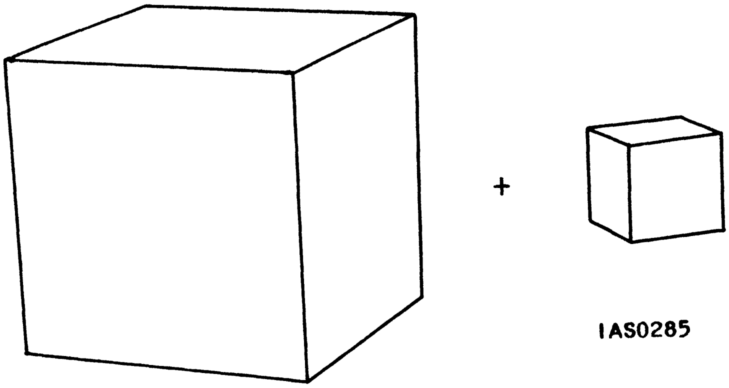


Figure 22. Solid Composed of Two Cubes

However, this is incorrect because one face of the small cube coincides with a portion of a face of the large cube. Another way of attempting to construct this solid fails for the same reason (Figure 23).

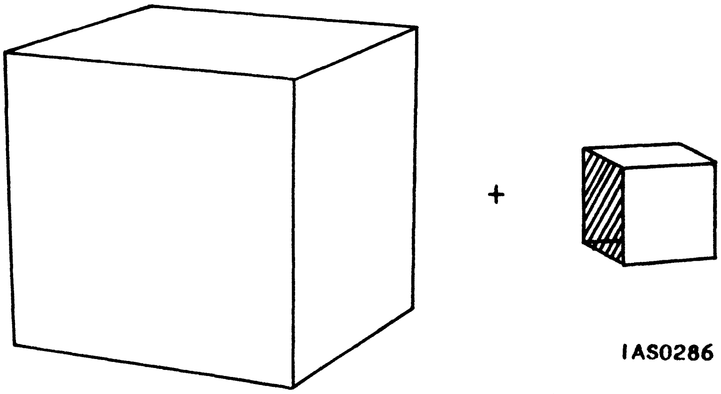


Figure 23. Invalid Solid

In this formulation, four edges of the small box coincide with the interior of a larger polygon making the solid invalid. Also, these edges violate the requirement for solids that each polygon edge coincide with the edge of another polygon.

The correct construction of this solid requires an inner contour at the site of the protrusion (Figure 24).

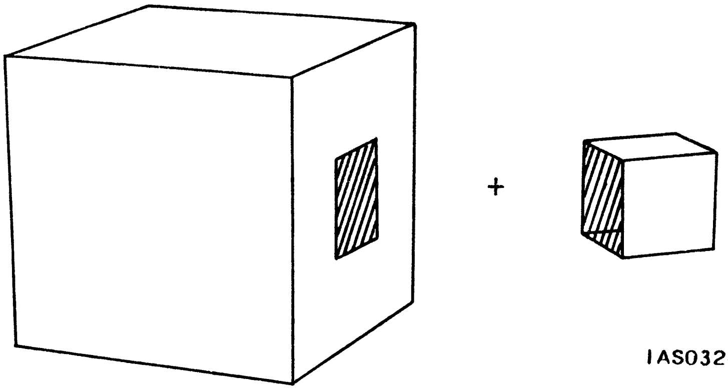


Figure 24. Correct Solid Construction

In the correct construction of the solid cube with a smaller cube as a protrusion, each edge of each polygon coincides with another polygon edge, and no portions of any polygon intersects the interior of any polygon. Only this construction guarantees correct renderings.

A correct POLYGON command to define this object would be as follows.

```
CUBEPROT :=
    POLYGON  -.3,-.3, .9    -.3,-.3, .6    .3,-.3, .6     .3,-.3, .9
    POLYGON   .3,-.3, .9     .3,-.3, .6    .3, .3, .6     .3, .3, .9
    POLYGON   .3, .3, .9     .3, .3, .6   -.3, .3, .6    -.3, .3, .9
    POLYGON  -.3, .3, .9    -.3, .3, .6   -.3,-.3, .6    -.3,-.3, .9
    POLYGON  -.6,-.6,-.6    -.6, .6,-.6    .6, .6,-.6     .6,-.6,-.6
    POLYGON   .3,-.3, .9     .3, .3, .9   -.3, .3, .9    -.3,-.3, .9
    POLYGON  -.6,-.6, .6     .6,-.6, .6    .6, .6, .6    -.6, .6, .6
    POLYGON COPLANAR
             -.3, .3, .6     .3, .3, .6    .3,-.3, .6    -.3,-.3, .6
    POLYGON  -.6,-.6,-.6     .6,-.6,-.6    .6,-.6, .6    -.6,-.6, .6
    POLYGON   .6,-.6,-.6     .6, .6,-.6    .6, .6, .6     .6,-.6, .6
    POLYGON   .6, .6,-.6    -.6, .6,-.6   -.6, .6, .6     .6, .6, .6
    POLYGON  -.6, .6,-.6    -.6,-.6,-.6   -.6,-.6, .6    -.6, .6, .6;
```

Failure to use the coplanar specifier in the polygons clause can result in incorrect hidden-line renderings.

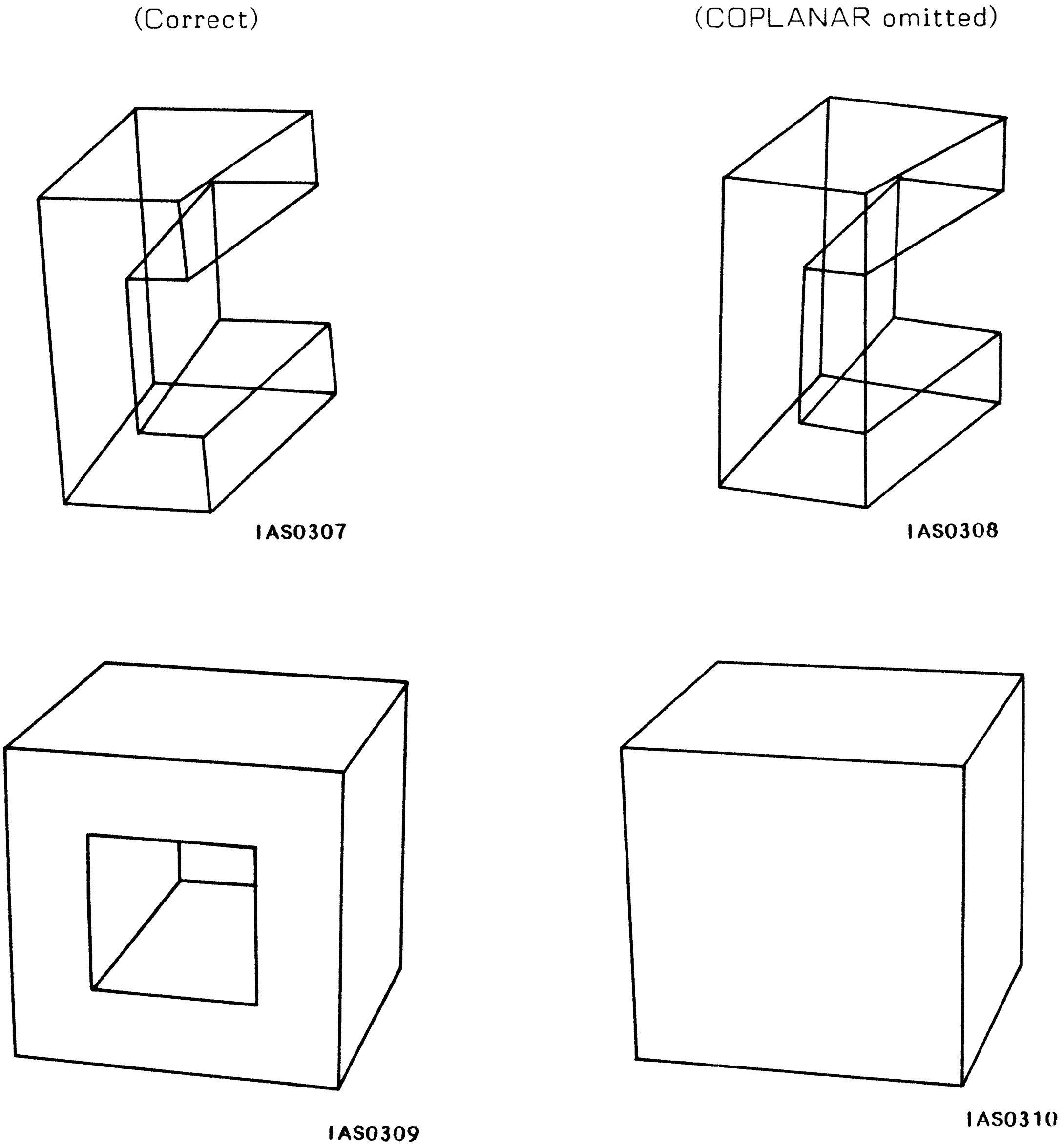
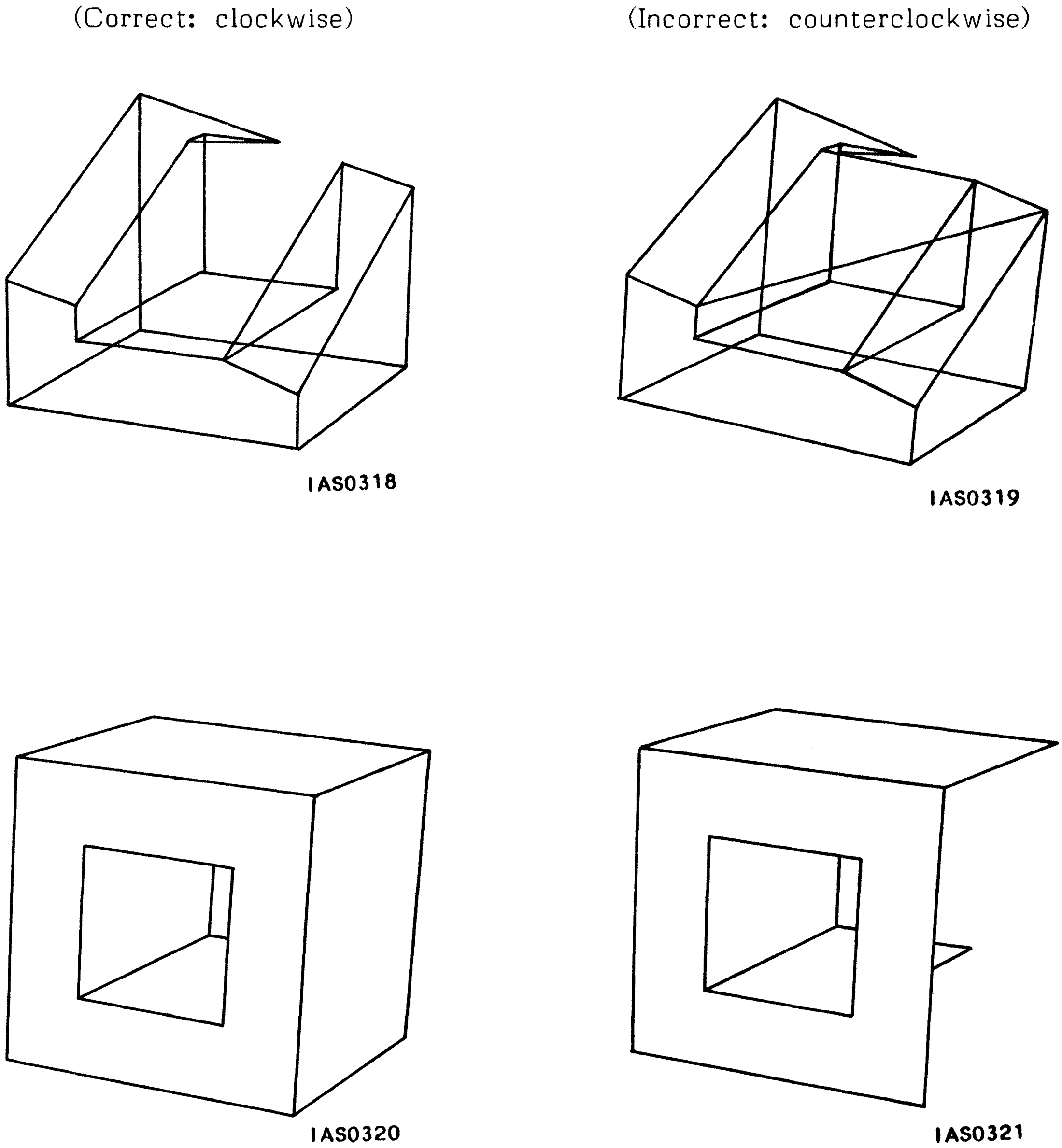


Figure 25. Hidden-Line Renderings of Objects Without the Coplanar Specifier

In solids, misplaced capping polygons and extra missing lines are often traceable to an outer contour defined with the wrong vertex order.



(Correct: clockwise)

(Incorrect: counterclockwise)

Figure 26. Objects With Incorrect Vertex Ordering

Defining Soft Edges

Soft edges, declared with the "S" specifier in the polygon clause, are invisible in hidden-line renderings except when they make up part of the profile of an object (or silhouette). They can, therefore, be used to approximate curved surfaces in hidden-line renderings.

For example, suppose that the twelve vertical edges in this object are soft edges.

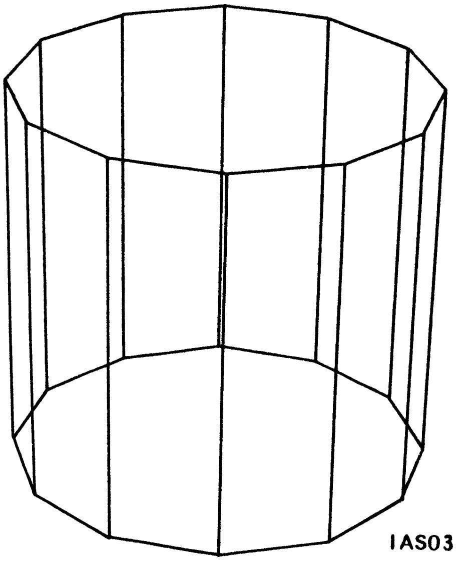


Figure 27. Solid With Declared Soft Edges

In a hidden-line rendering of this object, all soft edges become invisible, except for the two that contribute to the object's silhouette or profile. The result is an approximation of a cylinder's curved surface without intrusive edges which were provided for construction purposes only.

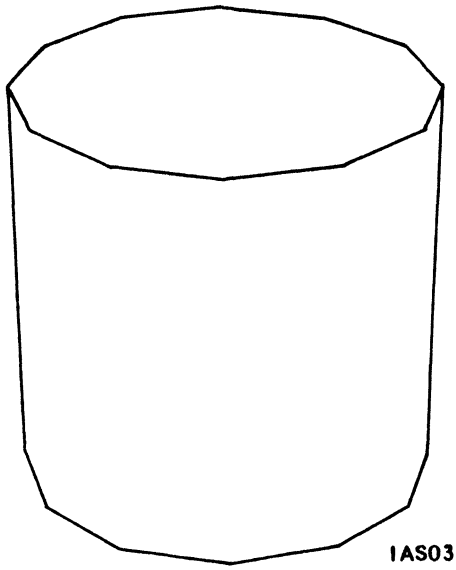


Figure 28. Rendering of Solids With Soft Edges

The "S" option before a set of X,Y,Z coordinates indicates that the edge drawn between the previous vertex and this one represents a soft edge of the polygon. If "S" is placed before the first set of X,Y,Z coordinates in a polygon clause, the edge connecting the last vertex with the first is soft.

When using the "S" specifier in the POLYGON command to define an object, there are some rules to remember about the way the system treats edges that are declared to be soft.

An "S" specifier causes the system to apply a positioning operation rather than a drawing operation to the associated polygon vertex. Therefore, if a single polygon containing a soft edge is displayed, the soft edge is "invisible" on the display.

Each polygon edge in a solid coincides with an edge of a neighboring polygon so that the solid is made up of common-edge pairs. If one edge of a common-edge pair is declared as soft, and the other is declared as "hard," ***the system considers the entire common edge pair to be soft in creating a hidden line rendering.*** This convention allows a solid's entire structure to be visible in its original view (since one edge in the pair is hard), but invisible in a hidden-line rendering. This is generally the way soft edges are defined. It is possible to define both edges of a common-edge pair as soft; in which case the common-edge pair would be invisible even in the original object.

In surfaces, polygon edges lying on the outline do not coincide with any neighboring polygon edges. All other polygon edges do belong to common-edge pairs, and it is only these "interior" edges which would be made invisible in a representation of a curved surface. In surfaces as well as solids, soft edges should be members of common-edge pairs, and only one edge need be declared soft.

In drawing a "hard" common-edge pair, the system line generator system strokes the same vector twice. If one member of the pair is soft, the vector is only stroked once; the result is slightly dimmer. This intensity variation indicates which edges of an unrendered object are soft. Hardcopies of objects containing edges will not show an intensity variation.

Remember the vertex ordering rule for polygons. Common-edge pairs should always run in opposite directions. This is especially important when one edge is soft. Otherwise, profile edges may be invisible in hidden-line renderings.

Defining Color and Intensity for Vector Displays

The color of the edges of a polygon on the CSM color display, or the intensity on a monochrome display, is set by the optional WITH OUTLINE ***h*** clause in the POLYGON command. (This has no effect on objects displayed on the raster screen.) The characteristics defined by the WITH OUTLINE clause apply to all subsequent polygons in the node until superseded by another WITH OUTLINE clause. The WITH OUTLINE clause comes before the word POLYGON in the polygon clause.

The parameter ***h*** sets the intensity or color, but how this parameter is interpreted is controlled by the presence or absence of a SET COLOR BLENDING node higher in the structure. For the rendered view to be displayed in the same form, the SET COLOR BLENDING node must be at a higher hierarchical level than the rendering operate node.

If h=0 or is in the range $1<h\leq360$, it will be inserted in the structure in a form suitable for interpretation as a hue (as in color blending for vectors); however, if $0\leq h\leq1$, the value is inserted in such a way as to be properly interpreted as intensity. If the SET COLOR BLENDING node is absent for the larger values of h, or present for smaller, the results are unspecified.

Color or intensity are specified for complete polygons, not individual edges. The hue (or intensity) of the capping polygon in a sectioning operation is inherited from the color (or intensity) of the sectioning plane. The default color is blue. The default intensity is 1.

You cannot specify white polygons on the CSM color display, unless they are all white. Also, there may be strange color effects if polygons sharing a common edge are colored differently. The intention of the SET COLOR BLENDING node and the WITH OUTLINE clause is to allow the use of color to distinguish different bodies or parts of bodies, such as protrusions.

Following is a command sequence using the WITH OUTLINE clause to define an object with color.

```
INIT disp;
DISP a;
a:= SET CONTRAST 0 THEN b;
b:= SET COLOR BLENDING 1 THEN c;
c := ROT Y 30 THEN Twosquares;
Twosquares :=
   WITH OUTLINE 120  POLYGON        {gives the square a red outline}
   -1,1,0 0,1,0 0,-1,0 -1,-1,0
   WITH OUTLINE 240  POLYGON        {gives the green a red outline}
   0,1,0 1,1,0 1,-1,0 0,-1,0;
```

Defining Color and Highlights for Raster Displays

Specifying the color, diffuse reflection, and specular highlights, (called attributes) of a polygon in the raster image is done via the WITH ATTRIBUTES clause of the POLYGON command.

The ATTRIBUTES command creates a named attribute node in mass memory that defines specific qualities to be applied to polygons when referenced by the polygon data structure. The attributes specified in a WITH ATTRIBUTES Name2 clause of a polygon command apply to all subsequent polygons until superseded by another WITH ATTRIBUTES clause. If no WITH ATTRIBUTES option is given for a polygon node, default attributes are assumed. The default attributes are 0,0,1 for color, 0.75 for diffuse, and 4 for specular.

Given the polygon syntax:

```
[ name := ] <polygon> <polygon> . . . <polygon> ;
```

the attributes option is,

```
<polygon> := [WITH [ATTRIBUTES name2] [OUTLINE h]] polygon
<vertex>...<vertex>
```

The WITH ATTRIBUTES clause and the ATTRIBUTES command are explained in the "Displaying Shaded Images" section.

Specifying Normals

When a polygon is used to approximate a curved surface, the smooth appearance of the surface can be restored in a smooth-shaded rendering by approximating a surface using normals. Normals only apply to shaded renderings. A normal to the surface is given with each vertex of the polygon specified N X,Y,Z. The shaded-rendering process interpolates between these normals when rendering the polygon. Normals must be specified for all vertices of a polygon or for none of them. If no normals are given for a polygon, they are defaulted to the same as the normals of the plane in which the polygon lies. Normals are needed only in smooth-shaded renderings and should usually be used. If you do not use normals and request a smooth-shaded rendering, the result will be a flat-shaded rendering (except that specular and diffuse attributes will apply).

The following is an example of a cylinder with the normals specified. Notice that the first two polygons do not have normals so the normals default to the polygon normal and no smoothing is done across these. These are the ends of the cylinder. The rule is all polygons do not need to have normals (in which case they default to the plane equation), but if any vertex of a polygon has a normal then all vertices for the polygon must. The cylinder also has soft edges (for display on the calligraphic display).

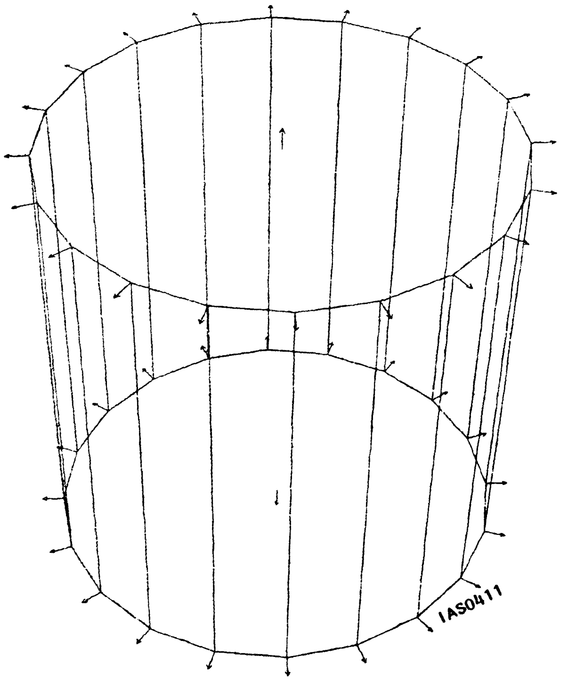


Figure 29. Cylinder With Normals and Soft Edges Specified

```
CYLINDER :=

  POLYGON
      1.00000,      0.00000,      1.00000
      0.95106,      0.30902,      1.00000
      0.80902,      0.58779,      1.00000
      0.58779,      0.80902,      1.00000
      0.30902,      0.95106,      1.00000
      0.00000,      1.00000,      1.00000
     -0.30902,      0.95106,      1.00000
     -0.58779,      0.80902,      1.00000
     -0.80902,      0.58779,      1.00000
     -0.95106,      0.30902,      1.00000
     -1.00000,      0.00000,      1.00000
     -0.95106,     -0.30902,      1.00000
     -0.80902,     -0.58779,      1.00000
     -0.58779,     -0.80902,      1.00000
     -0.30902,     -0.95106,      1.00000
      0.00000,     -1.00000,      1.00000
      0.30902,     -0.95106,      1.00000
      0.58779,     -0.80902,      1.00000
      0.80902,     -0.58779,      1.00000
      0.95106,     -0.30902,      1.00000
```

```
POLYGON
    0.95106,     -0.30902,     -1.00000
    0.80902,     -0.58779,     -1.00000
    0.58779,     -0.80902,     -1.00000
    0.30902,     -0.95106,     -1.00000
    0.00000,     -1.00000,     -1.00000
   -0.30902,     -0.95106,     -1.00000
   -0.58779,     -0.80902,     -1.00000
   -0.80902,     -0.58779,     -1.00000
   -0.95106,     -0.30902,     -1.00000
   -1.00000,      0.00000,     -1.00000
   -0.95106,      0.30902,     -1.00000
   -0.80902,      0.58779,     -1.00000
   -0.58779,      0.80902,     -1.00000
   -0.30902,      0.95106,     -1.00000
    0.00000,      1.00000,     -1.00000
    0.30902,      0.95106,     -1.00000
    0.58779,      0.80902,     -1.00000
    0.80902,      0.58779,     -1.00000
    0.95106,      0.30902,     -1.00000
    1.00000,      0.00000,     -1.00000

POLYGON
S   1.00000,    0.00000,    -1.00000    N  1.00000,    0.00000,    0.00000
    0.95106,    0.30902,    -1.00000    N  0.95106,    0.30902,    0.00000
S   0.95106,    0.30902,     1.00000    N  0.95106,    0.30902,    0.00000
    1.00000,    0.00000,     1.00000    N  1.00000,    0.00000,    0.00000

POLYGON
    0.95106,    0.30902,    -1.00000    N  0.95106,    0.30902,    0.00000
    0.80902,    0.58779,    -1.00000    N  0.80902,    0.58779,    0.00000
S   0.80902,    0.58779,     1.00000    N  0.80902,    0.58779,    0.00000
    0.95106,    0.30902,     1.00000    N  0.95106,    0.30902,    0.00000

POLYGON
    0.80902,    0.58779,    -1.00000    N  0.80902,    0.58779,    0.00000
    0.58779,    0.80902,    -1.00000    N  0.58779,    0.80902,    0.00000
S   0.58779,    0.80902,     1.00000    N  0.58779,    0.80902,    0.00000
    0.80902,    0.58779,     1.00000    N  0.80902,    0.58779,    0.00000

POLYGON
    0.58779,    0.80902,    -1.00000    N  0.58779,    0.80902,    0.00000
    0.30902,    0.95106,    -1.00000    N  0.30902,    0.95106,    0.00000
S   0.30902,    0.95106,     1.00000    N  0.30902,    0.95106,    0.00000
    0.58779,    0.80902,     1.00000    N  0.58779,    0.80902,    0.00000
```

```
POLYGON
    0.30902,      0.95106,     -1.00000    N  0.30902,      0.95106,      0.00000
    0.00000,      1.00000,     -1.00000    N  0.00000,      1.00000,      0.00000
S   0.00000,      1.00000,      1.00000    N  0.00000,      1.00000,      0.00000
    0.30902,      0.95106,      1.00000    N  0.30902,      0.95106,      0.00000

POLYGON
    0.00000,      1.00000,     -1.00000    N  0.00000,      1.00000,      0.00000
   -0.30902,      0.95106,     -1.00000    N -0.30902,      0.95106,      0.00000
S  -0.30902,      0.95106,      1.00000    N -0.30902,      0.95106,      0.00000
    0.00000,      1.00000,      1.00000    N  0.00000,      1.00000,      0.00000,

POLYGON
   -0.30902,      0.95106,     -1.00000    N -0.30902,      0.95106,      0.00000
   -0.58779,      0.80902,     -1.00000    N -0.58779,      0.80902,      0.00000
S  -0.58779,      0.80902,      1.00000    N -0.58779,      0.80902,      0.00000
   -0.30902,      0.95106,      1.00000    N -0.30902,      0.95106,      0.00000

POLYGON
   -0.58779,      0.80902,     -1.00000    N -0.58779,      0.80902,      0.00000
   -0.80902,      0.58779,     -1.00000    N -0.80902,      0.58779,      0.00000
S  -0.80902,      0.58779,      1.00000    N -0.80902,      0.58779,      0.00000
   -0.58779,      0.80902,      1.00000    N -0.58779,      0.80902,      0.00000

POLYGON
   -0.80902,      0.58779,     -1.00000    N -0.80902,      0.58779,      0.00000
   -0.95106,      0.30902,     -1.00000    N -0.95106,      0.30902,      0.00000
S  -0.95106,      0.30902,      1.00000    N -0.95106,      0.30902,      0.00000
   -0.80902,      0.58779,      1.00000    N -0.80902,      0.58779,      0.00000

POLYGON
   -0.95106,      0.30902,     -1.00000    N -0.95106,      0.30902,      0.00000
   -1.00000,      0.00000,     -1.00000    N -1.00000,      0.00000,      0.00000
S  -1.00000,      0.00000,      1.00000    N -1.00000,      0.00000,      0.00000
   -0.95106,      0.30902,      1.00000    N -0.95106,      0.30902,      0.00000

POLYGON
   -1.00000,      0.00000,     -1.00000    N -1.00000,      0.00000,      0.00000
   -0.95106,     -0.30902,     -1.00000    N -0.95106,     -0.30902,      0.00000
S  -0.95106,     -0.30902,      1.00000    N -0.95106,     -0.30902,      0.00000
   -1.00000,      0.00000,      1.00000    N -1.00000,      0.00000,      0.00000

POLYGON
   -0.95106,     -0.30902,     -1.00000    N -0.95106,     -0.30902,      0.00000
   -0.80902,     -0.58779,     -1.00000    N -0.80902,     -0.58779,      0.00000
S  -0.80902,     -0.58779,      1.00000    N -0.80902,     -0.58779,      0.00000
   -0.95106,     -0.30902,      1.00000    N -0.95106,     -0.30902,      0.00000
```

```
POLYGON
     -0.80902,   -0.58779,   -1.00000   N -0.80902,   -0.58779,   0.00000
     -0.58779,   -0.80902,   -1.00000   N -0.58779,   -0.80902,   0.00000
   S -0.58779,   -0.80902,    1.00000   N -0.58779,   -0.80902,   0.00000
     -0.80902,   -0.58779,    1.00000   N -0.80902,   -0.58779,   0.00000

POLYGON
     -0.58779,   -0.80902,   -1.00000   N -0.58779,   -0.80902,   0.00000
     -0.30902,   -0.95106,   -1.00000   N -0.30902,   -0.95106,   0.00000
   S -0.30902,   -0.95106,    1.00000   N -0.30902,   -0.95106,   0.00000
     -0.58779,   -0.80902,    1.00000   N -0.58779,   -0.80902,   0.00000

POLYGON
     -0.30902,   -0.95106,   -1.00000   N -0.30902,   -0.95106,   0.00000
      0.00000,   -1.00000,   -1.00000   N  0.00000,   -1.00000,   0.00000
   S  0.00000,   -1.00000,    1.00000   N  0.00000,   -1.00000,   0.00000
     -0.30902,   -0.95106,    1.00000   N -0.30902,   -0.95106,   0.00000

POLYGON
     0.00000,    -1.00000,   -1.00000   N 0.00000,    -1.00000,   0.00000
     0.30902,    -0.95106,   -1.00000   N 0.30902,    -0.95106,   0.00000
   S 0.30902,    -0.95106,    1.00000   N 0.30902,    -0.95106,   0.00000
     0.00000,    -1.00000,    1.00000   N 0.00000,    -1.00000,   0.00000

POLYGON
     0.30902,    -0.95106,   -1.00000   N 0.30902,    -0.95106,   0.00000
     0.58779,    -0.80902,   -1.00000   N 0.58778,    -0.80902,   0.00000
   S 0.58779,    -0.80902,    1.00000   N 0.58778,    -0.80902,   0.00000
     0.30902,    -0.95106,    1.00000   N 0.30902,    -0.95106,   0.00000

POLYGON
     0.58779,    -0.80902,   -1.00000   N 0.58778,    -0.80902,   0.00000
     0.80902,    -0.58779,   -1.00000   N 0.80902,    -0.58779,   0.00000
   S 0.80902,    -0.58779,    1.00000   N 0.80902,    -0.58779,   0.00000
     0.58779,    -0.80902,    1.00000   N 0.58778,    -0.80902,   0.00000

POLYGON
     0.80902,    -0.58779,   -1.00000   N 0.80902,    -0.58779,   0.00000
     0.95106,    -0.30902,   -1.00000   N 0.95106,    -0.30902,   0.00000
   S 0.95106,    -0.30902,    1.00000   N 0.95106,    -0.30902,   0.00000
     0.80902,    -0.58779,    1.00000   N 0.80902,    -0.58779,   0.00000

POLYGON
     0.95106,    -0.30902,   -1.00000   N 0.95106,    -0.30902,   0.00000
     1.00000,     0.00000,   -1.00000   N 1.00000,     0.00000,   0.00000
     1.00000,     0.00000,    1.00000   N 1.00000,     0.00000,   0.00000
     0.95106,    -0.30902,    1.00000   N 0.95106,    -0.30902,   0.00000
;
```

ESTABLISHING A WORKSPACE IN MEMORY

The rendering process requires that a large contiguous block of mass memory be available. This area is known as working storage and once reserved it is not available for other uses. Before any rendering operations can be performed, you must establish a workspace in mass memory. The best time to reserve working storage is immediately after booting, when large requests can be filled more easily.

Each polygon of a solid object with four vertices will require approximately 150 bytes of reserve working storage. Memory needs will vary from figure to figure depending on the complexity of the object, the operations to be performed on the data structure, and the view.

Working storage must be explicitly reserved with the RESERVE_WORKING_STORAGE command.

The syntax for working storage command is as follows:

```
RESERVE_WORKING_STORAGE n;
```

where

the current working storage block is replaced with another containing at least *n* bytes. If *n* is less than or equal to 0, no new block is allocated.

Typically, you should reserve 200,000 to 400,000 bytes of working storage when you begin a session. The command to do this is:

```
RESERVE_WORKING_STORAGE  400,000;
```

After one working storage request is made, subsequent requests do not add to the original working storage; they replace the original working storage.

Working storage is not freed by the INITIALIZE command. The only way to free all working storage is to enter RESERVE_WORKING_STORAGE with a number less than or equal to 0. If a working storage request is followed by another, smaller request, an amount of memory equal to the difference between the two requests is freed.

A previously allocated working storage area is released prior to filling the request for a new working storage area. Thus, a request for a smaller working storage area can always be fulfilled. However, because the working storage must be a contiguous block of memory, even slight increases in the working storage size may not be satisfied upon arbitrary request.

If a contiguous block of memory cannot be allocated, no working storage is allocated and any previous storage is deallocated. If working storage is too small or has not been reserved, the rendering request is ignored and an error message is issued.

Additional Memory Requirements

In addition to the working storage space, extra mass memory is needed to create hidden-line renderings. This memory is referred to as ***transient*** memory and is automatically allocated and deallocated by the system. If adequate mass memory is not available for transient storage, the hidden-line process will terminate prematurely, and an error message will be generated. For this reason E&S recommends 2Mb or more of memory for renderings of objects with numerous polygons.

For hidden-line removal, each polygon (with four vertices) in the object will require approximately 150 bytes of transient storage.

MARKING AN OBJECT FOR RENDERING

An object must be defined to be a surface or a solid before rendering operations can be applied. The commands to do this are:

> SOLID_RENDERING command. This command creates an operation node in the data structure (a "solid-rendering node") which declares all of its descendant polygon data nodes to define a solid.
>
> SURFACE_RENDERING command. This command creates an operation node in the data structure (a "surface-rendering node") which declares all of its descendant polygon data nodes to define a surface.

These commands declare a POLYGON data object to be either a solid or a surface and mark it to perform renderings on it. The nodes they establish are called rendering operation nodes.

Rendering nodes should never be multiply instanced either directly or indirectly.

Only polygon nodes are used in renderings. Vector and character nodes occurring beneath a rendering node are ignored by the rendering operations. Transformation nodes are not retained in the rendering, but their effect is incorporated into the data nodes.

A sectioned rendering concatenates all transformations below the rendering node into the rendering, backface and hidden-line renderings also incorporate the current transformation matrix at the point of the rendering node. For this reason, a saved hidden-line or backface removal rendering should be placed beneath a

```
MATRIX_4X4  1,0,0,0  0,1,0,0  0,0,0,0  0,0,1,1;
```

command to be properly re-displayed. If this is not done, the rendering will have two sets of transformations applied to it when it is re-displayed (the transformations applied when the rendering was created and the transformations again applied when the rendering is re-displayed).

While conditional nodes (IF) are not incorporated into renderings, the rendering will account for the state of the conditional node when the rendering is created.

A POLYGON data node can be displayed by itself. However, if the POLYGON data node is to be rendered, it must have a rendering node as an ancestor. All rendering and display operations involving the object are done via the rendering node rather than the data node itself.

Syntaxes for the rendering commands are:

```
name := SOLID_RENDERING APPLIED TO name1;

name := SURFACE_RENDERING APPLIED TO name1;
```

where

- name1 names either (a) a POLYGON node, or (b) an ancestor of one or more POLYGON nodes.
- If (b) is the case, any rendering referring to name is performed on all of the POLYGON objects descended from name1 at once.
- SOLID and SURFACE are acceptable abbreviations for these commands.

Non-Polygon Data Nodes Marked for Rendering

If non-POLYGON data nodes (such as VECTOR_LIST, CHARACTERS, LABELS, POLYNOMIALS, and B-SPLINES) are included in name1, these data objects are displayed along with the POLYGON objects prior to rendering but are omitted from renderings. Rendering operations have no effect on these data nodes.

Admissible Descendants for Rendering Operate Nodes

IF and SET CONDITIONAL BIT, IF and SET LEVEL_OF_DETAIL, INCREMENT LEVEL_OF_DETAIL, DECREMENT LEVEL_OF_DETAIL, IF PHASE, SET RATE, SET RATE EXTERNAL, SET DEPTH_CLIPPING, and BEGIN_STRUCTURE... END_STRUCTURE may be placed between a rendering node and its data. A rendering takes into account any effects of these nodes at the time the request is made. For example, if IF PHASE and SET RATE are being used to blink an object and that object is "off" at the moment the request is made, the object is excluded from the rendering.

The nodes mentioned above can also be placed above the rendering node with the same result.

The transformations ROTATE, TRANSLATE, SCALE, MATRIX_2X2, MATRIX_3X3, MATRIX_4X3 and LOOK may be placed between a rendering node and its data node(s). However, these nodes should be used with caution, since, like the operate nodes mentioned above, their effects will be incorporated into renderings, and precision problems may result.

Another potential problem with interposing these transformations between a rendering node and the data arises when renderings are being saved.

Since most vertices in an object usually belong to more than one polygon, each vertex should be defined with the same numerical value in each of its polygons; otherwise, precision discrepancies may cause inaccurate renderings.

In general, the five nodes WINDOW, VIEWPORT, EYE, FIELD_OF_VIEW, and MATRIX_4X4 should NOT be made descendants of a rendering node. Like other transformations, these five are incorporated into the output data from a rendering operation. However, these rendered data are generally displayed within a framework that already includes global 4x4-matrix transformations of its own. Including these transformations as part of the rendering, then, usually has the net effect of applying an unwanted double-WINDOW (double-VIEWPORT, etc.) to the rendered object.

SOLID_RENDERING and SURFACE_RENDERING may not be descendants of a rendering node, especially if multiply-instanced rendering nodes are involved. If this rule is not observed, bad renderings or a system crash may result. ***The system does not check for this condition.***

Other nodes, including character transformations and the SET nodes (SET RATE, SET COLOR, SET PLOTTER) are not carried over by rendering operations into a rendering when placed beneath a rendering node. Such nodes must be placed above a rendering node to produce their customary effects on renderings.

Rendering Nodes Must Be Displayed Before Rendering

Before you can render an object, its rendering node must be part of a structure which is displayed (using the DISPLAY command). If the object itself is displayed but its rendering node is not, no renderings can be created.

For example, if the command sequence

```
A := SOLID_RENDERING APPLIED TO B;
B := POLYGON ... ... ... ... ;
```

has been entered, the DISPLAY command should be DISPLAY A; and not DISPLAY B.

CREATING RENDERINGS

An appropriate integer sent to a SOLID_RENDERING or SURFACE_RENDERING node produces a rendering of that node's descendant polygon object. When a rendering is first created for an object, a second set of data is created and "grafted" just below the rendering node for the original object. To display the rendering, the Graphics Control Processor traverses the path to this new data. This happens automatically when the rendering is requested. The original data existent before the rendering was applied remain intact and are accessible via input to the rendering node.

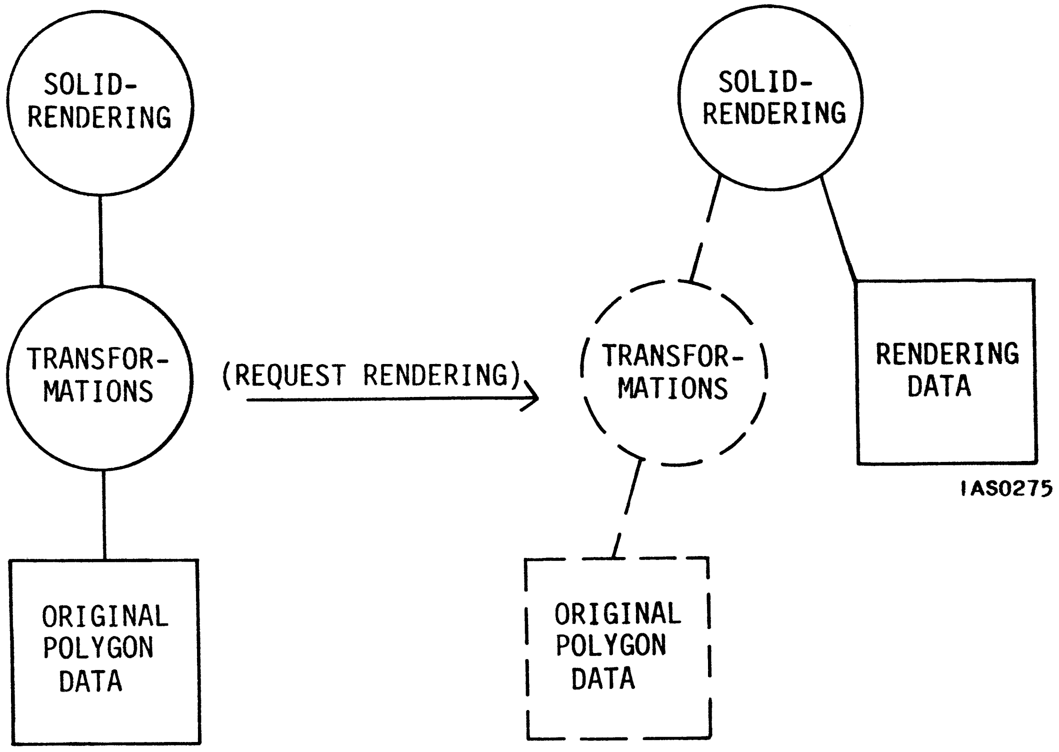


Figure 30. Path to Rendering Data

When the original object is re-displayed, the path to the original object is traversed, however, the rendering data remains intact.

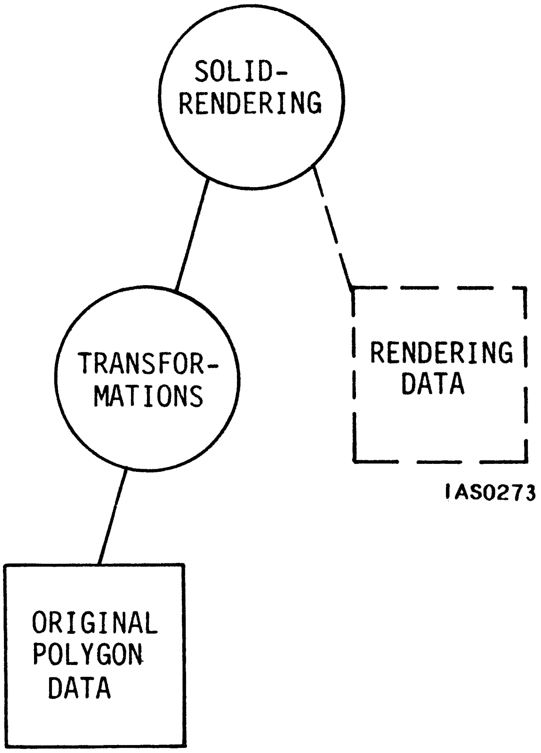


Figure 31. Path to Original Data

At this point, the rendering can easily be displayed again, since its data still exists.

When a second rendering is done on this object, it replaces the first rendering.

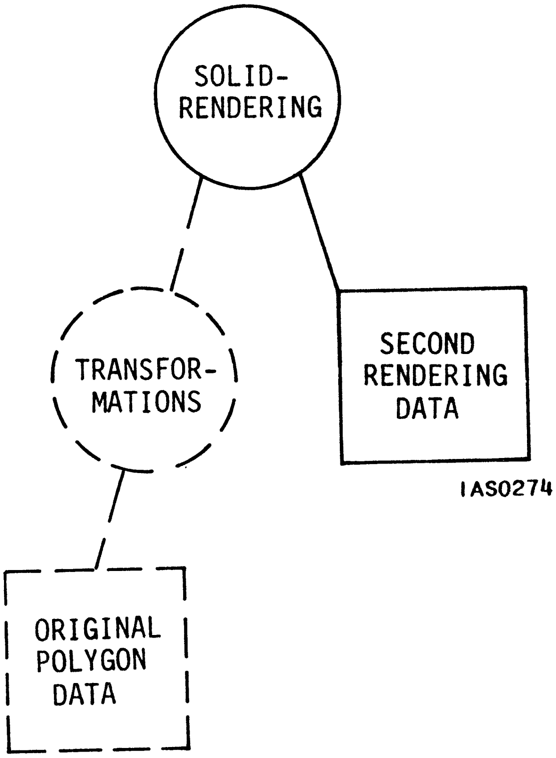


Figure 32. Path to Second Rendering

The rendering whose data occupy this place in the structure at a particular time is called the "current rendering." Thus, the current rendering is always the one most recently created, even if it is not currently displayed. Each rendering node has its own current rendering.

After requesting a rendering operation you cannot communicate with the host or do any other PS 340 processing until the rendering is completed.

Rendering Node Connections

Rendering nodes have two inputs. Input <1> accepts an integer, a Boolean, or a string designating the rendering operation to be performed. Tokens sent to input <1> of the rendering node cause a rendering to be created, saves a rendering under a particular name, or toggles the display from the rendering to the original object.

Input <2> accepts a Boolean to change the object definition from a surface to a solid or vice versa. After defining an object to be a surface or a solid with the SOLID_RENDERING or SURFACE_RENDERING commands, you can change the definition by sending a Boolean to input <2> of the rendering operate node. This input allows you to have one rendering node (created with either command) and alternate between a surface and a solid definition. A true sent to the input <2> declares the object to be a solid; a false declares the object to be a surface. Solids are always rendered correctly (although not as efficiently) as surfaces. Surfaces are handled by the system as solids (they will not cause the system to fail); however, they may not be rendered correctly.

Rendering nodes also have an output which outputs a true if the rendering is displayed and a false if it is not displayed. You can connect this output via the CONNECT command to trigger some other action that was waiting on completion of the rendering process.

For example, the commands

```
A := SOLID RENDERING THEN B;
CONNECT A<1>:<1>C;
```

cause the output of a rendering node to be sent to input <1> of C.

Any input to input<1> of a rendering node causes an output. Inputs sent to input<2> will not cause an output to be sent. If output<1> has not been connected, and an integer, string, or Boolean is sent to input<1>, a message will appear on the screen upon successful completion of the rendering operation. An error message will appear if the rendering was not completed.

The connections for the SOLID_RENDERING and SURFACE_RENDERING operate nodes are:

```
               ______________________________________________
Integer,----->| <1>                                      <1> |------->Boolean
String, or    |                                              | {True if displayed}
Boolean       |                                              | {False if not displayed}
              |         Solid_Rendering Applied To name1;    |
Boolean-----=>| <2>  Surface_Rendering Applied To name1;     |
              |______________________________________________|
```

Acceptable Values for Input <1>

0: Toggles between the current rendering and the original object.
1: Creates and displays a cross-section of an object defined by the sectioning plane (solid only).
2: Creates and displays a sectioned rendering.
3: Creates and displays a rendering using backface removal (solid only).
4: Creates and displays a rendering using hidden-line removal.
5: Generates a wash-shaded image on the raster display.
6: Generates a flat-shaded image on the raster display.
7: Generates a smooth-shaded image on the raster display.

String: Causes the current rendering to be saved under the name given in the string.
False: Sets the original view. The original descendant structure of the rendering operate node is displayed.
True: Sets the rendered view. The rendered view of the original descendent structure of the rendering operate node.

Acceptable Values for Input <2>

True: Declares the object to be a solid.
False: Declares the object to be a surface.

These operations are discussed in the following sections.

Backface Removal

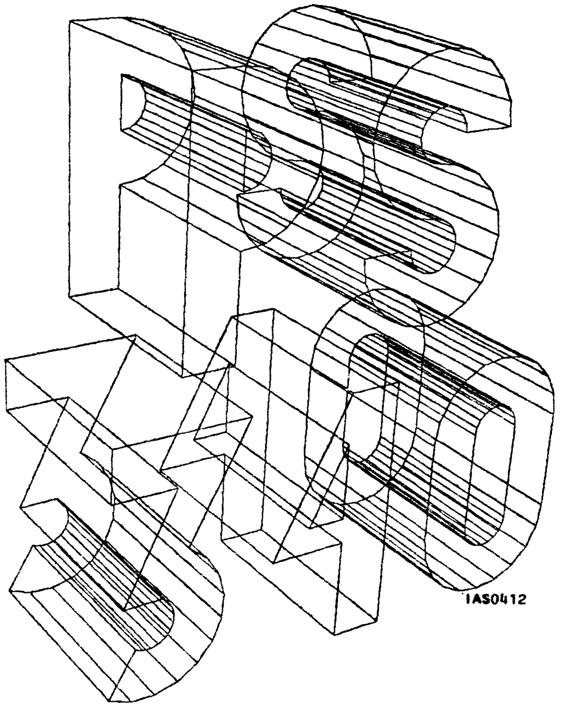
Backface removal is an intermediate step in hidden-line removal, during which all polygons facing away from the viewer are removed. Since backface removal takes considerably less time than hidden-line removal, this operation is provided separately to allow you to see what a hidden-line rendering will look like.

This operation is especially useful in obtaining quick previews of hidden-line renderings of complex solids, when an appropriate viewing angle is being decided upon by trial and error. The backface-removed rendering is an unfinished hidden-line rendering. It is not identical to the end-product in every line segment, but close enough to give a rough idea of the outcome.

Only solids can be subjected to backface removal; the operation has no visual effect on surfaces.

Figure 33 is an example of a solid before and after backface removal.

(Before Backface Removal)



(After Backface Removal)

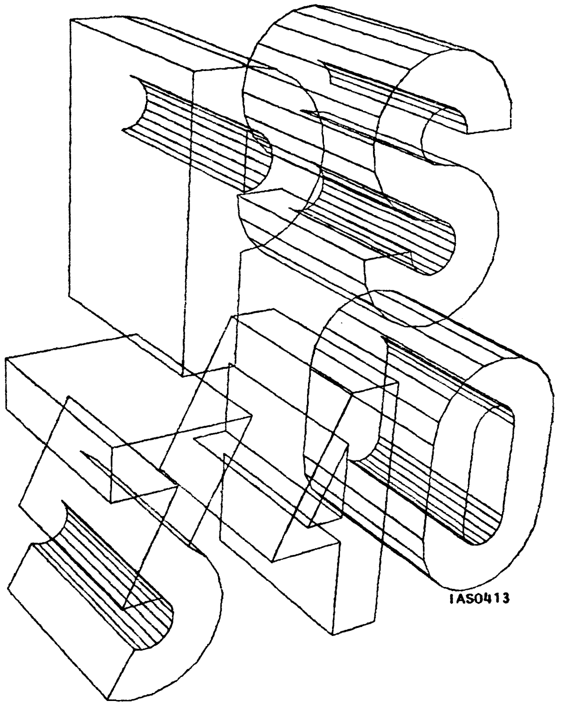


Figure 33. Solids Before and After Backface Removal

Sending an integer of 3 to input <1> of the rendering node creates a backface-removed rendering in the working storage area.

Exercise

Load the PS 340 Cat demonstration diskette on the PS 340 and apply the backface removal operation by selecting from the menu on the right side of the screen. Refer to the ***PS 340 Installation Manual*** for instructions on loading and operating the CAT program.

Hidden-Line Removal

Hidden-line removal generates a view in which only the unobstructed portions of an object are displayed. All parts of the edges of polygons that would be obscured by other polygons are removed.

Three steps are involved in the hidden-line removal process.

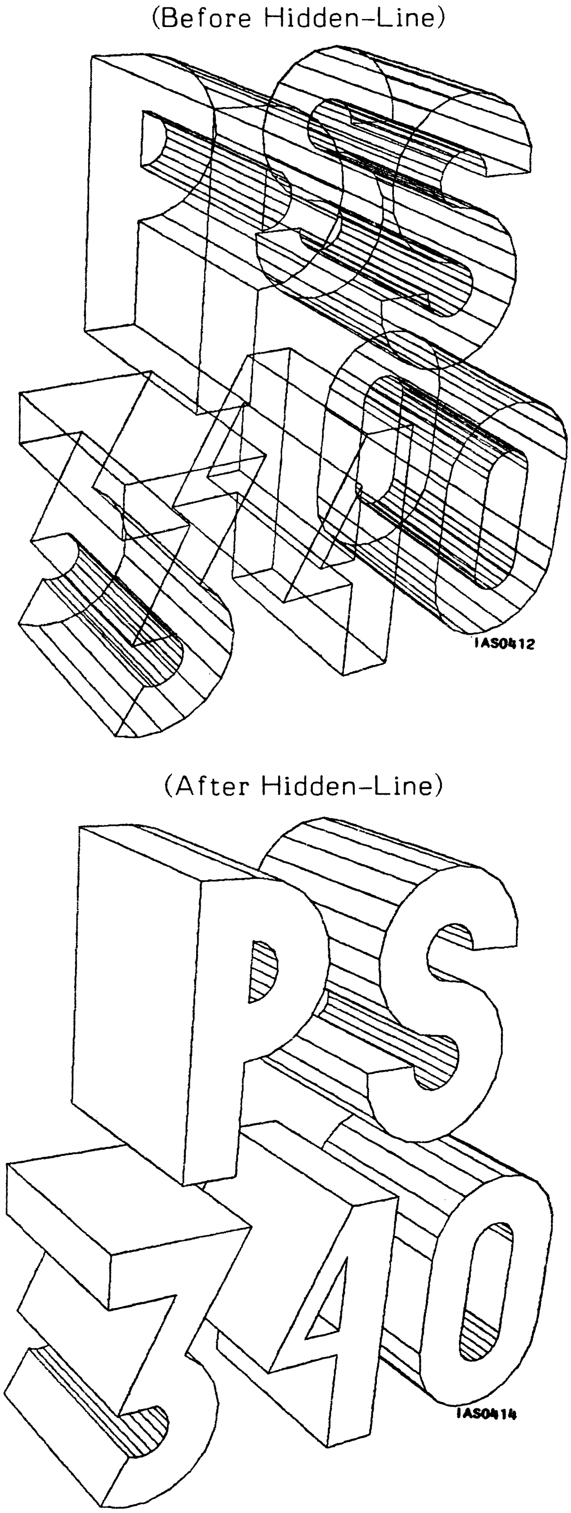
1. Back faces are removed or made front facing. This happens very quickly (1-3 seconds), during which time the screen will appear blank.

2. The remaining polygons' edges are sorted by their Y-coordinates. This step takes approximately 30 seconds for 3,000 polygons, during which time the backface picture is created. The time required for sorting depends on the number of polygons and the order in which they are defined.

3. Edges are tested against polygons and clipped by those that obscure them. During this time, the backface picture is removed and the final hidden-line picture appears from top to bottom of the display.

   This step may take 5 minutes or more for approximately 3,000 polygons, depending on the number of polygons and the view. In general, it takes more time to process polygons along the X and Z axis than those along the Y axis.

Hidden-line removal may be performed on both solids and surfaces. Hidden-line views cannot be subjected to further rendering operations.


(Before Hidden-Line)

(After Hidden-Line)

Figure 34. Solids Before and After Hidden-Line Removal

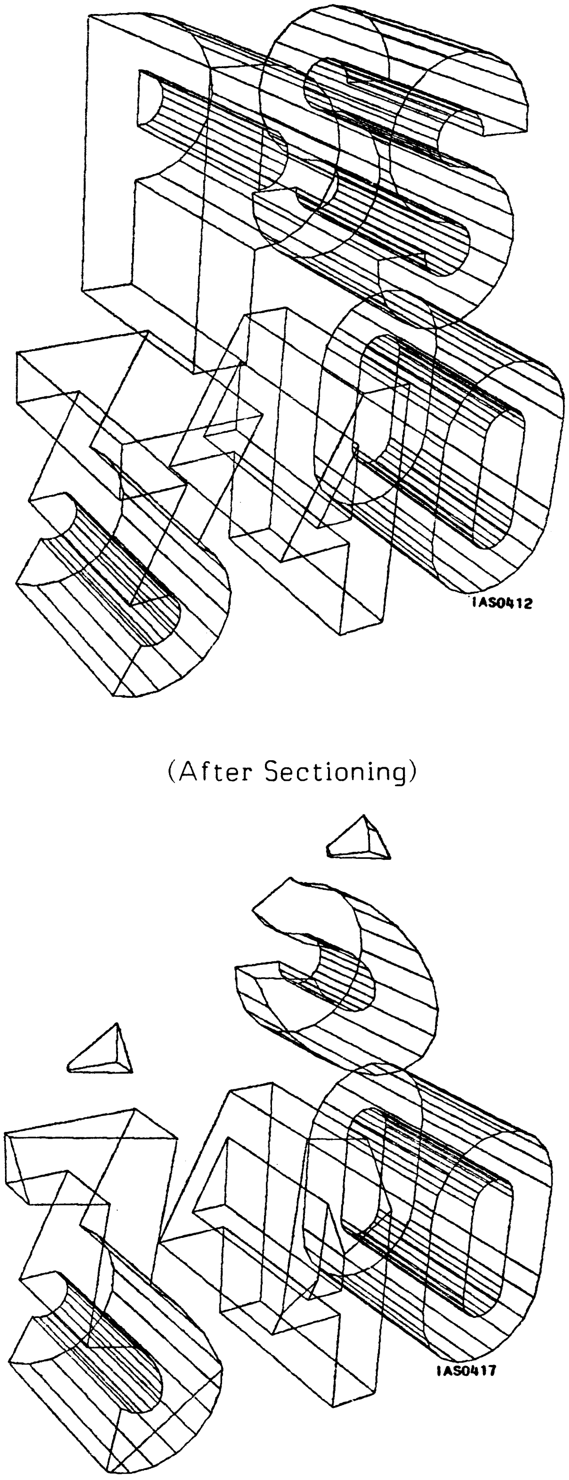
Sending an integer of 4 to input <1> of the rendering node creates a hidden line rendering in the working storage area.

Exercise

Use the PS 340 CAT demonstration program to see the effects of the hidden-line operation on the various objects.

Sectioning

Sectioning yields a "cutaway view" of an object. This operation makes use of a sectioning plane passing through the object and dividing the object into two pieces. The half of the object that is behind the plane is discarded and only the front section of the object is displayed.



(Before Sectioning)

(After Sectioning)

Figure 35. Solid Objects Before and After Sectioning

Both solids and surfaces can be sectioned. For solids, a capping polygon (or a set of coplanar capping polygons) is generated where the sectioning plane intersects the solid. Such capping polygon(s) "close off" the sectioned object so that it, too, becomes a solid.

Sectioning does not generate capping polygons for surfaces.

A sectioned object may be saved and then subjected to further surface-rendering operations such as, re-sectioning, hidden-line removal, or backface removal.

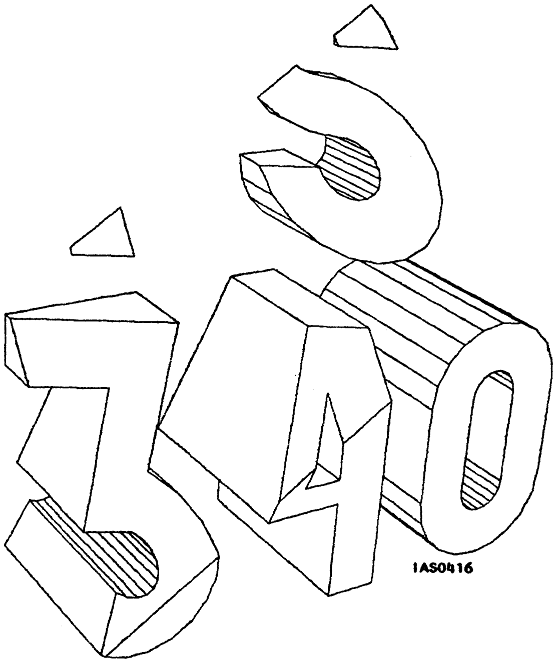


Figure 36. Hidden-Line Rendering of Sectioned Object

Although there is generally no immediate visual evidence that a capping polygon has been produced, capping polygons become a part of the definition of the sectioned solid, and further rendering can disclose their existence. For example, suppose that a solid and a surface are each sectioned vertically, yielding the two sectioned objects below. Assume that each object intersects with its sectioning plane at its two right-most faces. It is impossible to tell which object is capped.

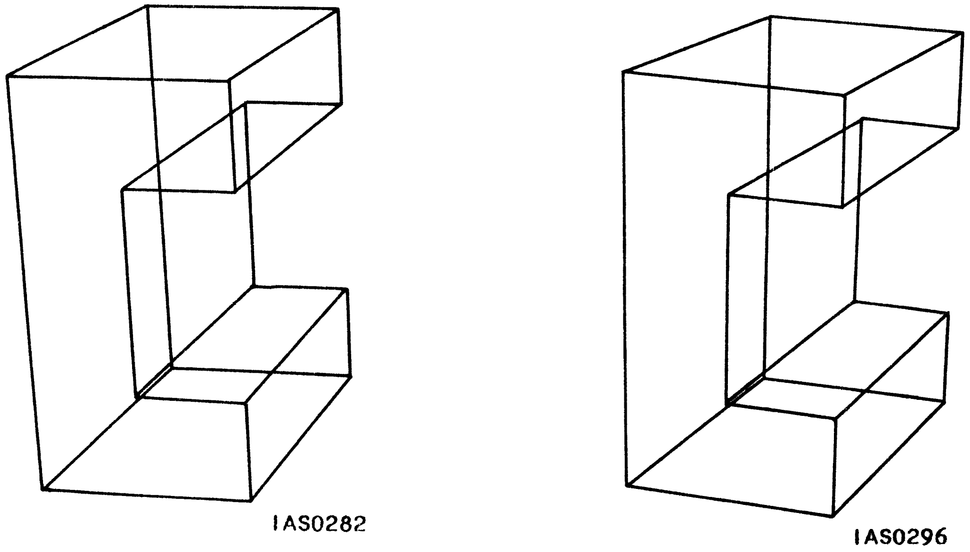


Figure 37. Sectioned Object With Capping Polygons

However, hidden-line removal shows that the object on the left is a solid, while the object on the right is open at its right-most faces.

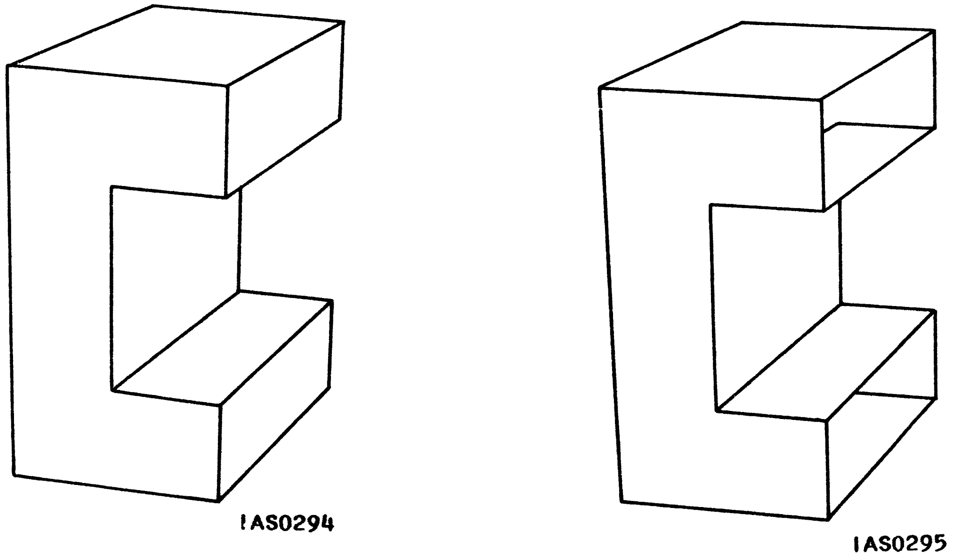


Figure 38. Sectioned Object With After Hidden-Line Removal

Sectioning proceeds very rapidly (1-3 seconds); the display may blink briefly while it is being performed.

Sectioned objects are also sliced along the planes of the viewing frustum. The sectioning plane must be encountered by the display processor prior to the rendering node. If a sectioning plane has not been found, the screen will blank for 15 seconds and an appropriate error message will be generated.

Sending an integer of 2 to input <1> of the rendering node makes use of the established sectioning plane to create a sectioned rendering in the working storage area.

Using the PS 340 CAT demonstration program, apply the sectioning operation to the objects available on the menu.

Establishing a Sectioning Plane

Defining, displaying, and positioning a sectioning plane are the first steps in producing a sectioned rendering of an object. Hidden-line removal and backface removal do not require sectioning planes, but they can be applied when a sectioned rendering is saved and subjected to further renderings.

The SECTIONING_PLANE command creates a sectioning-plane node which indicates that a descendant POLYGON is a sectioning plane. The syntax is:

```
name := SECTIONING_PLANE APPLIED TO name1;
```

where

- name1 names either (a) a POLYGON command or (b) an ancestor of a POLYGON command.
- SECTIONING_PLANE may be abbreviated SECT.

The Sectioning Plane's Data Definition

The data which actually define a sectioning plane are contained in a POLYGON node; SECTIONING_PLANE simply indicates that a given POLYGON represents a sectioning plane.

The sectioning plane is the plane in which a specified POLYGON lies. The polygon itself need not intersect the object to be sectioned, as long as some part of the plane does.

The sectioning plane is the plane containing the polygon defined by the first POLYGON clause of the first polygon node encountered by the Display Processor as it traverses the branch beneath a sectioning-plane node.

If the polygon node has more than one POLYGON, only the first polygon determines the sectioning plane. The other polygons have no effect on sectioning operations, but are displayed along with the defining polygon. This can be put to use in designing an indicator which shows the side of the plane at which sectioning will remove (or preserve) polygon data (below). For example, the command

```
SPDATA :=
POLYGON    -.9,-.9,0      -.9,.9,0       .9,.9,0        .9,-.9,0

POLYGON    .1,0,0         .1,0,-.3       .15,0,-.3      0,0,-.45
           -.15,0,-.3     -.1,0,-.3      -.1,0,0

POLYGON    0,.1,0         0,.1,-.3       0,.15,-.3      0,0,-.45
           0,-.15,-.3     0,-.1,-.3      0,-.1,         0 ;
```

defines a sectioning plane with two polygonal arrow-indicators as shown in Figure 39.

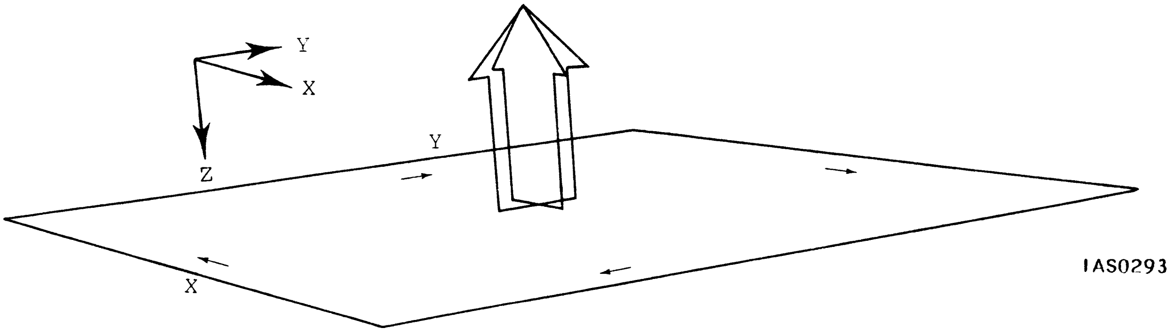


Figure 39. Sectioning Plane Definition

Sectioning preserves those parts of an object lying in front of the plane, and removes those parts lying in back of the plane. (The front side of a sectioning plane is the side on which you see the vertices of the plane's defining polygon running clockwise, where the vertices are considered in the order of their appearance in the POLYGON clause.)

No SOLID_RENDERING or SURFACE_RENDERING operation node, whether below or above the sectioning-plane node, may be an ancestor of a sectioning plane's defining POLYGON. The PS 340 interprets such POLYGONs as objects to be rendered rather than as sectioning-plane definitions, and issues a "Sectioning plane not found" message when a sectioning attempt is made.

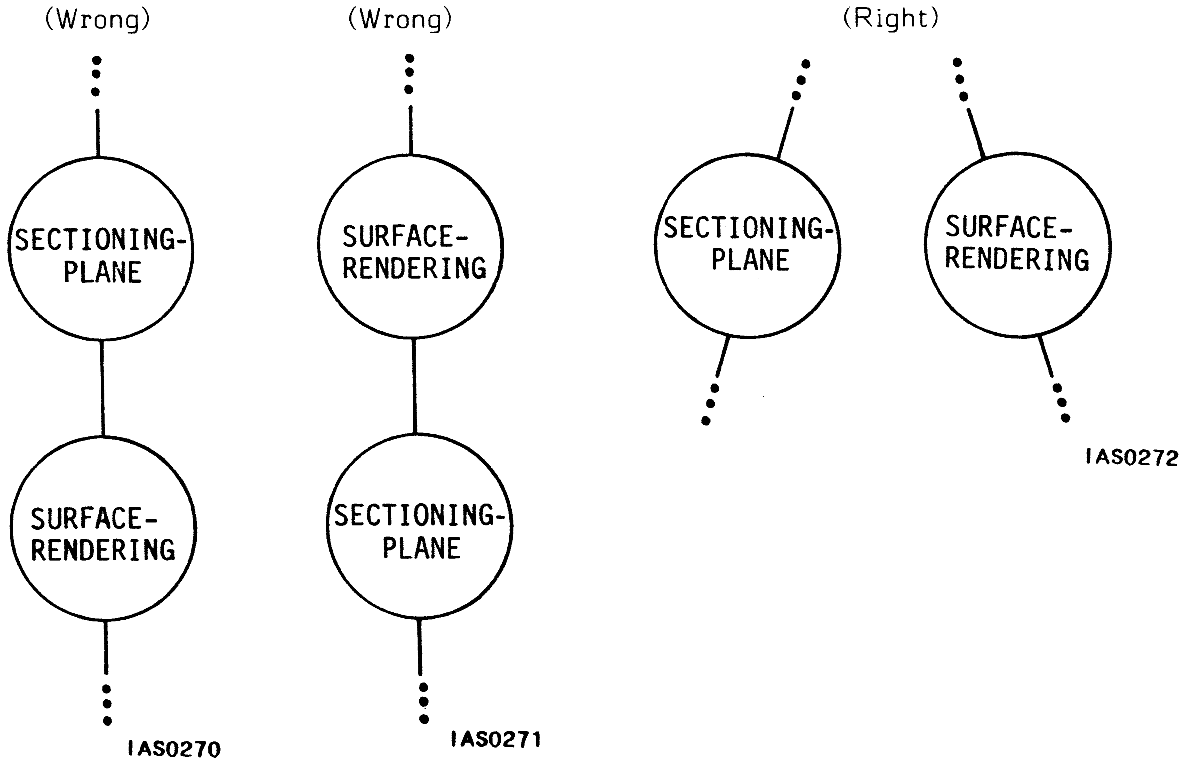


Figure 40. Data Structure of Sectioning Plane

Other nodes nodes which do not represent matrix viewing transformations, such as SET RATE and SET PLOTTER, may be placed either above or below the sectioning-plane node as needed.

Typically, you will want to orient the plane interactively, by connecting an interactive device via function networks.

Sectioning-Plane Node Must Be Displayed Before Rendering

Before an object can be sectioned, the sectioning-plane node must be part of a structure which is DISPLAYed. If the plane's defining POLYGON is itself DISPLAYed but its sectioning-plane node is not, no renderings can be created.

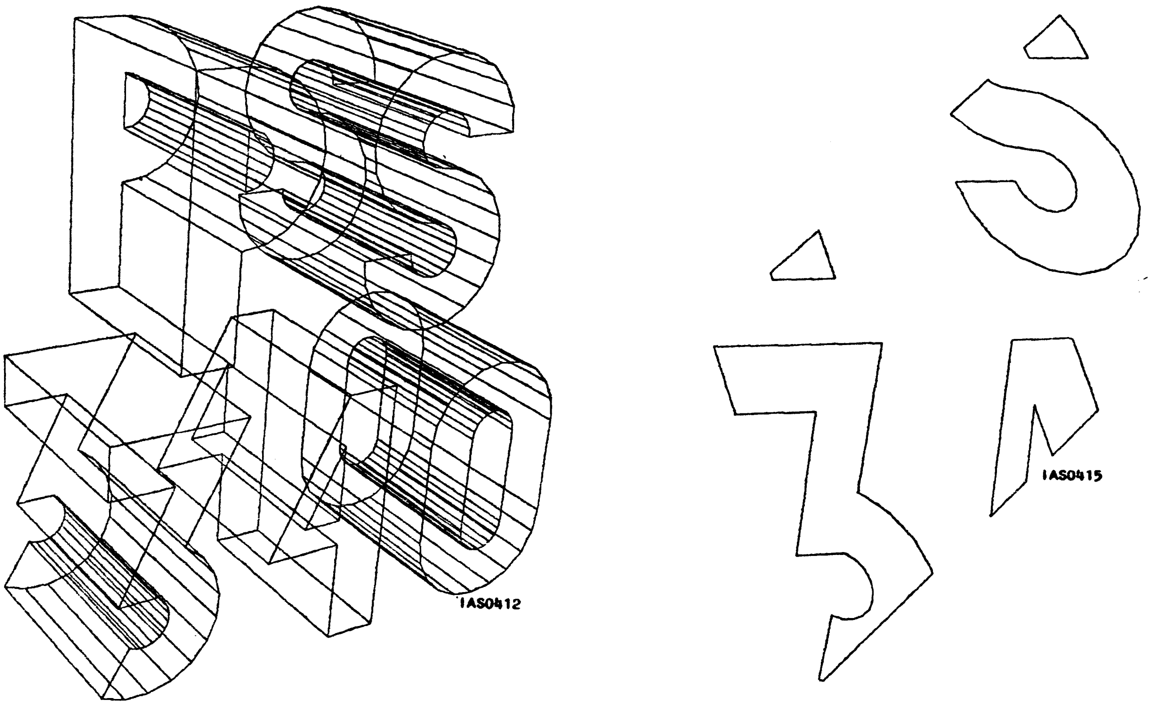
For example, if the command sequence

```
A := SECTIONING_PLANE APPLIED TO B;
B := POLYGON ... ;
```

has been entered, the DISPLAY command should be DISPLAY A; and not DISPLAY B.

Cross Sectioning

The cross sectioning operation makes use of a defined sectioning plane to create a cross section of an object. When this operation is used, both sides of the object are thrown away and only the slice of the object defined by the sectioning plane remains. Essentially, the object is sectioned and only the capping polygons remain.



Original Object
(Before Cross sectioning)

Rendered Object
(After Cross Sectioning)

Figure 41. Solids Before and After Cross Sectioning

Cross sections can only be created for solid rendering nodes. This operation proceeds very rapidly (1-3 seconds), in which time the display blanks momentarily while the object is being sectioned. The cross-section is also clipped by the planes of the viewing frustum.

Sending an integer of 1 to Input <1> of the rendering node creates a cross section in the working storage area.

Exercise

Use the PS 340 CAT demonstration program to experiment with the Cross Sectioning operation, or send the integer 1 to the rendering node of a polygon object you have created.

Toggle Between the Rendering Object and the Original Object

It is often useful to compare objects before and after rendering operations have been applied. The TOGGLE operation allows you to do this. Sending a 0 to input <1> of the rendering node toggles the display between the rendering and the original object. Both the rendering and the original object are left intact and can be redisplayed until overwritten or saved.

Setting the View

Sending a false to input <1> of the rendering operation node causes the original descendent structure of the SOLID_RENDERING or SURFACE_RENDERING node to be displayed (sets the view to the original structure). The rendered view is not affected, other than being removed from the display. The rendered view can be restored and displayed again by sending true or fix(0) to the rendering operation node.

Sending a true causes the rendered view (if any) to be displayed instead of the original descendent structure of the rendering operation node (sets the view to the rendered view). The original view remains intact, apart from being removed from display.

Changing the Definition of the Object

Sending a Boolean to input <2> of the rendering node controls whether the descendant polygons are to be treated as a solid or a surface, enabling a solid rendering node to be converted to a surface rendering node and vice versa. True sent to input <2> defines the node as a SOLID-RENDERING node whatever the original state was. False defines the node as a SURFACE_RENDERING node. The default is determined by the word SOLID or SURFACE in the original command that created the node.

SAVING AND COMPOUNDING RENDERINGS

To save a rendering is to give it a name by which it can be referenced.

Requesting and displaying a rendering creates rendering data, but does not create a "node" in the normal sense. It cannot be referenced nor subjected to further rendering operations until it is "saved" by naming it. Saving the rendering, which establishes a rendering as a separate named data node, is therefore a prerequisite to ***compounded renderings,*** or further renderings of the rendered object.

After a rendering is saved, it is no longer considered a "current" rendering. Therefore, the toggle operation (Booleans and a fix(0) sent to the rendering node) no longer affect the rendering.

How to Save a Rendering

To save a rendering, send a string message to input <1> of the SOLID_RENDERING or SURFACE_RENDERING operation node. All illegal PS 300 names are rejected and an error message is generated.

The string should specify the name of the node which is to contain the saved-rendering data. If the named node does not exist, it is created; if it does exist, the saved-rendering data replace the original contents of the node.

All polygons in the rendering are taken into account in the saved rendering. It is not possible to exclude selected polygons or polygon data nodes from saved renderings.

Contents of a Saved Rendering

Backface removal and sectioned renderings are saved as polygon lists; hidden-line renderings are saved as vector lists.

When a sectioned rendering is saved, all transformations between the rendering operation node and the polygon data node are applied to the polygon data. The result is stored in the new data node.

When a backface or hidden-line rendering is saved, ***all ancestor transformations of the polygon data node*** are applied to the polygon data before the result is stored in the new node. This occurs even if those transformations are above the rendering operation node.

Common Uses of Saved Renderings

The most common reason for saving a rendering is to create a compound rendering from it.

Common types of compound renderings are: (a) re-sectioning of a sectioned rendering and (b) hidden-line removal applied to a sectioned rendering. Backface renderings, which are useful mainly for previewing time-consuming hidden-line operations on complex objects, are not generally rendered further. Hidden-line renderings cannot be rendered further because they are vector lists, and only polygons can be rendered. (The example at the end of this module gives a program which can be used to create compounded renderings.)

Saved renderings are also useful when multiple hidden-line renderings of the same object, seen from different viewpoints, are to be displayed in separate viewports. A sectioned rendering can be viewed from multiple viewpoints without saving, but a hidden-line rendering is a vector list which loses its hidden-line character when the viewpoint shifts. Therefore, a separate hidden-line rendering must be saved for each view to be displayed.

Displaying a Saved Rendering

When displaying a saved rendering, the rendering already incorporates some or all of the transformations which existed in the data structure at the time the rendering was requested.

Displaying Saved Sectioned Renderings

Since sectioned renderings already incorporate the transformations which existed between the rendering operation node and the original polygon data node, the appropriate place to attach a saved sectioned rendering is either:

- at the same level as the rendering operate node, OR
- just below the rendering operate node (without intervening transformations).

Attaching the saved rendering further down than this (for example, at the same level as the original polygon data node) causes a misleading display. Any transformations lying between the rendering operation node and the saved rendering will actually be applied twice. This will be applied once explicitly in the data structure, and once implicitly in the saved-rendering data.

Attaching the saved rendering above the rendering operation node may also cause a misleading display. This excludes some of the viewing (or other) transformations globally applied to the original data.

It is not necessary to attach a saved rendering anywhere in the existing structure. The rendering can be saved in a node apart from this hierarchy. Any desired new transformations can then be applied to it. The program example at the end of this module illustrates this guideline.

Displaying Saved Backface and Hidden-Line Renderings

Backface and hidden-line renderings incorporate all of the transformations which are applied to the original data node. Saved backface and hidden-line renderings should be attached beneath the following matrix for proper display:

```
MATRIX_4X4   1,0,0,0
             0,1,0,0
             0,0,0,0
             0,0,1,1  ;
```

No other transformations should be applied to the saved rendering. To include other transformations is to raise the double-transformation problem discussed above for saved sectioned renderings. The saved rendering and its matrix should be either (1) attached at the very top of the existing display data structure (as shown in the programming example at the end of this module) or (2) separated from that structure altogether.

The purpose of the special MATRIX_4X4, is to display the object without Z-values and perspective.

Exercise

Use the PS 340 CAT demonstration program or define an object of your own, apply a rendering operation, and save the rendering.

DISPLAYING SHADED IMAGES

The PS 340 optional raster system consists of a printed circuit card that outputs static images to a pixel raster display. The raster system can be used as an "image buffer" to display host-generated images or it can display "shaded images" derived locally from PS 340 polygonal models.

When using the raster display as an image or frame buffer, the PS 340 is only used as a communications link between the host and the raster system. No standard PS 340 commands or data structures are used to display host generated images.

This module deals only with displaying shaded images derived locally from PS 340 polygonal models. Run-length encoding, the process of displaying host generated images, is documented in *The PS 340 Raster Programmer Guide.*

Requesting a shaded image computed locally on the PS 340 and displaying it on the raster monitor requires that an integer be sent to the rendering node input of the data structure. When a shaded image is requested, the hidden-line view of the object is displayed concurrently on the calligraphic display.

Because the refresh processor is used to generate the raster image, the calligraphic hidden-line image may flicker or disappear entirely while shaded renderings are created.

There are three types of shaded renderings: ***wash shading, flat shading*** and ***smooth shading.***

Wash shading (area fill) generates a shaded image of the raster image buffer concurrent to the generation of the hidden-line picture. In wash shading, the color of each polygon is determined from the color given in the attribute node corresponding to the polygon. All normals, light sources, other lighting parameters, and depth cueing parameters are ignored. Sending the integer 5 to input <1> of the rendering node creates a wash-shaded object and displays it on the raster screen.

Flat shading generates a flat shaded image on the raster image buffer concurrent to the generation of the hidden-line picture. The process considers color, one light source, and the depth cueing parameter and shades the polygons accordingly.

A polygon's color is affected by its orientation as well as the color and direction of the light source. If specified in the polygon data definition, vertex normals and the diffuse and specular highlight specifications are ignored. Sending the integer 6 to input <1> of the rendering operation node displays the object with flat shading.

Smooth shading generates a smooth-shaded image on the image buffer while the hidden-line rendering is being created and displayed on the calligraphic monitor. Smooth shading varies the color of the polygon across its surface combined with the normals at the polygon's vertices, the color and direction of various active light sources, the polygons' attributes, and the depth cueing parameters. Sending the integer 7 to input <1> of the rendering operation node displays a smooth shaded object.

Specifying Attributes

In the section "Defining Polygonal Objects," you were introduced to the WITH ATTRIBUTES option. Attributes are applied to a collection of polygons by specifying the name of the attribute node after WITH ATTRIBUTES in the POLYGON command. If the WITH ATTRIBUTES option is not used in the POLYGON clause, the default attributes 0,0,1 for color, 0.75 for diffuse, and 0 for specular are assumed.

Using the ATTRIBUTES Command

The ATTRIBUTES command specifies the various characteristics of polygons used in the creation of shaded renderings. Attribute nodes are created with the ATTRIBUTES command and exist in mass memory (not as part of the data structure). The ATTRIBUTES command creates a named attribute node in mass memory that defines specific attributes to be applied to data when this node is referenced by the object's data structure.

When the display processor traverses the data structure with a polygon node containing a WITH ATTRIBUTES name1, the attributes in name1 are assigned to all polygons in the node until superseded with another WITH ATTRIBUTES clause. The various attributes may be changed from a function network via inputs to an attribute node or by reassigning the name, but the changes have no affect until a new rendering is created. No type checking is done by the shading process to ensure that WITH ATTRIBUTES indeed refers to an attribute node and not some other entity. If it does refer to some other entity, the display processor will interpret any values in that node as attributes, and display the object incorrectly.

Given:

```
<attr> :=   [ Color h [ ,s [ ,i ] ] ]
            [ Diffuse d]
            [ Specular s ]
```

The ATTRIBUTES command is:

```
Name := ATTRIBUTES <attr> [ AND <attr> ] ;
```

Meaning:

Color

The color attribute sets the basic color for the surface of a polygon. This attribute pertains only to shaded renderings on the raster display--it has no effect on the color of a polygon's edges on the calligraphic display. (These are changed using the WITH OUTLINE clause in the POLYGON command.) Color is given as hue (h), saturation (s), and intensity (i) and will change according to such things as shading style, light sources, orientation, depth cueing, ambient lighting, and highlights.

Hue specifies degrees around the color circle with 0 being pure blue, 120 pure red, and 240 pure green. Saturation varies from 0 for no saturation (grays) to 1 for full saturation. Intensity varies from 0 for no intensity (black) to 1 for full intensity.

If no color is specified, the default is white (s=0, i=1). If not specified, saturation and intensity default to 1. If only hue and saturation are specified, intensity defaults to 1. Values greater than 1 or less than 0 for saturation or intensity will become 1 or 0. Hue and saturation correspond to hue and saturation in the SET COLOR command but have greater precision. Remember that the color applies only to the shaded image; the color of the vector image displayed on the CSM color screen is set using the WITH OUTLINE clause of the POLYGON command.

Diffuse

Diffuse specifies the proportion of color contributed by diffuse reflection versus that contributed by specular reflection. Increasing *d* reduces the intensity of specular highlights, making the surface more matte; decreasing *d* makes the surface more shiney with a value of 1 eliminating specular highlights entirely. Values larger than 1 or less than 0 will be changed to 1 or 0. If no diffuse attribute is given, it defaults to 0.75. The diffuse attribute only affects smooth-shaded renderings.

Specular

The specular attribute adjusts the concentration of specular highlights, with increasing values of *s* increasing their concentration. Specular is a property of the object so the size of the highlight spot is not influenced by the light source, only by the *s* value. The more metallic the object is, the more concentrated the specular highlights. In the real world, objects are never completely specular (or diffuse) so you will get artificial effects if you have these values at a maximum.

Acceptable values of *s* are integers between 0 and 10, with values outside that rounded to 0 or 10 and a default of 4. As with diffuse, the specular attribute only affects smooth-shaded renderings.

And

A second set of attributes may be given after the word AND in the ATTRIBUTES command which apply to the obverse side of the polygon(s) concerned; in other words, the two sides of an object may have different attributes. The polygons considered on the obverse (backfacing) side by the system are those seen in a counterclockwise order for the view in which the rendering is carried out. The second set of attributes will only be applied in surface renderings (not solid).

The attributes defined for the first <attr> specify attributes for front-facing polygons. The <attr> after the AND specify the attributes of backfacing polygons.

You are not required to include the AND <attri> to specify different attributes for backfacing polygons. The command syntax for specifying just one set of polygons is:

```
Name := ATTRIBUTES <attr> ;
```

If the WITH ATTRIBUTES clause in a structure refers to an attribute node with two sets of attributes and no backfacing polygons exist for that object, the second set is ignored.

In the following example, an attribute node is created that defines the object to be blue. Since only the hue is specified for the color parameter, the default values for saturation and intensity (s=1, i=1) are assumed. The defaults for diffuse and specular (d=.75, s=0) are also assumed.

```
Blue := ATTRIBUTES COLOR 120;
Object :=  WITH ATTRIBUTES Blue
           POLYGON
              .
              .
              .
           POLYGON ;
```

All the polygons in the object are blue since the attribute clause assigns the attributes defined by Blue for all polygons until superseded by another WITH ATTRIBUTES clause.

In the following example, the <attri> before AND specify attributes for front-facing polygons in the object and the <attri> after AND specify the attributes for all backfacing polygons.

```
Red_Green:=  ATTRIBUTES COLOR 120,.5,.75  DIFFUSE .25 SPECULAR 1
             AND COLOR 240,1,.25;
  Object :=  WITH ATTRIBUTES Red_Green
             POLYGON
                .
                .
                .
             POLYGON ;
```

All front-facing polygons are colored red with .5 saturation and .75 intensity. The value for diffuse is .25 and the value for specular is 1. All backfacing polygons are green with 0 saturation and .25 intensity. Since no values for specular or diffuse are given in the second set of attributes, the defaults are assumed.

The following object definition specifies attributes for display on the raster screen and also specifies the color of the polygon's edges (using the WITH OUTLINE clause) for display on the color calligraphic display.

```
Pastel_Blue :=   ATTRIBUTES COLOR 3,.5,1 DIFFUSE .75 SPECULAR 5;
   Object :=     WITH ATTRIBUTES Pastel_Blue Outline 0
                 POLYGON
                    .
                    .
                    .
                 POLYGON ;
```

In this example, the shaded polygons on the raster display would be blue, with full saturation and .5 intensity. The specular value is .75 and the diffuse value is 5. The edges of the polygons are blue (Outline 0) when displayed on the CSM display.

Attribute Node Inputs

Inputs to the attribute node are as follows:

<1> accepts a real number as hue, a 2D vector as hue and saturation, or a 3D vector as hue, saturation, and intensity to specify COLOR for the front of the appropriate polygon(s) or both sides if no obverse attributes are given.

<2> accepts a real number as DIFFUSE

<3> accepts an integer as SPECULAR

<4>....<10> are undefined

<11>, <12>, and <13> correspond to <1>, <2>, and <3> but affect the obverse attributes if they exist.

If you send to input <1> or input <11> changing only the hue, the saturation and intensity return to the default values of s=1 and i=1. You cannot change just one value and keep the remaining values as they were before you made the change. Essentially, if you do not send a 3D vector, default values for the missing variables will be assumed.

For example, with the data definition

```
Dim_Red :=  ATTRIBUTES COLOR 130,1,.5 DIFFUSE .75 SPECULAR 8;
 Object :=  WITH ATTRIBUTES Dim_Red
            POLYGON
               .
               .
               .
            POLYGON;
```

If you sent 200 to input <1> of Dim_Red the resulting color parameter in the attribute node would be 200,1,1. To keep the saturation and intensity the same and change only the hue, you would send 200,1,.5 to input <1> of Dim_Red. This is the same if you want to change hue, saturation or intensity individually by sending a new value to the attribute node.

After changing the values in the attribute node, the changes will not be reflected until another rendering is requested.

Specifying Light Sources

Lights sources are specified with the ILLUMINATION command which creates "illumination nodes." Illumination nodes may be placed anywhere in the structure, allowing lights to be stationary or to rotate with the object or both. Illumination nodes are ignored during the calligraphic refresh and only those illumination nodes occurring in the descendent structure of a triggered solid- or surface-rendering operation node have any affect in shaded renderings. An unlimited number of light sources are valid for smooth-shaded renderings, but only the last illumination node encountered is used in creating flat-shaded renderings. Light sources are not used in wash-shaded (area-filled) images.

All light sources are presumed to be an infinite distance from the object; however, you can specify the direction at which they hit the object. This direction is multiplied by the current rotation matrix to determine the direction to the light in image space. If, after transformation, the light source appears to originate from behind the object, it will cause the whole object to be unilluminated (appear black), except, perhaps "glancing" specular highlights near the silhouette.

If no ILLUMINATION commands are given, a default white light at (0,0,-1) with an ambient proportion of 1.0 is assumed. If not specified, intensity and saturation default to 1. If only hue and saturation are specified, intensity defaults to 1.

Syntax:

```
[ name := ] ILLUMINATION x,y,z [ COLOR h [ ,s [ ,i ] ] ] [ AMBIENT a ] ;
```

where

x,y,z is a vector from the origin pointing toward the light source.

COLOR specifies the color of the light source by defining hue, saturation, and intensity. Color is specified identical to COLOR in the ATTRIBUTES command; the defaults are also the same.

AMBIENT controls the contribution of a light source to the ambient light. The net ambient lighting is determined by taking the sum of the products of the color and ambient proportion of each active light, dividing by the total number of active lights and then combining the result with the ambient input of the SHADINGENVIRONMENT function (in the next section). AMBIENT is defined by a real number between 0 and 1. Increasing *a* for one light increases its contribution to ambient light. Values outside this range are changed to 0 or 1. The default value for *a* is 1.0.

Changing the values of the SHADINGENVIRONMENT (explained in the next section) allows you to increase or decrease the intensity and color of the ambient light without the need to change each light source.

Whatever the values, if all active light sources have the same specified proportion, then all lights will contribute equally to the ambient light. Decreasing *a* for one light decreases its contribution to ambient light. Values outside this range are changed to 0 or 1. The default value is 1.

In the following example, the ILLUMINATION command

```
Light := ILLUMINATION 1,1,-1 COLOR 180;
```

creates a node which defines a yellow light over the right shoulder. Since saturation and intensity are not specified, the defaults s=1 and i=1 are assumed. A default of 1.0 for the ambient proportion is also assumed.

Since the illumination node occurs in the data structure (unlike the attribute node which exists alone in mass memory), it is not explicitly referenced by the polygon data node.

The hierarchy with an illumination node is shown in Figure 42.

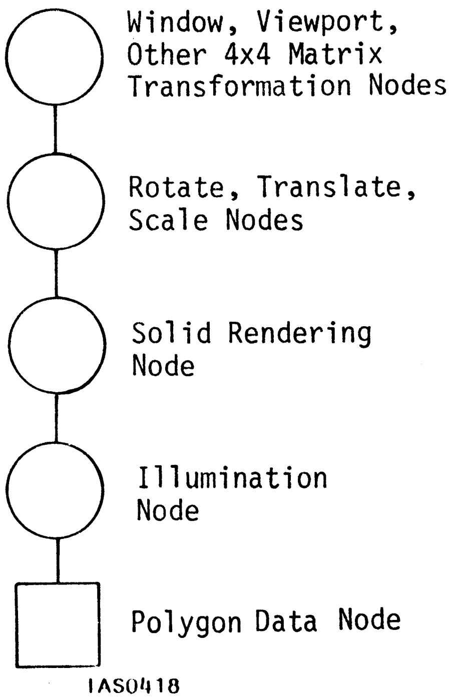


Figure 42. Hierarchy With Illumination Node

The illumination node must be under the rendering node in the display structure of the object.

Following is an example of how to use ILLUMINATION nodes. There are two lights in the example: SUN.LIGHT, which can be rotated independently of the object, and MOON.LIGHT, which rotates with the object. To achieve this:

1. Both lights are underneath the rendering node in the structure.

2. Placing the ILLUMINATION nodes underneath the rendering node implies that they will have the object's transformations also applied to them. This is what happens for MOON (sending a rotation to MOON.ROT will concatenate with the object's transformations).

3. This is not desired for the sun, so a FIELD_OF_VIEW (FOV) is inserted before the illumination node of SUN. This causes a rotation matrix sent to SUN.ROT to be the only matrix applied to SUN.LIGHT.

4. Inserting a 4D matrix (caused by the FOV) underneath a rendering node is not recommended. To avoid any problems, the 4D matrix defined by SUN.PERSP is identical to the 4D matrix defined by WORLD.PERSP and any change made to one (e.g., by a function network) should be made to both. Failure to follow this suggestion may result in bad renderings.

```
Sun :=    BEGIN_STRUCTURE  {a light which can be rotated independently}
          Persp := FOV 90 FRONT=2.2 BACK=3.6;
          LOOK AT 0,0,0 FROM 0,0,-3;
          Rot := SCALE BY 1;
          Light := ILLUMINATION 0,0,-1;
          END_STRUCTURE;

Moon :=   BEGIN_STRUCTURE  {a light which rotates with the object}
          Rot := SCALE BY 1;
          Light := ILLUMINATION 0,0,-1;
          END_STRUCTURE;

World :=  BEGIN_STRUCTURE
          Persp := FOV 45 FRONT=2.2 BACK=3.6;
          LOOK AT 0,0,0 FROM 0,0,-3;
          viewport horizontal=-1:1 vertical=-1:1 intensity=1:0;
          SET DEPTH_CLIPPING ON;
          Trans := TRANSLATE BY 0,0,0;
          Rot   := SCALE BY 1;
          Rendering := SURFACE_RENDERING;  {rendering node}
          instance object, Moon, Sun;
          END_STRUCTURE;

DISPLAY World;
```

Illumination Node Inputs

Inputs to the illumination node are:

<1> accepts a 3D vector as direction

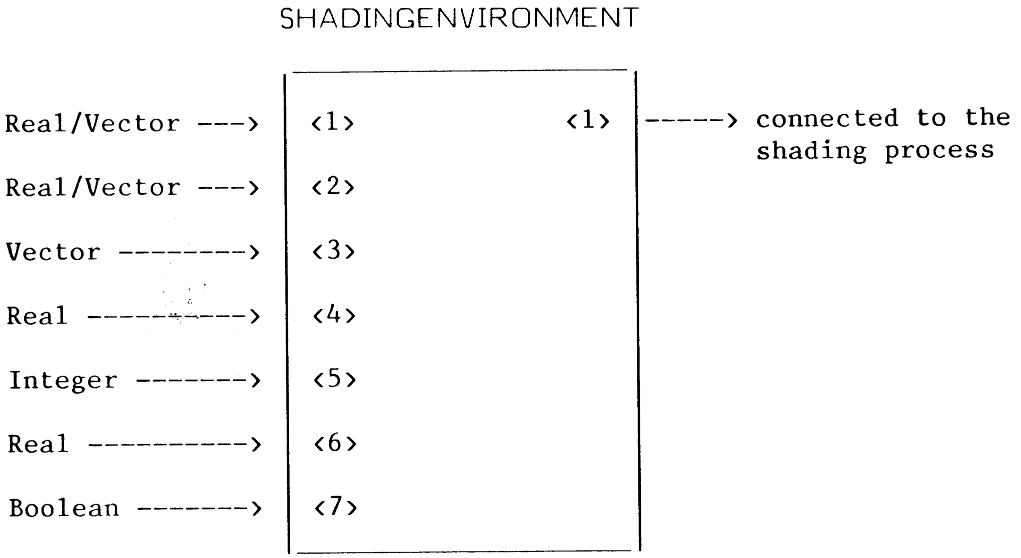
<2> accepts a real number as hue, a 2D vector as hue and saturation, and a 3D vector as hue, saturation, and intensity.

<3> accepts a real number as the ambient proportion

Like the attribute node, if you send a real number to input <2> to change only the hue, the saturation and intensity return to the default values of s=1 and i=1. You cannot change just one value and keep the remaining values as they were before you made the change. If you do not send a 3D vector, the defaults for the variables not specified are assumed.

The SHADINGENVIRONMENT Function

An Initial Function Instance, called SHADINGENVIRONMENT, allows you to control various non-dynamic factors of shaded renderings displayed on the raster screen. Sending values to the SHADINGENVIRONMENT function generally sets a parameter for the next requested shaded rendering rather than taking immediate effect. *Note that SHADINGENVIRONMENT is different from other PS 340 functions in that any input will activate the function independent of the other inputs.* SHADINGENVIRONMENT is like seven separate functions each with one input, but bundled together.



The inputs to the SHADINGENVIRONMENT function are as follows:

Ambient Color

<1> accepts a real number as hue, a 2D vector as hue and saturation, and a 3D vector as hue, saturation, and intensity, to specify the ambient color. Refer to the COLOR parameter of the ATTRIBUTES command for the meaning of the values. The ambient color is combined with the result obtained from the light sources to determine the color of ambient light. The default ambient color is white, with a default intensity of .25. The ambient color is analagous to the color reflected off a wall.

Background Color

<2> accepts a real number as hue, a 2D vector as hue and saturation, and or a 3D vector as hue, saturation, and intensity to specify the background color. Refer to the COLOR parameter of the ATTRIBUTES command for the meaning of the values. The raster screen will be colored with the background color prior to any shaded rendering. The default background color is black (0,0,0).

Raster Viewport

<3> accepts a 3D vector as the viewport on the raster image buffer where shaded renderings will be displayed. Raster viewports are always square, the lower left corner being given by the X and Y coordinates of the vector, and its size given by the Z coordinate, such that the upper right corner is at (x+z,y+z). Values are rounded to the nearest pixel. The default viewport is (80,0,480). The viewport is not intended for magnification of small parts of the calligraphic image, but for mapping the square vector display onto the rectangular raster display.

The viewport is also intended to allow multiple images to be generated side by side on the raster display. Thus, the largest recommended value for the viewport is (0,-80,640). The actual largest viewport is somewhat larger and depends on combinations of the three values. The image is clipped to the physical raster for which $0 \leq X < 640$ and $0 \leq Y < 480$.

Exposure

<4> accepts a real number as the exposure, controlling the overall brightness of the picture. The exposure is like that on a camera. If a picture is taken of an object with a very bright specular highlight, it may be so bright that the rest of the object is darkened. If three light sources exist, the object would be about three times brighter, making the object too bright. The exposure should be brought down to control this.

The exposure is multiplied by the intensity at each pixel and the result clipped to the maximum intensity. This enables the overall brightness of a rendering to be increased without causing bright spots to exceed maximum intensity (instead forming "plateaus" of maximum intensity). Note that this may cause changes in color on a plateau, where color has reached its maximum, but the others have not. Exposure values may vary between .3 and 3, values outside that range being changed to .3 or 3. The default exposure is 1.

Quality Level

<5> accepts an integer as quality level. The quality controls the number of pixels over which filtering applied. Jagged edges are characteristic of a raster display, so the fuzzier the edges, the better quality the picture. Values of 1, 3, 5, and 7 are allowed, meaning that the effect of coloring a pixel will be spread over a square of pixels with that number on a side, centered on the colored pixel. Because of anti-aliasing, pictures are good at quality 1. (The default value 1 is the typical choice.) Values of 3, 5, and 7 produce better quality renderings in terms of anti-aliasing but are time-consuming to process.

Depth Cueing

<6> accepts a real number in the range of 0 to 1 to control depth cueing in the shaded image (0 specifying no depth cueing and 1 specifying maximum depth cueing). As perceived depth from the viewer increases, the intensity of the colors decreases from maximum (1) at the nearest point to the given proportion of maximum at the farthest. Thus 0 gives a ramp ending in black at the back clipping plane, while 1 turns off the effect of depth cueing. The default is 0.2 giving a fairly large depth cueing effect.

Screen Wash

<7> accepts a Boolean, and is the only input to cause a visual effect immediately. True causes the whole physical raster screen to be filled with the current background color, while false just fills the currently defined viewport (clipped to the screen).

NOTE

If values are sent to the SHADINGENVIRONMENT function and your PS 340 is not configured with a raster system, an error message is issued. If values are again sent to the SHADINGENVIRONMENT function and no raster system exists, the function will destroy itself.

SUMMARY

The POLYGON command defines collections of polygons from which renderings can be created. This is a data-definition command that creates a polygon data node in the object's data structure.

Objects defined as polygons are the only objects that are eligible for rendering operations.

Rendering operations for vector displays can obtain a cross section of a displayed object, section an object relative to a sectioning plane, remove hidden line segments, and create shaded images of the object on a color raster screen.

Rendering operations for raster displays are flat shading, wash shading, and smooth shading.

Polygonal objects must be defined correctly to produce correct renderings.

POLYGON Command Syntax

Given,

```
        <vertex>  := [ S ] x,y,z [ N x,y,z ]
        <polygon> := [WITH [ATTRIBUTES name2] [OUTLINE h]]
                           POLYGON [COPLANAR] <vertex> ... <vertex>
```

The polygon command is:

```
[ Name := ] <Polygon> <Polygon> . . . <Polygon> ;
```

where:

- A vertex definition has the form [S] x,y,z [N x,y,z]

  where

  - S indicates that the edge drawn between the previous vertex and this one represents a soft edge of the polygon. If the S specifier is used for the first vertex in a polygon definition, the edge connecting the last vertex with the first is soft.

  - N indicates a normal to the surface with each vertex of the polygon. Normals are used only in smooth-shaded renderings. Normals must be specified for all vertices of a polygon or for none of the vertices of a polygon. Normals do not need to be present for all polygons in the object. If no normals are given for a polygon, they are defaulted to the same as the plane equation for the polygon.

  - x, y, and z are coordinates in a left-handed Cartesian system.

- WITH ATTRIBUTES is an option that assigns the attributes defined by the ATTRIBUTES command for all polygons until superseded by another WITH ATTRIBUTES clause.

- WITH OUTLINE is an option that specifies the color of the edges of a polygon on the color CSM display, or their intensity on a black and white display. A SET COLOR BLENDING node must be in the data structure to use this option.

- COPLANAR declares that the specified polygon and the one immediately preceding it has the same plane equation.

Defining Polygonal Objects

There is no syntactical limit on the number of polygon clauses in the group. POLYGON may be abbreviated POLYG.

Polygons are implicitly closed. The first vertex should not be repeated when defining a polygon.

No more than 250 vertices per polygon may be specified and no less than three.

The vertices of a polygon must be coplanar. Its plane equation is determined from any three non-colinear vertices.

Concave polygons are acceptable. Degenerate polygons and polygons that intersect themselves or others are not acceptable. No specific checks are made for these conditions.

Polygons are not pickable and polygon nodes have no inputs from which they can be modified with function networks.

Constructing Surfaces and Solids

Surfaces and solids can be defined. Solids enclose a volume of space, while surfaces do not.

In a solid, every edge of every polygon must coincide with the edge of a neighboring polygon.

For surfaces and solids, polygons are defined by listing their vertices in a clockwise order in the polygon clause.

In a solid, the common edge where two polygons join must run in opposite directions. This arrangement is essential to produce correct renderings. The system does not check for this condition.

A solid cannot contain three or more polygons which have a single edge is common, although surfaces may.

The SURFACE_RENDERING and SOLID_RENDERING commands determine the nature of a polygonal object.

Using the COPLANAR Option

Inner contours may be defined to create objects with holes or protrusions.

Vertices of inner contours must be listed in the opposite direction to the corresponding outer contour.

An inner contour should not be defined unless it is coplanar with some surrounding outer contour.

All members of a set of consecutive COPLANAR polygons are taken to have the same plane equation, that of the previous polygon not containing the COPLANAR option.

If COPLANAR is specified for the first polygon in a polygon list, it has no effect.

Using the Color Option (for Vector Displays)

Color for polygons displayed on the CSM monitor or intensity on the monochrome display are specified with the WITH OUTLINE *h* clause where *h*=0 or $1 < h \le 360$ for color, and $0 \le h \le 1$ for intensity.

To use the WITH OUTLINE clause to specify color, you must use the SET COLOR BLENDING command to create a node in the structure.

Color and intensity are specified for complete polygons, not individual edges. If you specify white polygons to be displayed on the CSM, all polygons must be white.

Using the Soft Edge Option (for Vector Displays)

The S specifier before a set of X,Y,Z coordinates indicates that the edge drawn between the previous vertex and this one represent a soft edge of the polygon.

Soft edges, declared with the S specifier in the polygon clause are invisible in hidden-line renderings except when they make up part of an object's profile.

Soft edges are positions in the original object.

If either edge of a common-edged pair is declared soft, the entire edge is considered soft.

Memory Usage

The rendering process requires that a contiguous block of mass memory be available as working storage. This memory must be explicitly reserved with the command RESERVE_WORKING_STORAGE *n*, where the current working storage is replaced with another containing at least *n* bytes. If *n* is less than or equal to 0 or there is insufficient memory to allocate a new block, the current working storage is disposed and no new block is allocated.

The best time to reserve working storage is immediately after booting; typically, you should reserve 200,000 to 400,000 bytes of working storage when you begin a session.

Working storage is not freed by the INITIALIZATION command.

In addition to the working storage space, extra mass memory is needed to create hidden-line renderings. This memory is referred to as transient memory and is automatically allocated and deallocated by the system.

Declaring the Object a Solid or a Surface

Syntaxes for the rendering commands are:

```
name := SOLID_RENDERING APPLIED TO name1;

name := SURFACE_RENDERING APPLIED TO name1;
```

where

- name1 names either (a) a POLYGON node, or (b) an ancestor of one or more POLYGON nodes.
- If (b) is the case, any rendering referring to name is performed on all of the POLYGON objects descended from name1 at once.

Only polygons nodes are used in renderings. Vector and character nodes occurring beneath a rendering node are ignored by the rendering operations.

Transformation nodes are lost in the rendering, but their effect is incorporated into the data nodes.

Rendering Node Connections

Rendering nodes have two inputs. Input <1> accepts an integer, a Boolean, or a string designating the rendering operation to be performed.

Input <2> accepts a Boolean to change the object definition from a surface to a solid or vice versa.

Rendering nodes also have an output which outputs a true if the rendering is displayed and a false if it is not displayed. You can connect this output via the CONNECT command to trigger some other action that was waiting on completion of the rendering process.

Rendering Node Inputs

Acceptable values for input <1> are :

0: Toggles between the current rendering and the original object.
1: Creates and displays a cross-section of an object defined by the sectioning plane (solids only).
2: Creates and displays a sectioned rendering.
3: Creates and displays a rendering using backface removal (solids only).
4: Creates and displays a rendering using hidden-line removal.
5: Generates a wash-shaded image on the raster display.
6: Generates a flat-shaded image on the raster display.
7: Generates a smooth-shaded image on the raster display.

String: Causes the current rendering to be saved under the name given in the string.
False: Sets the original view. The original descendant structure of the rendering operation node is displayed.
True: Sets the rendered view. The rendered view of the original descendent structure of the rendering operation node.

Acceptable values for input <2> are :

True: Declares the object to be a solid.
False: Declares the object to be a surface.

Establishing a Sectioning Plane

The SECTIONING_PLANE command creates a sectioning-plane node which indicates that a descendant POLYGON is a sectioning plane. The syntax is:

```
name := SECTIONING_PLANE APPLIED TO name1;
```

where

- name1 names either (a) a POLYGON command or (b) an ancestor of a POLYGON command.
- SECTIONING_PLANE may be abbreviated SECT.

The Sectioning Plane's Data Definition

The sectioning plane is the plane containing the polygon defined by the first POLYGON clause of the first polygon node encountered by the Display Processor as it traverses the branch beneath a sectioning-plane node.

The sectioning plane is the plane in which a specified POLYGON lies. The polygon itself need not intersect the object to be sectioned, as long as some part of the plane does.

No SOLID_RENDERING or SURFACE_RENDERING operation node, whether below or above the sectioning-plane node, may be an ancestor of a sectioning plane's defining POLYGON. The PS 340 interprets such polygons as objects to be rendered rather than as sectioning-plane definitions, and issues a "Sectioning plane not found" message when a sectioning attempt is made.

Saving a Rendering

A rendering is saved by a string sent to input <1> of the SOLID_RENDERING or SURFACE_RENDERING operation node. The string should specify the name of the node which is to contain the saved-rendering data. If the named node does not exist, it is created; if it does exist, the saved-rendering data replaces the original contents of the node.

All polygons in the rendering are taken into account in the saved rendering. It is not possible to exclude selected polygons or polygon data nodes from saved renderings.

Specifying Color and Highlights for Raster Displays

Specifying color, specular, and diffuse highlights, (called attributes) of a polygon for display on the raster screen, is done via the WITH ATTRIBUTES clause of the POLYGON command.

Given the polygon syntax:

```
[ name := ] <polygon> <polygon> . . . <polygon> ;
```

the attributes option is,

```
<polygon> :=  [WITH [ATTRIBUTES name2]  [OUTLINE h]] polygon
                <vertex>...<vertex>
```

The ATTRIBUTES Command

Given:

```
<attr> :=   [ Color h [ ,s [ ,i ] ] ]
            [ Diffuse d]
            [ Specular s ]
```

The ATTRIBUTES command is:

```
Name := ATTRIBUTES <attr> [ AND <attr> ] ;
```

meaning:

Color

Hue (h) specifies degrees around the color circle with 0 being pure blue, 120 pure red, and 240 pure green. Saturation (s) varies from 0 for no saturation (grays) to 1 for full saturation. Intensity (i) varies from 0 for no intensity (black) to 1 for full intensity.

If no color is specified, the default is white (s=0, i=1). If not specified, saturation and intensity default to 1.

Diffuse

Diffuse specifies the proportion of color contributed by diffuse reflection versus that contributed by specular reflection. Increasing ***d*** reduces the intensity of specular highlights, making the surface more matte; decreasing the intensity of specular highlights makes the surface more shiny with a value of 1 eliminating specular highlights entirely.

Values larger than 1 or less than 0 will be changed to 1 or 0. If no diffuse attribute is given, it defaults to 0.75.

The diffuse attribute only affects smooth-shaded renderings.

Specular

The specular attribute adjusts the concentration of specular highlights, with increasing values of ***s*** increasing their concentration.

Acceptable values of ***s*** are integers between 0 and 10. As with diffuse, the specular attribute only affects smooth-shaded renderings.

And

The attributes defined for the first <attr> specify attributes for front-facing polygons. The <attr> after the AND specify the attributes of backfacing polygons (applicable to surfaces only).

Attribute Node Inputs

Inputs to the attribute node are as follows:

<1> accepts a real number as hue, a 2D vector as hue and saturation, or a 3D vector as hue, saturation, and intensity to specify COLOR for the front of the appropriate polygon(s) or both sides if no obverse attributes are given.

<2> accepts a real number as DIFFUSE

<3> accepts a real number as SPECULAR

<4>....<10> are undefined

<11>, <12>, and <13> correspond to <1>, <2>, and <3> but affect the obverse attributes if they exist.

Specifying NORMALS

When a polygon is used to approximate a curved surface, the smooth appearance of the surface can be restored in a smooth-shaded rendering by approximating a surface using normals. A normal to the surface is given with each vertex of the polygon specified N x,y,z.

Specifying Light Sources

Lights may be stationary or rotate with the object or both.

If no ILLUMINATION commands are given, a default white light at (0,0,-1) with an ambient proportion of .25 is assumed. If not specified, intensity and saturation default to 1.

Syntax:

```
[ name := ] ILLUMINATION x,y,z [ COLOR h [ ,s [ ,i ] ] ] [ AMBIENT a ] ;
```

where

x,y,z is a vector from the origin pointing toward the light source.

COLOR specifies the color of the light source by defining hue, saturation, and intensity.

Color is specified identical to COLOR in the ATTRIBUTES command; the defaults are also the same.

AMBIENT controls the contribution of a light source to the ambient light and is defined by a real number between 0 and 1. Increasing *a* for one light, increases its contribution to ambient light. The default value for *a* is 1.

Illumination Node Inputs

Inputs to the illumination node are:

<1> accepts a 3D vector as direction

<2> accepts a real number as hue, a 2D vector as hue and saturation, and a 3D vector as hue, saturation, and intensity.

<3> accepts a real number as the ambient proportion

Like the attribute node, if you send a real number to Input <2> to change only the hue, the saturation and intensity return to the default values of s=1 and i=1.

The SHADINGENVIRONMENT Function

```
                        SHADINGENVIRONMENT
                     ______________________
                    |                      |
Real/Vector --->    | <1>              <1> | -----> connected to the
                    |                      |        shading process
Real/Vector --->    | <2>                  |
                    |                      |
Vector -------->    | <3>                  |
                    |                      |
Real ---------->    | <4>                  |
                    |                      |
Integer ------->    | <5>                  |
                    |                      |
Real ---------->    | <6>                  |
                    |                      |
Boolean ------->    | <7>                  |
                    |______________________|
```

The inputs to the SHADINGENVIRONMENT function are as follows:

Ambient Color

<1> accepts a real number as hue, a 2D vector as hue and saturation, and a 3D vector as hue, saturation, and intensity, to specify the ambient color. The default ambient color is white, with a default intensity of .25..

Background Color

<2> accepts a real number as hue, a 2D vector as hue and saturation, and or a 3D vector as hue, saturation, and intensity to specify the background color. The default background color is black (0,0,0).

Raster Viewport

<3> accepts a 3D vector as the viewport on the raster image buffer where shaded renderings will be displayed. Raster viewports are always square, the lower left corner being given by the X and Y coordinates of the vector, and its size given by the Z coordinate, such that the upper right corner is at (x+z,y+z). Values are rounded to the nearest pixel. The default viewport is (80,0,480).

Exposure

<4> accepts a real number as the exposure, controlling the overall brightness of the picture. Exposure values may vary between .3 and 3, values outside that range being changed to .3 or 3. The default exposure is 1.

Quality Level

<5> accepts an integer as quality level. The quality controls the number of pixels over which filtering applied. Values of 1, 3, 5, and 7 are allowed, meaning that the effect of coloring a pixel will be spread over a square of pixels with that number on a side, centered on the colored pixel. Pictures are good at quality 1.

Depth Cueing

<6> accepts a real number in the range of 0 to 1 to control depth cueing in the shaded image (0 specifying no depth cueing and 1 specifying maximum depth cueing). The default is 0.2 giving a fairly large depth cueing effect.

Screen Wash

<7> accepts a Boolean, and is the only input to cause a visual effect immediately. True causes the whole physical raster screen to be filled with the current background color, while false just fills the currently defined viewport (clipped to the screen).

PS 340 Systems Without a Raster Screen

If values are sent to the SHADINGENVIRONMENT function and your PS 340 is not configured with a raster screen, an error message is issued. If values are again sent to the shadingenvironment function and no raster system exists, the function will destroy itself.

Programming Example

```
INITIALIZE;

{reserve memory for rendering}

RESERVE_WORKING_STORAGE 120000;

{define a sectioning plane which can be rotated independently}

Spattributes := ATTRIBUTES;

Sect := BEGIN_STRUCTURE
        SECTIONING_PLANE;
        Trans := TRANSLATE BY 0,0,0;
        Rot   := ROTATE 0;
        With ATTRIBUTES Spattributes
        POLYGON  -0.9,-0.9,0.0 -0.9,0.9,0.0 0.9,0.9,0.0 0.9,-0.9,0.0
        POLYGON   0.1,0.0,0.0 0.1,0.0,-0.3 0.15,0.0,-0.3 0.0,0.0,-0.45
                 -0.15,0.0,-0.3 -0.1,0.0,-0.3 -0.1,0.0,0.0
        POLYGON   0.0,0.1,0.0 0.0,0.1,-0.3 0.0,0.15,-0.3 0.0,0.0,-0.45
                  0.0,-0.15,-0.3 0.0,-0.1,-0.3 0.0,-0.1,0.0;
        END_STRUCTURE;

{define a light which can be rotated independently}

Sunset := BEGIN_STRUCTURE
          FIELD_OF_VIEW 90 FRONT=2.2 BACK=3.6;
          LOOK_AT 0,0,0 FROM 0,0,-3;
          SET DEPTH_CLIPPING OFF;
          Rot := ROTATE 0;
          VECTOR  N=2 0,0,-.9 0,0,0;
          INSTANCE Sun;
          TRANSLATE 0,0,-.9;
          RATIONAL POLYNOMIAL .2,0,8 -.2,-.2,-8 0,.1,4 CHORDS=15;
          RATIONAL POLYNOMIAL .2,0,-8 -.2,-.2,8 0,.1,-4 CHORDS=15;
          VECTOR SEPARATE n=15  -.1,0 -.05,0 .05,0 .1,0 0,-.1 0,-.05 0,.05 0,.1
                                -.0707,-.0707 -.0354,-.0354 .0354,.0354 .0707,
                                 .0707-.0707,.0707 -.0354,.0354 .0354,-.0354
                                 .0707,-.0707;
          END_STRUCTURE;
Sun := ILLUMINATION 0,0,-1;

{define a light which can be rotated with the object}
```

```
Moonset := BEGIN_STRUCTURE
              SET DEPTH_CLIPPING OFF;
              Rot := ROTATE 0;
              VECTOR N=2 0,0,-.9 0,0,0;
              INSTANCE Moon;
              TRANSLATE 0,0,-.9;
              RATIONAL POLYNOMIAL .2,0,4 -.2,-.2,-4 0,.1,2 CHORDS=15;
              RATIONAL POLYNOMIAL .12,0,4 -.12,-.2,-4 0,.1,2 CHORDS=15;
              END_STRUCTURE;
Moon := ILLUMINATION 0,0,-1;

{set up a place to re-display a saved hidden-line picture}

Disphlview := MATRIX_4x4 1,0,0,0 0,1,0,0 0,0,0,0 0,0,1,1 THEN Hlview;

{set up initial display structure}

World := BEGIN_STRUCTURE
           Bits := SET CONDITION 1 ON;
           IF CONDITION 1 OFF THEN Disphlview;
           IF CONDITION 1 ON;
           Persp := FIELD_OF_VIEW 45 FRONT=2.2 BACK=3.6;
           LOOK AT 0,0,0 FROM 0,0,-3;
           VIEWPORT HORIZONTAL=-1:1 VERTICAL=-1:1 INTENSITY=1:1;
           SET DEPTH_CLIPPING ON;
           Trans := TRANSLATE by 0,0,0;
           Rot   := ROTATE 0;
           IF CONDITION 2 ON THEN Sect;
           Rendering := Surface; { rendering operation node, initially a surface }
           IF CONDITION 3 ON THEN Sunset;
           IF CONDITION 4 ON THEN Moonset;
           INSTANCE Object;
           END_STRUCTURE;
DISPLAY World;

{network to translate object}

A := F:ADDC;
CONNECT A<1>:<1>World.trans;
CONNECT A<1>:<2>A;
SEND V3D(0,0,0) TO <2>A;

{network to rotate/scale object}
```

```
M := F:CMUL;
CONNECT M<1>:<1>World.rot;
CONNECT M<1>:<1>M;
SEND M3D(1,0,0 0,1,0 0,0,1) TO <1>M;

{network to translate sectioning plane}

A2 := F:ADDC;
CONNECT A2<1>:<1>Sect.trans;
CONNECT A2<1>:<2>A2;
SEND V3D(0,0,0) TO <2>A2;

{network to rotate/scale sectioning plane}

M2 := F:CMUL;
CONNECT M2<1>:<1>Sect.rot;
CONNECT M2<1>:<1>M2;
SEND M3D(1,0,0 0,1,0 0,0,1) TO <1>M2;

{network to rotate sun}

Msun := F:CMUL;
CONNECT Msun<1>:<1>Sunset.rot;
CONNECT Msun<1>:<1>Msun;
SEND M3D(1,0,0 0,1,0 0,0,1) TO <1>Msun;

{network to rotate moon}

Mmoon := F:CMUL;
CONNECT Mmoon<1>:<1>Moonset.rot;
CONNECT Mmoon<1>:<1>Mmoon;
SEND M3D(1,0,0 0,1,0 0,0,1) TO <1>Mmoon;

{network selecting original or rendered view}

Original := F:CONSTANT;
CONNECT Original<1>:<1>World.rendering;
SEND FALSE TO <2>Original; { to switch to original view }
CONNECT DIALS<1>:<1>Original;
CONNECT DIALS<2>:<1>Original;
CONNECT DIALS<3>:<1>Original;
CONNECT DIALS<4>:<1>Original;
CONNECT DIALS<5>:<1>Original;
CONNECT DIALS<6>:<1>Original;
CONNECT DIALS<7>:<1>Original;
CONNECT DIALS<8>:<1>Original;
```

```
{color network}

Tripcolor := F:SYNC(5);
SETUP CNESS TRUE <2>Tripcolor;
SETUP CNESS TRUE <3>Tripcolor;
SETUP CNESS TRUE <4>Tripcolor;
SETUP CNESS TRUE <5>Tripcolor;
CONNECT Tripcolor<2>:<3>SHADINGENVIRONMENT;
CONNECT Tripcolor<3>:<7>SHADINGENVIRONMENT;
CONNECT Tripcolor<4>:<3>SHADINGENVIRONMENT;
CONNECT Tripcolor<5>:<2>SHADINGENVIRONMENT;
SEND V3D(600,440,40) TO <2>Tripcolor;
SEND FALSE TO <3>Tripcolor;
SEND V3D(0,0,0) TO <5>Tripcolor;

Suncolor := F:ACCUMULATE;
CONNECT Suncolor<1>:<2>Sun;
CONNECT Suncolor<1>:<2>SHADINGENVIRONMENT;
CONNECT Suncolor<1>:<1>Tripcolor;
SEND V3D(0,0,1) TO <2>Suncolor;
SEND 0 TO <3>Suncolor;
SEND V3D(20,.25,.25) TO <4>Suncolor;
SEND V3D(360,1,1) TO <5>Suncolor;
SEND V3D(0,0,0) TO <6>Suncolor;

Mooncolor := F:ACCUMULATE;
CONNECT Mooncolor<1>:<2>Moon;
CONNECT Mooncolor<1>:<2>SHADINGENVIRONMENT;
CONNECT Mooncolor<1>:<1>Tripcolor;
SEND V3D(0,0,1) TO <2>Mooncolor;
SEND 0 TO <3>Mooncolor;
SEND V3D(20,.25,.25) TO <4>Mooncolor;
SEND V3D(360,1,1) TO <5>Mooncolor;
SEND V3D(0,0,0) TO <6>Mooncolor;

Backgroundcolor := F:ACCUMULATE;
CONNECT Backgroundcolor<1>:<2>SHADINGENVIRONMENT;
CONNECT Backgroundcolor<1>:<1>Tripcolor;
CONNECT Backgroundcolor<1>:<5>Tripcolor;
SEND V3D(0,0,0) TO <2>Backgroundcolor;
SEND 0 TO <3>Backgroundcolor;
SEND V3D(20,.25,.25) TO <4>Backgroundcolor;
SEND V3D(360,1,1) TO <5>Backgroundcolor;
SEND V3D(0,0,0) TO <6>Backgroundcolor;
```

```
{mux the dials}

Dialmux := F:CROUTE(5);
CONNECT Dialmux<1>:<1>A;
CONNECT Dialmux<2>:<1>A2;
CONNECT Dialmux<3>:<1>Suncolor;
CONNECT Dialmux<4>:<1>Mooncolor;
CONNECT Dialmux<5>:<1>Backgroundcolor;

Dialmux2 := F:CROUTE(5);
CONNECT Dialmux2<1>:<2>M;
CONNECT Dialmux2<2>:<2>M2;
CONNECT Dialmux2<3>:<2>Msun;
CONNECT Dialmux2<4>:<2>Mmoon;

{network to translate in x}

Tx := F:XVEC;
CONNECT Tx<1>:<2>Dialmux;
CONNECT DIALS<1>:<1>Tx;

{network to translate in y}

Ty := F:YVEC;
CONNECT Ty<1>:<2>Dialmux;
CONNECT DIALS<2>:<1>Ty;

{network to translate in z}

Tz := F:ZVEC;
CONNECT Tz<1>:<2>Dialmux;
CONNECT DIALS<3>:<1>Tz;

{network to scale}

S := F:SCALE;
CONNECT S<1>:<2>Dialmux2;

Sa := F:ADDC;
CONNECT Sa<1>:<1>S;
SEND 1 TO <2>Sa;
CONNECT DIALS<4>:<1>Sa;

{network to rotate in x}
```

```
Rx := F:XROTATE;
CONNECT Rx<1>:<2>Dialmux2;

Sx := F:MULC;
CONNECT Sx<1>:<1>Rx;
SEND 100 TO <2>Sx;
CONNECT DIALS<5>:<1>Sx;

{network to rotate in y}

Ry := F:YROTATE;
CONNECT Ry<1>:<2>Dialmux2;

Sy := F:MULC;
CONNECT Sy<1>:<1>Ry;
SEND 100 TO <2>Sy;
CONNECT DIALS<6>:<1>Sy;

{network to rotate in z}

Rz := F:ZROTATE;
CONNECT Rz<1>:<2>Dialmux2;

Sz := F:MULC;
CONNECT Sz<1>:<1>Rz;
SEND -100 TO <2>Sz;
CONNECT DIALS<7>:<1>Sz;

{network to adjust BACK clipping plane}

Backclip := F:FOV;
CONNECT Backclip<1>:<1>World.persp;
SEND 45 TO <2>Backclip;
SEND 2.2 TO <3>Backclip;

Backclipaccum := F:ACCUM;
CONNECT Backclipaccum<1>:<1>Backclip;
CONNECT Backclipaccum<1>:<4>Backclip;
CONNECT DIALS<8>:<1>Backclipaccum;
SEND 3.6 TO <2>Backclipaccum;
SEND 0 TO <3>Backclipaccum;
SEND 1 TO <4>Backclipaccum;
SEND 30 TO <5>Backclipaccum;
SEND 2.2 TO <6>Backclipaccum;

{network to reset transformations}
```

```
Rs := F:SYNC(2);
CONNECT Rs<1>:<1>World.trans;
CONNECT Rs<1>:<2>A;
CONNECT Rs<2>:<1>World.rot;
CONNECT Rs<2>:<1>M;
CONNECT Rs<2>:<2>Rs;
SEND M3D(1,0,0 0,1,0 0,0,1) TO <2>Rs;

Rs2 := F:SYNC(2);
CONNECT Rs2<1>:<1>Sect.trans;
CONNECT Rs2<1>:<2>A2;
CONNECT Rs2<2>:<1>Sect.rot;
CONNECT Rs2<2>:<1>M2;
CONNECT Rs2<2>:<2>Rs2;
SEND M3D(1,0,0 0,1,0 0,0,1) TO <2>Rs2;

Rssun := F:CONSTANT;
CONNECT Rssun<1>:<1>Msun;
CONNECT Rssun<1>:<1>Sunset.rot;
SEND M3D(1,0,0 0,1,0 0,0,1) TO <2>Rssun;

Rsmoon := F:CONSTANT;
CONNECT Rsmoon<1>:<1>Mmoon;
CONNECT RsMoon<1>:<1>Moonset.rot;
SEND M3D(1,0,0 0,1,0 0,0,1) TO <2>Rsmoon;

R := F:CROUTE(5);
CONNECT R<1>:<1>Rs;
CONNECT R<2>:<1>Rs2;
CONNECT R<3>:<1>Rssun;
CONNECT R<4>:<1>Rsmoon;

{network to turn bits on and off}

Bits := F:CONSTANT;
CONNECT Bits<1>:<5>World.bits;

{network to send to object or sectioning plane}

Waylabel := F:INPUTS_CHOOSE(6);
CONNECT Waylabel<1>:<1>Flabel12;
SEND 'OBJECT'  TO <1>Waylabel;
SEND 'PLANE' TO <2>Waylabel;
SEND 'SUN' TO <3>Waylabel;
SEND 'MOON' TO <4>Waylabel;
SEND 'BACK' TO <5>Waylabel;
```

```
Diallabel := F:SYNC(9);
CONNECT Diallabel<1>:<1>Dlabel1;
CONNECT Diallabel<1>:<1>Diallabel;
CONNECT Diallabel<2>:<1>Dlabel2;
CONNECT Diallabel<2>:<2>Diallabel;
CONNECT Diallabel<3>:<1>Dlabel3;
CONNECT Diallabel<3>:<3>Diallabel;
CONNECT Diallabel<4>:<1>Dlabel4;
CONNECT Diallabel<4>:<4>Diallabel;
CONNECT Diallabel<5>:<1>Dlabel5;
CONNECT Diallabel<5>:<5>Diallabel;
CONNECT Diallabel<6>:<1>Dlabel6;
CONNECT Diallabel<6>:<6>Diallabel;
CONNECT Diallabel<7>:<1>Dlabel7;
CONNECT Diallabel<7>:<7>Diallabel;
CONNECT Diallabel<8>:<1>Dlabel8;
CONNECT Diallabel<8>:<8>Diallabel;
SEND 'X-TRANS' TO <1>Diallabel;
SEND 'X-TRANS' TO <1>Diallabel;
SEND 'HUE' TO <1>Diallabel;
SEND 'HUE' TO <1>Diallabel;
SEND 'HUE' TO <1>Diallabel;
SEND 'Y-TRANS' TO <2>Diallabel;
SEND 'Y-TRANS' TO <2>Diallabel;
SEND 'SAT' TO <2>Diallabel;
SEND 'SAT' TO <2>Diallabel;
SEND 'SAT' TO <2>Diallabel;
SEND 'Z-TRANS' TO <3>Diallabel;
SEND 'Z-TRANS' TO <3>Diallabel;
SEND 'INT' TO <3>Diallabel;
SEND 'INT' TO <3>Diallabel;
SEND 'INT' TO <3>Diallabel;
SEND 'SCALE' TO <4>Diallabel;
SEND 'X-ROT' TO <5>Diallabel;
SEND 'Y-ROT' TO <6>Diallabel;
SEND 'Z-ROT' TO <7>Diallabel;
SEND 'BACKCLIP' TO <8>Diallabel;

Way := F:SYNC(2);
CONNECT Way<2>:<1>Dialmux;
CONNECT Way<2>:<1>Dialmux2;
CONNECT Way<2>:<2>Way;
CONNECT Way<2>:<1>R;
CONNECT Way<2>:<6>Waylabel;
CONNECT Way<2>:<2>Bits;
CONNECT Way<2>:<9>Diallabel;
```

```
SEND FIX(1) TO <2>Way;
SEND FIX(2) TO <2>Way;
SEND FIX(3) TO <2>Way;
SEND FIX(4) TO <2>Way;
SEND FIX(5) TO <2>Way;
SEND TRUE TO <1>Way; {activate it}

{network to change from solid to surface}

Sslabel := F:BOOLEAN_CHOOSE;
CONNECT Sslabel<1>:<1>Flabel7;
SEND 'SOLID' TO <2>Sslabel;
SEND 'SURFACE' TO <3>Sslabel;

Issolid := F:NOP;
CONNECT Issolid<1>:<2>World.rendering;
CONNECT Issolid<1>:<1>Sslabel;

Ss := F:SYNC(2);
CONNECT Ss<2>:<2>Ss;
CONNECT Ss<2>:<1>Issolid;
SEND TRUE TO <2>Ss;
SEND FALSE TO <2>Ss;
SEND FALSE TO <1>Ss; { initially a surface }

{network to control rendering style}

Stylab := F:SYNC(2);
Styval := F:SYNC(2);
Style := F:CONST;
SEND 'HIDDEN' TO <1>Stylab;
SEND 'WASH' TO <1>Stylab;
SEND 'FLAT' TO <1>Stylab;
SEND 'SMOOTH' TO <1>Stylab;
SEND 'XSECTION' TO <1>Stylab;
SEND 'SECTION' TO <1>Stylab;
SEND 'BACKFACE' TO <1>Stylab;
SEND 'SAVE-SEC' TO <1>Stylab;
SEND 'SAVE-HL' TO <1>Stylab;
SEND FIX(4) TO <1>Styval;
SEND FIX(5) TO <1>Styval;
SEND FIX(6) TO <1>Styval;
SEND FIX(7) TO <1>Styval;
SEND FIX(1) TO <1>Styval;
SEND FIX(2) TO <1>Styval;
SEND FIX(3) TO <1>Styval;
SEND 'OBJECT' TO <1>Styval;
SEND 'HLVIEW' TO <1>Styval;
CONNECT Stylab<1>:<1>Stylab;
CONNECT Stylab<1>:<1>Flabel3;
CONNECT Styval<1>:<1>Styval;
CONNECT Styval<1>:<2>Style;

CONNECT Style<1>:<1>World.rendering;
SEND FIX(0) TO <2>Styval;
SEND FIX(0) TO <2>Stylab;

{ some useful viewports }

Piclab := F:SYNC(2);
Picval := F:SYNC(2);
SEND 'SQUARE' TO <1>Piclab;
SEND 'BIG-PIC' TO <1>Piclab;
SEND '1-OF-2' TO <1>Piclab;
SEND '2-OF-2' TO <1>Piclab;
SEND '1-OF-6' TO <1>Piclab;
SEND '2-OF-6' TO <1>Piclab;
SEND '3-OF-6' TO <1>Piclab;
SEND '4-OF-6' TO <1>Piclab;
SEND '5-OF-6' TO <1>Piclab;
SEND '6-OF-6' TO <1>Piclab;
SEND V3D (80,0,480) TO <1>Picval;
SEND V3D (0,-80,640) TO <1>Picval;
SEND V3D (0,80,320) TO <1>Picval;
SEND V3D (320,80,320) TO <1>Picval;
SEND V3D (5,240,210) TO <1>Picval;
SEND V3D (215,240,210) TO <1>Picval;
SEND V3D (425,240,210) TO <1>Picval;
SEND V3D (5,30,210) TO <1>Picval;
SEND V3D (215,30,210) TO <1>Picval;
SEND V3D (425,30,210) TO <1>Picval;

CONNECT Piclab<1>:<1>Piclab;
CONNECT Piclab<1>:<1>Flabel2;
CONNECT Picval<1>:<1>Picval;
CONNECT Picval<1>:<3>SHADINGENVIRONMENT;
CONNECT Picval<1>:<4>Tripcolor;

SEND 1 TO <2>Piclab;
SEND 1 TO <2>Picval;
```

```
{ buttons }

Fkmo := F:SWITCH;
CONNECT FKEYS<1>:<1>Fkmo;
CONNECT Fkmo<1>:<1>Style;
CONNECT Fkmo<2>:<2>Piclab;
CONNECT Fkmo<2>:<2>Picval;
CONNECT Fkmo<3>:<2>Stylab;
CONNECT Fkmo<3>:<2>styval;
CONNECT Fkmo<4>:<1>World.rendering;
CONNECT Fkmo<6>:<7>SHADINGENVIRONMENT;
CONNECT Fkmo<7>:<1>Ss;
CONNECT Fkmo<10>:<2>R;
CONNECT Fkmo<10>:<1>Original;
CONNECT Fkmo<11>:<1>Bits;
CONNECT Fkmo<11>:<1>Original;
CONNECT Fkmo<12>:<1>Way;

Fkm := F:INPUTS_CHOOSE(13);
CONNECT Fkm<1>:<2>Fkmo;
CONNECT FKEYS<1>:<13>Fkm;
SEND FIX(1) TO <1>Fkm;
SEND FIX(2) TO <2>Fkm;
SEND FIX(3) TO <3>Fkm;
SEND FIX(0) TO <4>Fkm;
SEND FIX(0) TO <5>Fkm;
SEND TRUE   TO <6>Fkm;
SEND FIX(7) TO <7>Fkm;
SEND FIX(0) TO <8>Fkm;
SEND FIX(9) TO <9>Fkm;
SEND V3D(0,0,0) TO <10>Fkm;
SEND FIX(11) TO <11>Fkm;
SEND FIX(12) TO <12>Fkm;

SEND 'RENDER' TO <1>Flabel1;
SEND 'TOGGLE' TO <1>Flabel4;
SEND 'CLEAR' TO <1>Flabel6;
SEND 'RESET' TO <1>Flabel10;
SEND 'ON/OFF' TO <1>Flabel11;

{ some useful colors }

Blue    := ATTRIBUTE COLOR 0;
Magenta := ATTRIBUTE COLOR 60;
Red     := ATTRIBUTE COLOR 120;
Yellow  := ATTRIBUTE COLOR 180;
Green   := ATTRIBUTE COLOR 240;
Cyan    := ATTRIBUTE COLOR 300;
White   := ATTRIBUTE COLOR 0,0,1;
```

```
{some other names for shadingenvironment}

Se := F:PASS(7);
CONNECT Se<1>:<1>SHADINGENVIRONMENT;
CONNECT Se<2>:<2>SHADINGENVIRONMENT;
CONNECT Se<3>:<3>SHADINGENVIRONMENT;
CONNECT Se<4>:<4>SHADINGENVIRONMENT;
CONNECT Se<5>:<5>SHADINGENVIRONMENT;
CONNECT Se<6>:<6>SHADINGENVIRONMENT;
CONNECT Se<7>:<7>SHADINGENVIRONMENT;

Ambient := F:PASS(1);
CONNECT Ambient<1>:<1>SHADINGENVIRONMENT;
Background := F:PASS(1);
CONNECT Background<1>:<2>SHADINGENVIRONMENT;
Rasterviewport := F:PASS(1);
CONNECT Rasterviewport<1>:<3>SHADINGENVIRONMENT;
Exposure := F:PASS(1);
CONNECT Exposure<1>:<4>SHADINGENVIRONMENT;
Quality := F:PASS(1);
CONNECT Quality<1>:<5>SHADINGENVIRONMENT;
Depth := F:PASS(1);
CONNECT Depth<1>:<6>SHADINGENVIRONMENT;
Screenwash := F:PASS(1);
CONNECT Screenwash<1>:<7>SHADINGENVIRONMENT;

{ make PS300 come up in shift line/local }
SEND 'R' TO <1>KBHANDLER;

{EOF}
```

SAMPLE PROGRAMS

EVANS & SUTHERLAND

February 1985
E&S #901172-095 P1

Evans & Sutherland assumes no responsibility for errors or inaccuracies in this document. It contains the most complete and accurate information available at the time of publication, and is subject to change without notice.

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

CONTENTS

INTRODUCTION

The sample programs in this section illustrate various applications of the PS 300 for design and analysis. Each application has two programs: a data structure file with an extension of .300 and a function network file with an extension of .FUN. A header section in each file explains what the application does. General practices illustrated in the sample programs can give you ideas for your own applications programs.

A great deal of care has been taken to make these programs examples of good PS 300 programming practices. In the data structure files, notice particularly the use of BEGIN_STRUCTURE ... END_STRUCTURE versus explicit naming. Notice also that the code is tabbed and commented in a way that makes it very easy to read.

The sample programs are listed on the following pages and also distributed in loadable form on magnetic tape. A selection in the command file TUTORIALS.COM lets you load the sample programs individually from the host.

ADAM.300

Programmed by: Neil Jon Harrington
Software Support Specialist
Evans & Sutherland
P.O. Box 8700
Salt Lake City, Utah 84108

Created: April 21, 1983
Last update:

Data Structure for an articulated anthropoid robot called ADAM (A Dial Activated Man). The data nodes (vector lists) for the sphere and the cylinder are not included in this file. The sphere has a radius of 1 and is centered at the origin. The base of the cylinder is at the origin lying in the XZ plane with the cylinder centered about the positive Y axis. The cylinder has a radius of 1 and a height of 1.

ADAM.FUN is the function network file that will articulate this structure.

```
INIT DISP;
DISP Adam;

Adam := BEGIN_S
          WINDOW X=-8.5:8.5 Y=-8.5:5.5
            FRONT=0 BACK=10;
          LOOK AT 0,0,0 FROM 0,0,-1;
Tran :=   TRAN 0,0,0;
Rot :=    ROT Y 0;
Scale :=  SCALE 1;
Pick :=   SET PICKING OFF;
          INST Upper_Body,Lower_Body;
        END_S;


Upper_Body := BEGIN_S
                SET PICK ID = B;
Rot :=          ROT 0;
{Chest}         SCALE .8,2.4,.7 THEN Cylinder;
                INST Right_Arm,Left_Arm,Head;
              END_S;
```

```
Right_Arm  := BEGIN_S
                TRAN -1.15,2.4,0;
{ Right Shoulder Joint }
                SET PICK ID = C;
Rot :=          ROT 0;
                INST Upper_Arm,Right_Lower_Arm;
              END_S;

Upper_Arm  := BEGIN_S
{Shoulder Ball} SCALE .3,.2,.2 THEN Sphere;
                TRAN 0,-2.1,0;
                SCALE .25,2.1,.25 THEN Cylinder;
              END_S;


Right_Lower_Arm := BEGIN_S
                     TRAN 0,-2.2,0;
Rot :=             ROT 0;
                     INST Lower_Arm,Right_Hand;
                   END_S;

Lower_Arm  := BEGIN_S
{Elbow}         SCALE .219 THEN Sphere; {7/32 rad.}
                TRAN 0,-1.8,0;
                SCALE .225,1.7,.225 THEN Cylinder;
              END_S;

Right_Hand := BEGIN_S
                TRAN 0,-1.9,0;
                SET PICK ID = D;
Rot :=          ROT 0 THEN Hand;
              END_S;

Hand := BEGIN_S
{Wrist}   SCALE .175 THEN Sphere;
{Hand}    TRAN 0,-.4,0;
          SCALE .15,.4,.25 THEN Sphere;
        END_S;


Left_Arm := BEGIN_S
              TRAN 1.15,2.4,0;
              SET PICK ID = C;
Rot :=        ROT 0;
              INST Upper_Arm,Left_Lower_Arm;
            END_S;
```

```
Left_Lower_Arm  := BEGIN_S
                     TRAN 0,-2.2,0;
Rot :=               ROT 0;
                     INST Lower_Arm,Left_Hand;
                   END_S;

Left_Hand := BEGIN_S
               TRAN 0,-1.9,0;
               SET PICK ID = D;
Rot :=         ROT 0 THEN Hand;
             END_S;


Head := BEGIN_S
          TRAN 0,2.4,0;
          SET PICK ID = A;
Rot :=    ROT 0;
{Neck}    SCALE .3,.6,.3 THEN Cylinder;
{Head}    TRAN 0,1.5,0;
          SCALE .6,1,.6 THEN Sphere;
        END_S;


Lower_Body := BEGIN_S
                SET PICK ID = B;
Rot :=          ROT 0;
                TRAN 0,-1,0;
                INST Right_Leg,Left_Leg;
{Waist & Hips}  SCALE .8,1,.7 THEN Cylinder;
              END_S;


Right_Leg := BEGIN_S
               TRAN -.45,-.25;
               SET PICK ID = E;
Rot :=         ROT 0;
               INST Upper_Leg,Right_Lower_Leg;
             END_S;

Upper_Leg := BEGIN_S
{Hip Joint}    SCALE .3 THEN Sphere;
               TRAN 0,-2.5,0;
               SCALE .35,2.5,.35 THEN Cylinder;
             END_S;

Right_Lower_Leg := BEGIN_S
                     TRAN 0,-2.6,0;
Rot :=               ROT x 0;
                     INST Lower_Leg,Right_Foot;
                   END_S;
```

```
Lower_Leg  := BEGIN_S
                INST Knee;
                TRAN 0,-2.6,0;
{Limb}          SCALE .3,2.5,.3 THEN Cylinder;
              END_S;

Knee := BEGIN_S
          ROT 90;
          TRAN 0,-.3,0;
          SCALE .15,.6,.15 THEN Cylinder;
        END_S;

Right_Foot := BEGIN_S
                TRAN 0,-2.75,0;
                SET PICK ID = F;
Rot :=          ROT 0 THEN Foot;
              END_S;

Foot := BEGIN_S
{Ankle}   SCALE .2 THEN Sphere;
          TRAN 0,-.2,.2;
          ROT x -90;
          SCALE .3,1,.2 THEN Cylinder;
        END_S;


Left_Leg := BEGIN_S
              TRAN .45,-.25;
              SET PICK ID = E;
Rot :=        ROT 0;
              INST Upper_Leg,Left_Lower_Leg;
            END_S;

Left_Lower_Leg := BEGIN_S
                    TRAN 0,-2.6,0;
Rot :=              ROT x 0;
                    INST Lower_Leg,Left_Foot;
                  END_S;

Left_Foot := BEGIN_S
               TRAN 0,-2.75,0;
               SET PICK ID = F;
Rot :=         ROT 0 THEN Foot;
             END_S;
```

ADAM.FUN

Programmed by: Neil Jon Harrington
Software Support Specialist
Evans & Sutherland
P.O. Box 8700
Salt Lake City, Utah 84108

Created: October, 1982
Last update: February, 1985

Network to modify the structure in ADAM.300. Point at the joint you want to rotate and the dials will be routed to modify that joint and others associated in that mode. If you want to rotate and translate the whole robot, point at the head.

```
{ Code generated by Network Editor 1.07 }
{ ADAM }
{ Frame-Prefix Macro-Prefix  }
{ Frame2:F2_ }
F2_P4:=F:CROUTE(6);
F2_P5:=F:CROUTE(6);
F2_P6:=F:DXROTATE;
F2_P7:=F:DXROTATE;
F2_P8:=F:DXROTATE;
F2_P9:=F:DXROTATE;
CONN F2_P4<3>:<1>F2_P6;
CONN F2_P4<5>:<1>F2_P7;
CONN F2_P5<3>:<1>F2_P8;
CONN F2_P5<5>:<1>F2_P9;
CONN F2_P6<1>:<1>Right_Lower_Arm.Rot;
CONN F2_P7<1>:<1>Right_Lower_Leg.Rot;
CONN F2_P8<1>:<1>Left_Lower_Arm.Rot;
CONN F2_P9<1>:<1>Left_Lower_Leg.Rot;
SEND 200 TO <3>F2_P7;
SEND 200 TO <3>F2_P8;
SEND 200 TO <3>F2_P9;
SEND 200 TO <3>F2_P6;
SEND 0 TO <2>F2_P7;
SEND 0 TO <2>F2_P8;
SEND 0 TO <2>F2_P9;
SEND 0 TO <2>F2_P6;
```

```
{ Frame3:F3_ }
F3_P11:=F:MULC;
F3_P12:=F:MULC;
F3_P13:=F:MULC;
F3_P14:=F:XROTATE;
F3_P15:=F:YROTATE;
F3_P16:=F:ZROTATE;
F3_P17:=F:CROUTE(6);
F3_P18:=F:MULC;
F3_P19:=F:MULC;
F3_P20:=F:MULC;
F3_P21:=F:MULC;
F3_P22:=F:MULC;
F3_P23:=F:MULC;
CONN F3_P11<1>:<1>F3_P14;
CONN F3_P12<1>:<1>F3_P15;
CONN F3_P13<1>:<1>F3_P16;
CONN F3_P14<1>:<2>F3_P17;
CONN F3_P15<1>:<2>F3_P17;
CONN F3_P16<1>:<2>F3_P17;
CONN F3_P17<1>:<1>F3_P18;
CONN F3_P17<2>:<1>F3_P19;
CONN F3_P17<3>:<1>F3_P20;
CONN F3_P17<4>:<1>F3_P21;
CONN F3_P17<5>:<1>F3_P22;
CONN F3_P17<6>:<1>F3_P23;
CONN F3_P18<1>:<1>Head.Rot;
CONN F3_P18<1>:<2>F3_P18;
CONN F3_P19<1>:<1>Upper_Body.Rot;
CONN F3_P19<1>:<2>F3_P19;
CONN F3_P20<1>:<1>Right_Arm.Rot;
CONN F3_P20<1>:<2>F3_P20;
CONN F3_P21<1>:<1>Right_Hand.Rot;
CONN F3_P21<1>:<2>F3_P21;
CONN F3_P22<1>:<1>Right_Leg.Rot;
CONN F3_P22<1>:<2>F3_P22;
CONN F3_P23<1>:<1>Right_Foot.Rot;
CONN F3_P23<1>:<2>F3_P23;
SEND 200 TO <2>F3_P11;
SEND 200 TO <2>F3_P12;
SEND 200 TO <2>F3_P13;
```

```
{ Frame4:F4_ }
F4_P24:=F:MULC;
F4_P25:=F:MULC;
F4_P26:=F:MULC;
F4_P27:=F:XROTATE;
F4_P28:=F:YROTATE;
F4_P29:=F:ZROTATE;
F4_P30:=F:CROUTE(6);
F4_P31:=F:CMUL;
F4_P32:=F:MULC;
F4_P33:=F:MULC;
F4_P34:=F:MULC;
F4_P35:=F:MULC;
F4_P36:=F:MULC;
CONN F4_P24<1>:<1>F4_P27;
CONN F4_P25<1>:<1>F4_P28;
CONN F4_P26<1>:<1>F4_P29;
CONN F4_P27<1>:<2>F4_P30;
CONN F4_P28<1>:<2>F4_P30;
CONN F4_P29<1>:<2>F4_P30;
CONN F4_P30<1>:<2>F4_P31;
CONN F4_P30<2>:<1>F4_P32;
CONN F4_P30<3>:<1>F4_P33;
CONN F4_P30<4>:<1>F4_P34;
CONN F4_P30<5>:<1>F4_P35;
CONN F4_P30<6>:<1>F4_P36;
CONN F4_P31<1>:<1>Adam.Rot;
CONN F4_P31<1>:<1>F4_P31;
CONN F4_P32<1>:<1>Lower_Body.Rot;
CONN F4_P32<1>:<2>F4_P32;
CONN F4_P33<1>:<1>Left_Arm.Rot;
CONN F4_P33<1>:<2>F4_P33;
CONN F4_P34<1>:<1>Left_Hand.Rot;
CONN F4_P34<1>:<2>F4_P34;
CONN F4_P35<1>:<1>Left_Leg.Rot;
CONN F4_P35<1>:<2>F4_P35;
CONN F4_P36<1>:<1>Left_Foot.Rot;
CONN F4_P36<1>:<2>F4_P36;
SEND 200 TO <2>F4_P25;
SEND 200 TO <2>F4_P26;
SEND 200 TO <2>F4_P24;
{ Picking Network:F5_ }
F5_P3:=F:PICKINFO;
F5_P39:=F:CHARCONVERT;
F5_P40:=F:SUBC;
```

```
CONN TABLETIN<4>:<1>Adam.Pick;
CONN TABLETIN<6>:<1>PICK;
CONN PICK<1>:<1>F5_P3;
CONN PICK<2>:<1>Adam.Pick;
CONN PICK<3>:<1>Adam.Pick;
CONN F5_P3<2>:<1>F5_P39;
CONN F5_P39<1>:<1>F5_P40;
SEND FIX(64) TO <2>F5_P40;
SEND FIX(1) TO <2>F5_P3;
{ Frame1:F1_ }
{ Setup cness true <2-3>P10 }
F1_P10:=F:SYNC(3);
SETUP CNESS TRUE <2>F1_P10;
SETUP CNESS TRUE <3>F1_P10;
CONN F1_P10<2>:<2>F2_P6;
CONN F1_P10<2>:<2>F2_P7;
CONN F1_P10<2>:<2>F2_P8;
CONN F1_P10<2>:<2>F2_P9;
CONN F1_P10<3>:<1>Right_Lower_Arm.Rot;
CONN F1_P10<3>:<1>Right_Lower_Leg.Rot;
CONN F1_P10<3>:<1>Left_Lower_Arm.Rot;
CONN F1_P10<3>:<1>Left_Lower_Leg.Rot;
CONN F1_P10<3>:<2>F3_P18;
CONN F1_P10<3>:<2>F3_P19;
CONN F1_P10<3>:<2>F3_P20;
CONN F1_P10<3>:<2>F3_P21;
CONN F1_P10<3>:<2>F3_P22;
CONN F1_P10<3>:<2>F3_P23;
CONN F1_P10<3>:<1>Head.Rot;
CONN F1_P10<3>:<1>Upper_Body.Rot;
CONN F1_P10<3>:<1>Right_Arm.Rot;
CONN F1_P10<3>:<1>Right_Hand.Rot;
CONN F1_P10<3>:<1>Right_Leg.Rot;
CONN F1_P10<3>:<1>Right_Foot.Rot;
CONN F1_P10<3>:<1>F4_P31;
CONN F1_P10<3>:<2>F4_P32;
CONN F1_P10<3>:<2>F4_P33;
CONN F1_P10<3>:<2>F4_P34;
CONN F1_P10<3>:<2>F4_P35;
CONN F1_P10<3>:<2>F4_P36;
CONN F1_P10<3>:<1>Adam.Rot;
CONN F1_P10<3>:<1>Lower_Body.Rot;
CONN F1_P10<3>:<1>Left_Arm.Rot;
CONN F1_P10<3>:<1>Left_Hand.Rot;
CONN F1_P10<3>:<1>Left_Leg.Rot;
CONN F1_P10<3>:<1>Left_Foot.Rot;
CONN FKEYS<1>:<1>F1_P10;
CONN DIALS<1>:<1>F3_P11;
CONN DIALS<2>:<1>F3_P12;
CONN DIALS<3>:<1>F3_P13;
CONN DIALS<4>:<2>F2_P4;
CONN DIALS<5>:<1>F4_P24;
CONN DIALS<6>:<1>F4_P25;
CONN DIALS<7>:<1>F4_P26;
CONN DIALS<8>:<2>F2_P5;
CONN F5_P40<1>:<1>F2_P4;
CONN F5_P40<1>:<1>F2_P5;
CONN F5_P40<1>:<1>F3_P17;
CONN F5_P40<1>:<1>F4_P30;
SEND FIX(1) TO <1>F2_P4;
SEND FIX(1) TO <1>F2_P5;
SEND FIX(1) TO <1>F3_P17;
SEND FIX(1) TO <1>F4_P30;
SEND 0 TO <2>F1_P10;
SEND M3D(1,0,0 0,1,0 0,0,1) TO <3>F1_P10;
SEND M3D(1,0,0 0,1,0 0,0,1) TO <1>F1_P10;
```

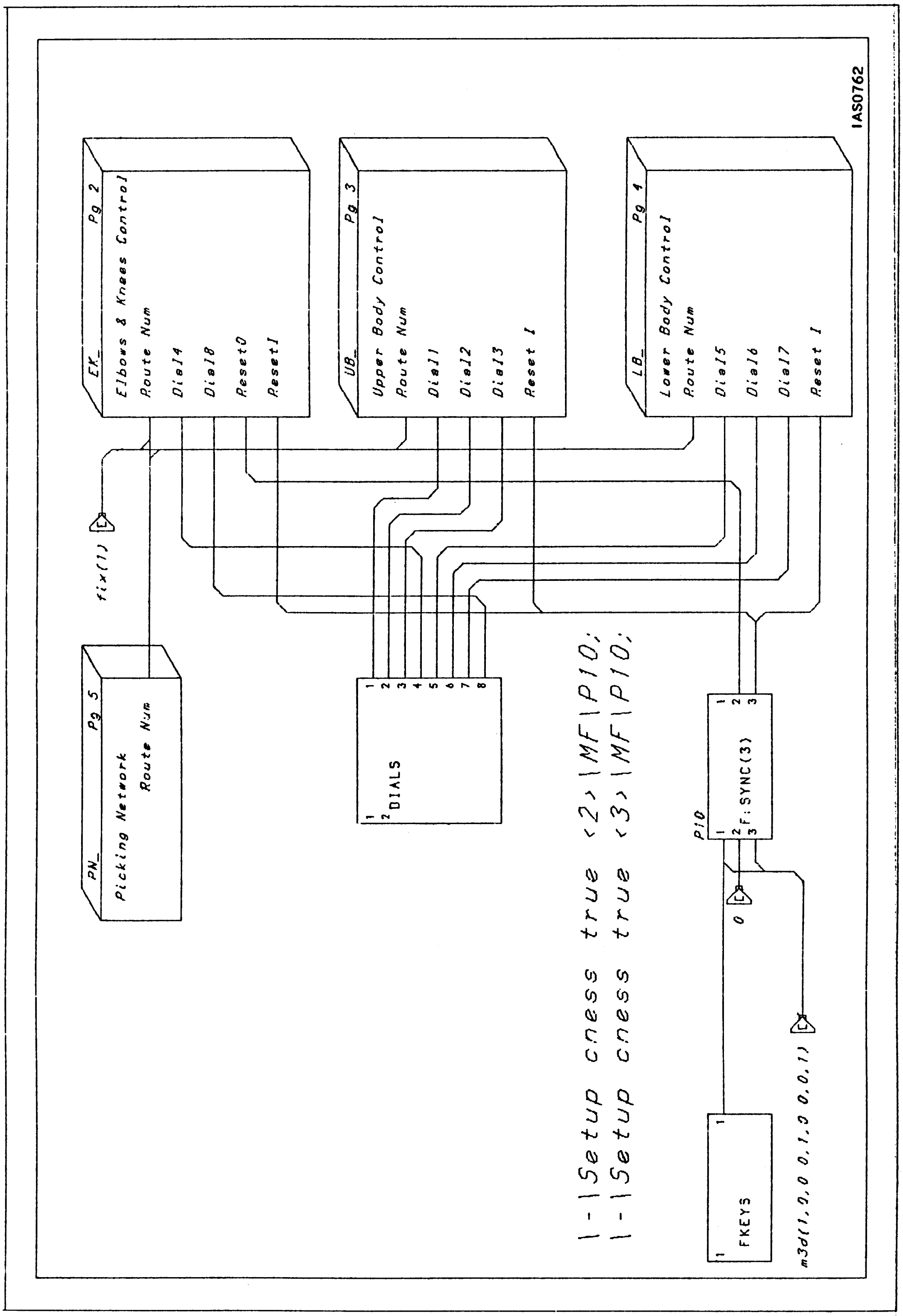


Figure 1. ADAM.FUN (Sheet 1 of 5)
(Function Network for ADAM.300)

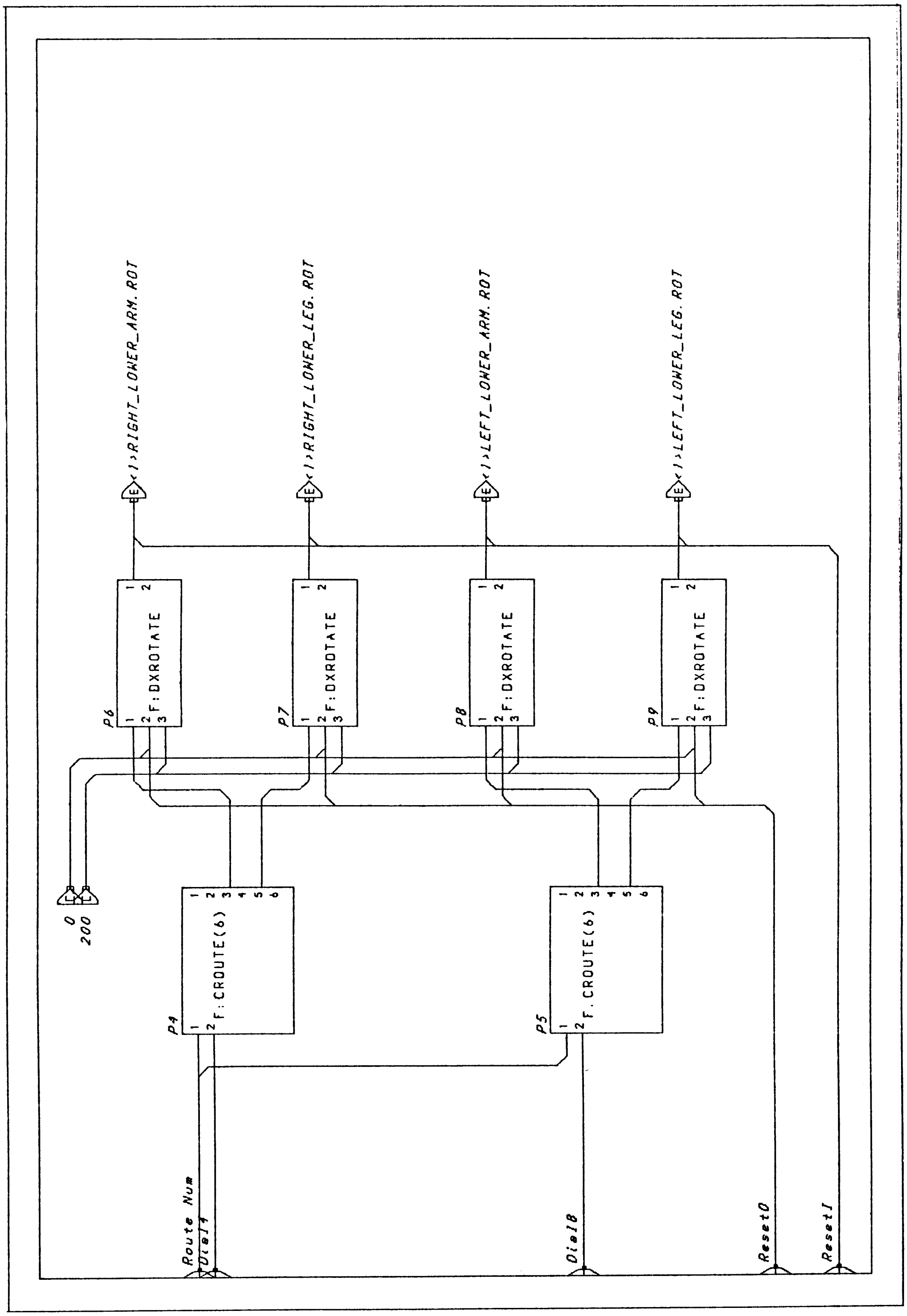


Figure 1. ADAM.FUN (Sheet 2 of 5)

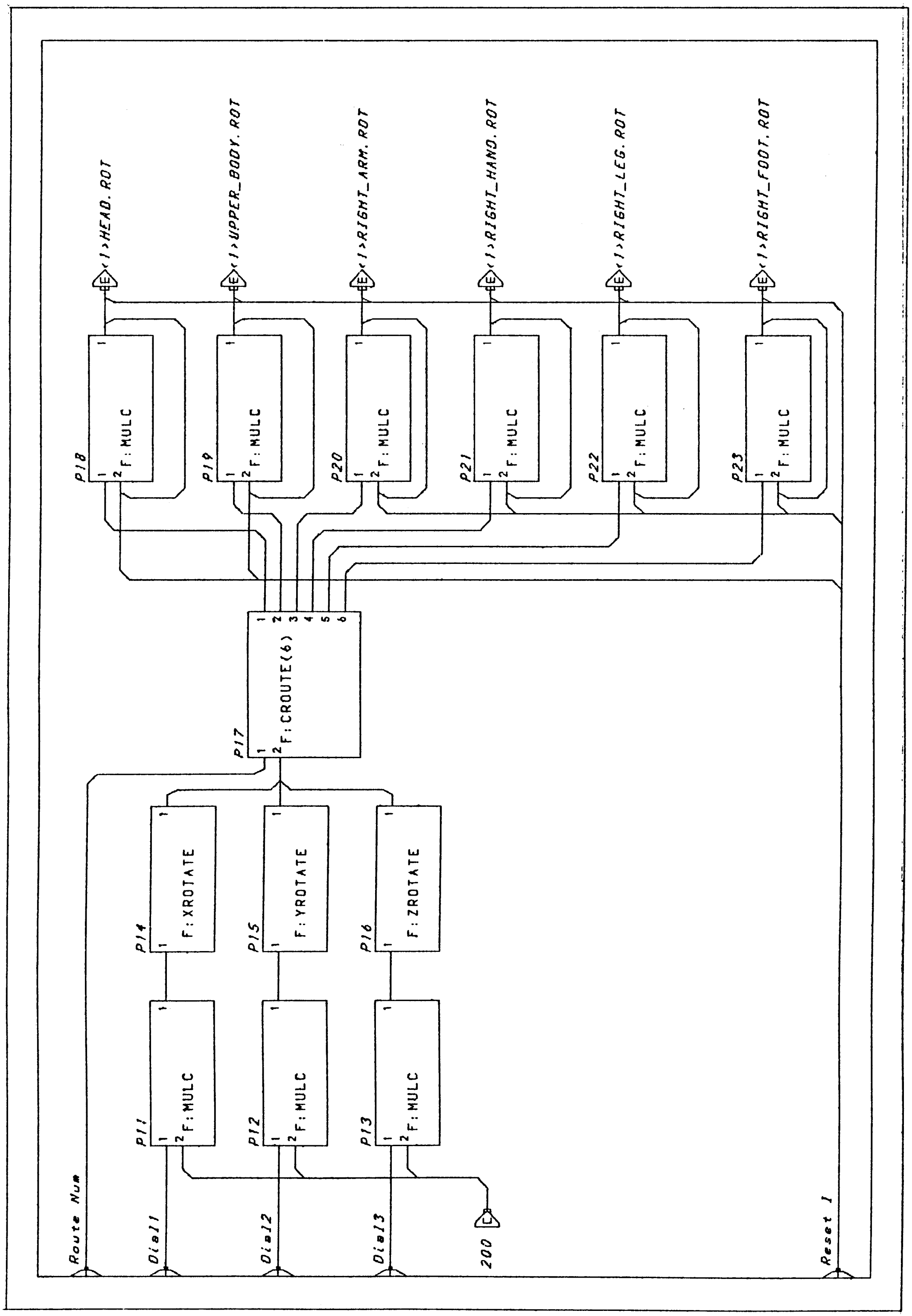


Figure 1. ADAM.FUN (Sheet 3 of 5)

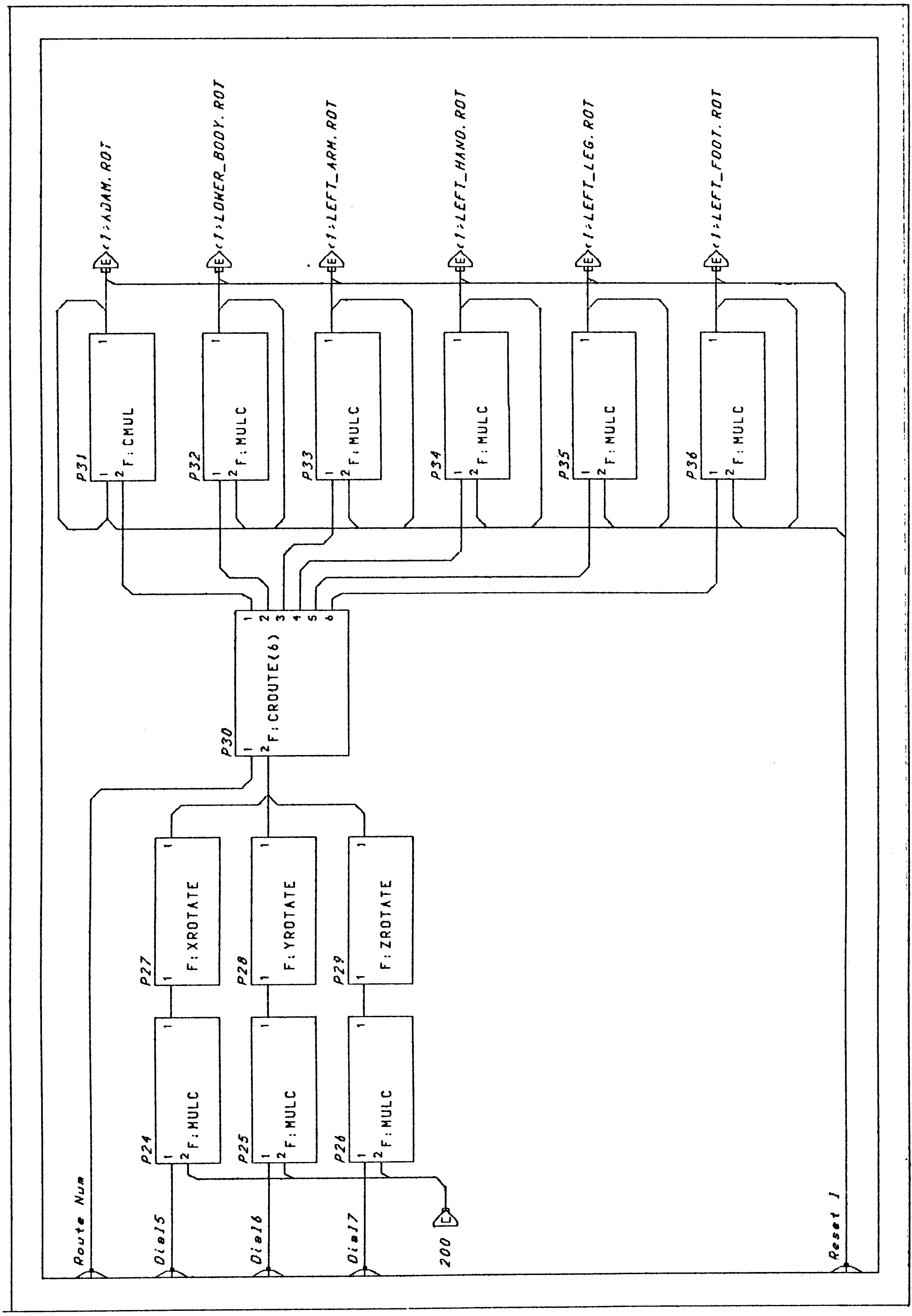


Figure 1. ADAM.FUN (Sheet 4 of 5)

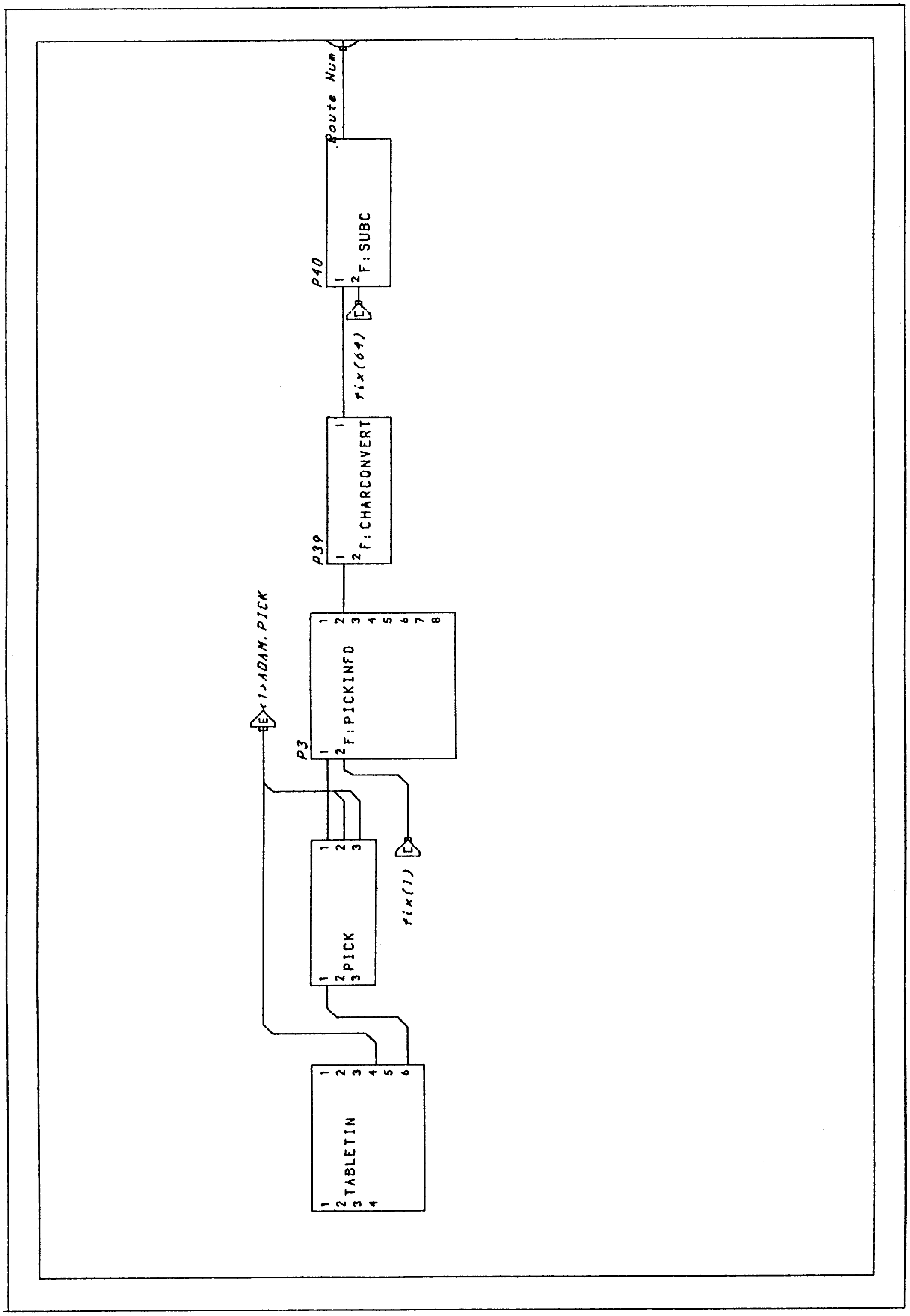


Figure 1. ADAM.FUN (Sheet 5 of 5)

COLLISION.300

Programmed by: Neil Jon Harrington
Software Support Specialist
Evans & Sutherland
P.O. Box 8700
Salt Lake City, Utah 84108

Created: October, 1984
Last update: February, 1985

PS 300 data structure, consisting of a ball in a box. The function network in Collision.fun modifies this structure to simulate the ball bouncing in the box with no gravity and elastic collisions.

```
INIT DISP;
DISP Collision;

Collision := BEGIN_S
               SET INTENSITY ON .75:1;
               SET DEPTH_CLIPPING ON;
               FOV 70 FRONT = 1.4 BACK = 5;
               LOOK AT 0,0,0 FROM 1.5,1.3,-2.4;
Yrot :=        ROT 0;
               SET COLOR 240,1 THEN Box;
               SET COLOR 120,1 THEN Ball;
               SET COLOR 0,1 THEN Path;
             END_S;

Box := SCALE 1 THEN Cube;

Ball := BEGIN_S
Tran :=   TRAN 0,0,0;
Rot :=    ROT 0;
Scale :=  SCALE .1 THEN Sphere;
        END_S;
```

```
Path := VEC n=10000 0,0,0;

Cube := VEC Item n=16
           P -1, 1,-1   L  1, 1,-1
           L  1,-1,-1   L -1,-1,-1
           P  1, 1,-1   L  1, 1, 1
           L  1,-1, 1   L  1,-1,-1
           P  1, 1, 1   L -1, 1, 1
           L -1,-1, 1   L  1,-1, 1
           P -1, 1, 1   L -1, 1,-1
           L -1,-1,-1   L -1,-1, 1;
```

COLLISION.FUN

Programmed by: Neil Jon Harrington
Software Support Specialist
Evans & Sutherland
P.O. Box 8700
Salt Lake City, Utah 84108

Created: October, 1984
Last update: February, 1985

Network to modify structure created in Collision.300. See description in that file.

```
{ Code generated by Network Editor 1.07 }
{ COLLISION }
{ Frame-Prefix Macro-Prefix  }
{ Framel:M1$F1_ }
M1$F1_P1:=F:INPUTS_CHOOSE(13);
M1$F1_P2:=F:ROUTE(12);
CONN M1$F1_P1<1>:<2>M1$F1_P2;
SEND TRUE TO <1>M1$F1_P1;
SEND TRUE TO <2>M1$F1_P1;
SEND TRUE TO <3>M1$F1_P1;
SEND TRUE TO <4>M1$F1_P1;
SEND TRUE TO <5>M1$F1_P1;
SEND TRUE TO <6>M1$F1_P1;
SEND TRUE TO <7>M1$F1_P1;
SEND TRUE TO <8>M1$F1_P1;
SEND TRUE TO <9>M1$F1_P1;
SEND TRUE TO <10>M1$F1_P1;
SEND TRUE TO <11>M1$F1_P1;
SEND TRUE TO <12>M1$F1_P1;
{ Motion Control:F2_ }
F2_P2:=F:SYNC(4);
F2_P6:=F:LIMIT;
F2_P7:=F:LIMIT;
F2_P8:=F:LIMIT;
F2_P9:=F:BROUTEC;
F2_P10:=F:BROUTEC;
F2_P11:=F:BROUTEC;
F2_P12:=F:MULC;
```

```
F2_P13:=F:MULC;
F2_P14:=F:MULC;
F2_P15:=F:XVECTOR;
F2_P16:=F:YVECTOR;
F2_P17:=F:ZVECTOR;
F2_P18:=F:ADD;
F2_P19:=F:ADD;
F2_P20:=F:ADD;
F2_P41:=F:ACCUMULATE;
F2_P42:=F:ACCUMULATE;
F2_P43:=F:ACCUMULATE;
F2_P38:=F:ADD;
F2_P39:=F:ADD;
CONN F2_P2<2>:<1>F2_P18;
CONN F2_P2<3>:<1>F2_P19;
CONN F2_P2<4>:<1>F2_P20;
CONN F2_P6<1>:<1>F2_P15;
CONN F2_P6<1>:<2>F2_P18;
CONN F2_P6<3>:<1>F2_P9;
CONN F2_P7<1>:<1>F2_P16;
CONN F2_P7<1>:<2>F2_P19;
CONN F2_P7<3>:<1>F2_P10;
CONN F2_P8<1>:<1>F2_P17;
CONN F2_P8<1>:<2>F2_P20;
CONN F2_P8<3>:<1>F2_P11;
CONN F2_P9<1>:<2>F2_P2;
CONN F2_P9<2>:<1>F2_P12;
CONN F2_P10<1>:<3>F2_P2;
CONN F2_P10<2>:<1>F2_P13;
CONN F2_P11<1>:<4>F2_P2;
CONN F2_P11<2>:<1>F2_P14;
CONN F2_P12<1>:<2>F2_P2;
CONN F2_P12<1>:<2>F2_P9;
CONN F2_P12<1>:<2>F2_P41;
CONN F2_P13<1>:<3>F2_P2;
CONN F2_P13<1>:<2>F2_P10;
CONN F2_P13<1>:<2>F2_P42;
CONN F2_P14<1>:<4>F2_P2;
CONN F2_P14<1>:<2>F2_P11;
CONN F2_P14<1>:<2>F2_P43;
CONN F2_P15<1>:<1>F2_P38;
CONN F2_P16<1>:<2>F2_P38;
CONN F2_P17<1>:<2>F2_P39;
CONN F2_P18<1>:<1>F2_P6;
CONN F2_P19<1>:<1>F2_P7;
CONN F2_P20<1>:<1>F2_P8;
CONN F2_P38<1>:<1>F2_P39;
CONN F2_P39<1>:<1>Ball.Tran;
CONN F2_P41<1>:<2>F2_P9;
CONN F2_P42<1>:<2>F2_P10;
CONN F2_P43<1>:<2>F2_P11;
SEND -.9 TO <3>F2_P6;
SEND -.9 TO <3>F2_P7;
SEND -.9 TO <3>F2_P8;
SEND .9 TO <2>F2_P6;
SEND .9 TO <2>F2_P7;
SEND .9 TO <2>F2_P8;
SEND 0 TO <6>F2_P41;
SEND 0 TO <6>F2_P42;
SEND 0 TO <6>F2_P43;
SEND 10 TO <5>F2_P41;
SEND 10 TO <5>F2_P42;
SEND 10 TO <5>F2_P43;
SEND .1 TO <4>F2_P41;
SEND .1 TO <4>F2_P42;
SEND .1 TO <4>F2_P43;
SEND 0 TO <3>F2_P41;
SEND 0 TO <3>F2_P42;
SEND 0 TO <3>F2_P43;
SEND .03 TO <4>F2_P2;
SEND .03 TO <2>F2_P11;
SEND .03 TO <2>F2_P43;
SEND .02 TO <3>F2_P2;
SEND .02 TO <2>F2_P10;
SEND .02 TO <2>F2_P42;
SEND .01 TO <2>F2_P2;
SEND .01 TO <2>F2_P9;
SEND .01 TO <2>F2_P41;
SEND 0 TO <2>F2_P18;
SEND 0 TO <2>F2_P19;
SEND 0 TO <2>F2_P20;
SEND -1 TO <2>F2_P12;
SEND -1 TO <2>F2_P13;
SEND -1 TO <2>F2_P14;
{ Clock Control:F3_ }
F3_P1:=F:CLFRAMES;
F3_P22:=F:CONSTANT;
F3_P23:=F:EDGE_DETECT;
F3_P25:=F:ACCUMULATE;
F3_P27:=F:FIX;
F3_P28:=F:XOR;
F3_P65:=F:XROTATE;
CONN F3_P1<2>:<1>F3_P22;
CONN F3_P1<2>:<1>F3_P65;
CONN F3_P1<2>:<5>F3_P1;
CONN F3_P22<1>:<1>F3_P23;
CONN F3_P25<1>:<1>F3_P27;
CONN F3_P27<1>:<1>F3_P1;
CONN F3_P28<1>:<6>F3_P1;
CONN F3_P28<1>:<2>F3_P28;
CONN F3_P65<1>:<1>Ball.Rot;
SEND FIX(0) TO <2>F3_P1;
SEND FALSE TO <3>F3_P1;
SEND FIX(1) TO <4>F3_P1;
SEND FIX(0) TO <5>F3_P1;
SEND FALSE TO <6>F3_P1;
SEND FIX(1) TO <1>F3_P1;
SEND FALSE TO <1>F3_P23;
SEND TRUE TO <2>F3_P22;
SEND TRUE TO <2>F3_P23;
SEND 1 TO <2>F3_P25;
SEND 1 TO <3>F3_P25;
SEND 10 TO <4>F3_P25;
SEND 60 TO <5>F3_P25;
SEND 1 TO <6>F3_P25;
SEND FALSE TO <2>F3_P28;
{ Frame1:M2$F1_ }
{ Box Size }
M2$F1_P1:=F:ACCUMULATE;
M2$F1_P2:=F:XVECTOR;
M2$F1_P3:=F:YVECTOR;
M2$F1_P4:=F:ZVECTOR;
M2$F1_P5:=F:CONSTANT;
M2$F1_P6:=F:NOP;
CONN M2$F1_P2<1>:<1>M2$F1_P1;
CONN M2$F1_P3<1>:<1>M2$F1_P1;
CONN M2$F1_P4<1>:<1>M2$F1_P1;
CONN M2$F1_P5<1>:<2>M2$F1_P1;
SEND V3D(.01,.01,.01) TO <6>M2$F1_P1;
SEND 1 TO <4>M2$F1_P1;
SEND V3D(1,1,1) TO <2>M2$F1_P1;
SEND V3D(1,1,1) TO <2>M2$F1_P5;
SEND V3D(1,1,1) TO <5>M2$F1_P1;
SEND V3D(1,1,1) TO <1>M2$F1_P6;
SEND 0 TO <3>M2$F1_P1;
{ Box/Ball Size:F4_ }
F4_P31:=F:SUBC;
F4_P32:=F:SCALE;
F4_P33:=F:PARTS;
F4_P34:=F:PARTS;
F4_P35:=F:MULC;
F4_P44:=F:DSCALE;
F4_P45:=F:VEC;
F4_P46:=F:VEC;
F4_P47:=F:FETCH;
VAR Box_Size;
```

```
CONN M2$F1_P1<1>:<1>F4_P32;
CONN M2$F1_P1<1>:<1>F4_P31;
CONN M2$F1_P1<1>:<1>Box_Size;
CONN M2$F1_P5<1>:<1>F4_P32;
CONN M2$F1_P5<1>:<1>F4_P31;
CONN M2$F1_P5<1>:<1>Box_Size;
CONN M2$F1_P6<1>:<1>F4_P32;
CONN M2$F1_P6<1>:<1>F4_P31;
CONN M2$F1_P6<1>:<1>Box_Size;
CONN F4_P31<1>:<1>F4_P33;
CONN F4_P31<1>:<1>F4_P35;
CONN F4_P32<1>:<1>Box;
CONN F4_P35<1>:<1>F4_P34;
CONN F4_P44<1>:<1>Ball.Scale;
CONN F4_P44<2>:<3>F4_P44;
CONN F4_P44<2>:<1>F4_P45;
CONN F4_P44<2>:<2>F4_P45;
CONN F4_P44<2>:<2>F4_P46;
CONN F4_P45<1>:<1>F4_P46;
CONN F4_P46<1>:<1>F4_P47;
CONN F4_P46<1>:<2>F4_P31;
CONN F4_P47<1>:<1>F4_P31;
SEND V3D(1,1,1) TO <1>Box_Size;
SEND 'Box_Size' TO <2>F4_P47;
SEND .05 TO <5>F4_P44;
SEND 1 TO <4>F4_P44;
SEND .1 TO <2>F4_P44;
SEND .1 TO <3>F4_P44;
SEND V3D(.1,.1,.1) TO <2>F4_P31;
SEND -1 TO <2>F4_P35;
{ Path:F5_ }
F5_P49:=F:CBROUTE;
F5_P50:=F:XOR;
CONN F5_P49<1>:<append>Path;
CONN F5_P50<1>:<2>F5_P50;
CONN F5_P50<1>:<1>F5_P49;
SEND TRUE TO <2>F5_P50;
SEND TRUE TO <1>F5_P49;
{ Labels:F6_ }
SEND 'RESET' TO <1>FLABEL11;
SEND 'STRT/STP' TO <1>FLABEL10;
SEND 'SLOWER' TO <1>FLABEL4;
SEND 'FASTER' TO <1>FLABEL3;
SEND 'CLR PATH' TO <1>FLABEL2;
SEND 'TRACE?' TO <1>FLABEL1;
SEND 'BALLSIZE' TO <1>DLABEL8;
SEND 'Z VEL' TO <1>DLABEL7;
SEND 'Y VEL' TO <1>DLABEL6;
SEND 'X VEL' TO <1>DLABEL5;
SEND 'OS Y ROTATE' TO <1>DLABEL4;
SEND 'Z SIZE' TO <1>DLABEL3;
SEND 'Y SIZE' TO <1>DLABEL2;
SEND 'X SIZE' TO <1>DLABEL1;
{ Frame1:F1_ }
F1_P48:=F:DYROTATE;
CONN DIALS<1>:<1>M2$F1_P2;
CONN DIALS<2>:<1>M2$F1_P3;
CONN DIALS<3>:<1>M2$F1_P4;
CONN DIALS<4>:<1>F1_P48;
CONN DIALS<5>:<1>F2_P41;
CONN DIALS<6>:<1>F2_P42;
CONN DIALS<7>:<1>F2_P43;
CONN DIALS<8>:<1>F4_P44;
CONN M1$F1_P2<1>:<1>F5_P50;
CONN M1$F1_P2<2>:<clear>Path;
CONN M1$F1_P2<3>:<1>F3_P25;
CONN M1$F1_P2<4>:<1>F3_P25;
CONN M1$F1_P2<10>:<1>F3_P28;
CONN M1$F1_P2<11>:<1>M2$F1_P5;
CONN FKEYS<1>:<13>M1$F1_P1;
CONN FKEYS<1>:<1>M1$F1_P2;
CONN F1_P48<1>:<1>Collision.Yrot;
CONN F2_P2<1>:<1>F3_P23;
CONN F2_P39<1>:<2>F5_P49;
CONN F3_P23<2>:<1>F2_P2;
CONN F4_P33<1>:<2>F2_P6;
CONN F4_P33<2>:<2>F2_P7;
CONN F4_P33<3>:<2>F2_P8;
CONN F4_P34<1>:<3>F2_P6;
CONN F4_P34<2>:<3>F2_P7;
CONN F4_P34<3>:<3>F2_P8;
SEND 2 TO <4>M1$F1_P1;
SEND -2 TO <3>M1$F1_P1;
SEND FIX(10000) TO <2>M1$F1_P1;
SEND 200 TO <3>F1_P48;
SEND 0 TO <2>F1_P48;
```

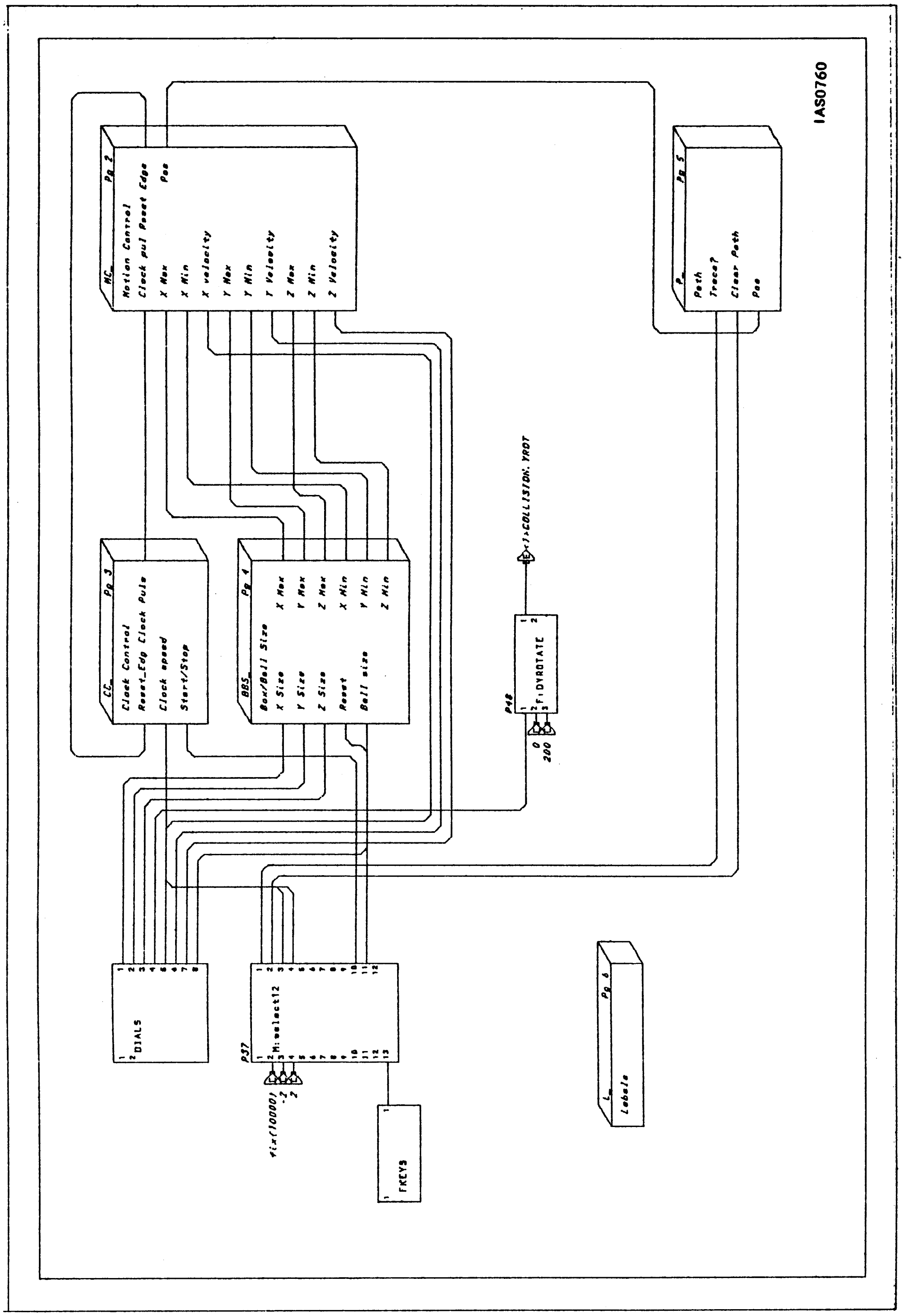


Figure 2. COLLISION.FUN (Sheet 1 of 6)
(Function Network for COLLISION.300)

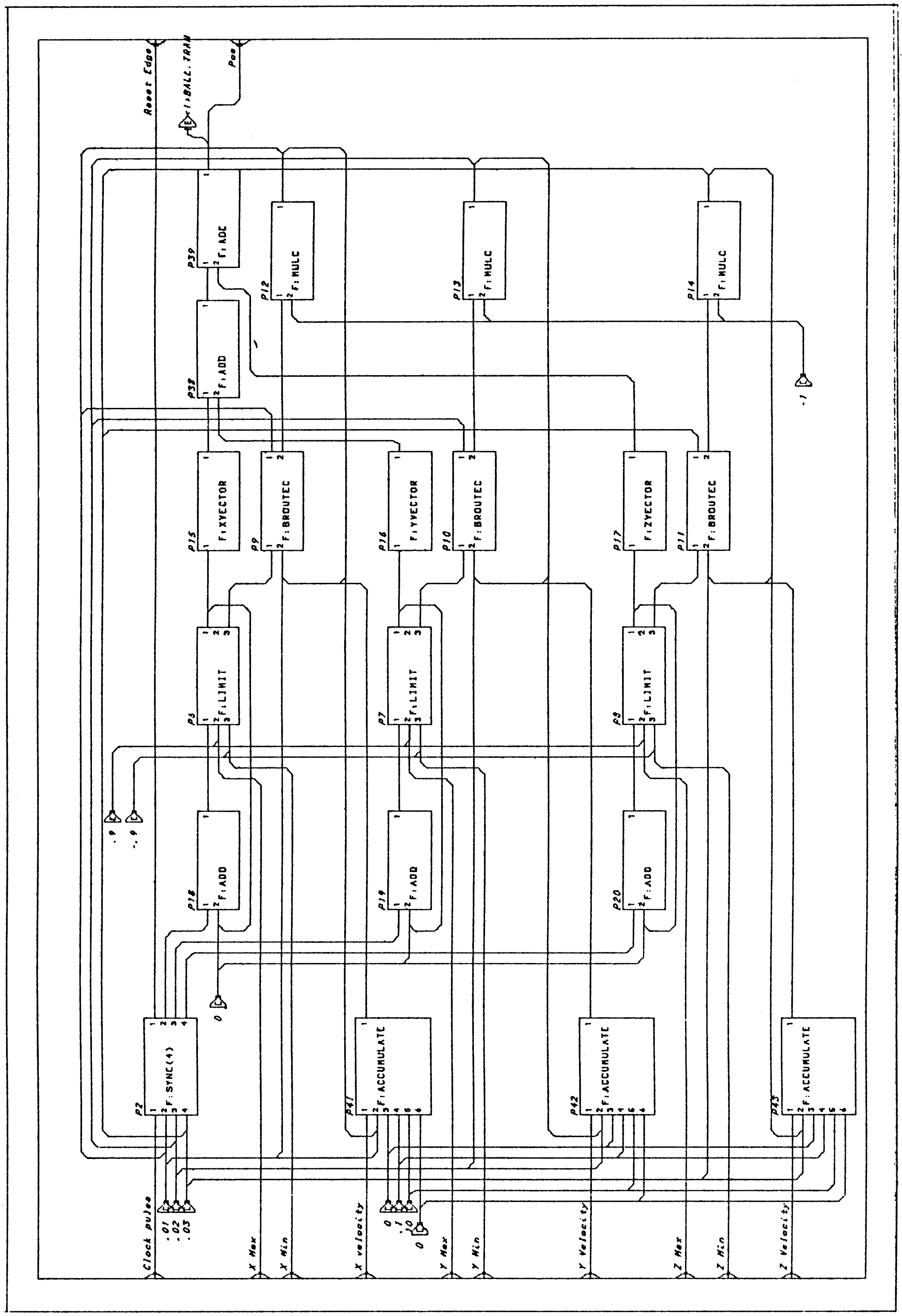


Figure 2. COLLISION.FUN (Sheet 2 of 6)

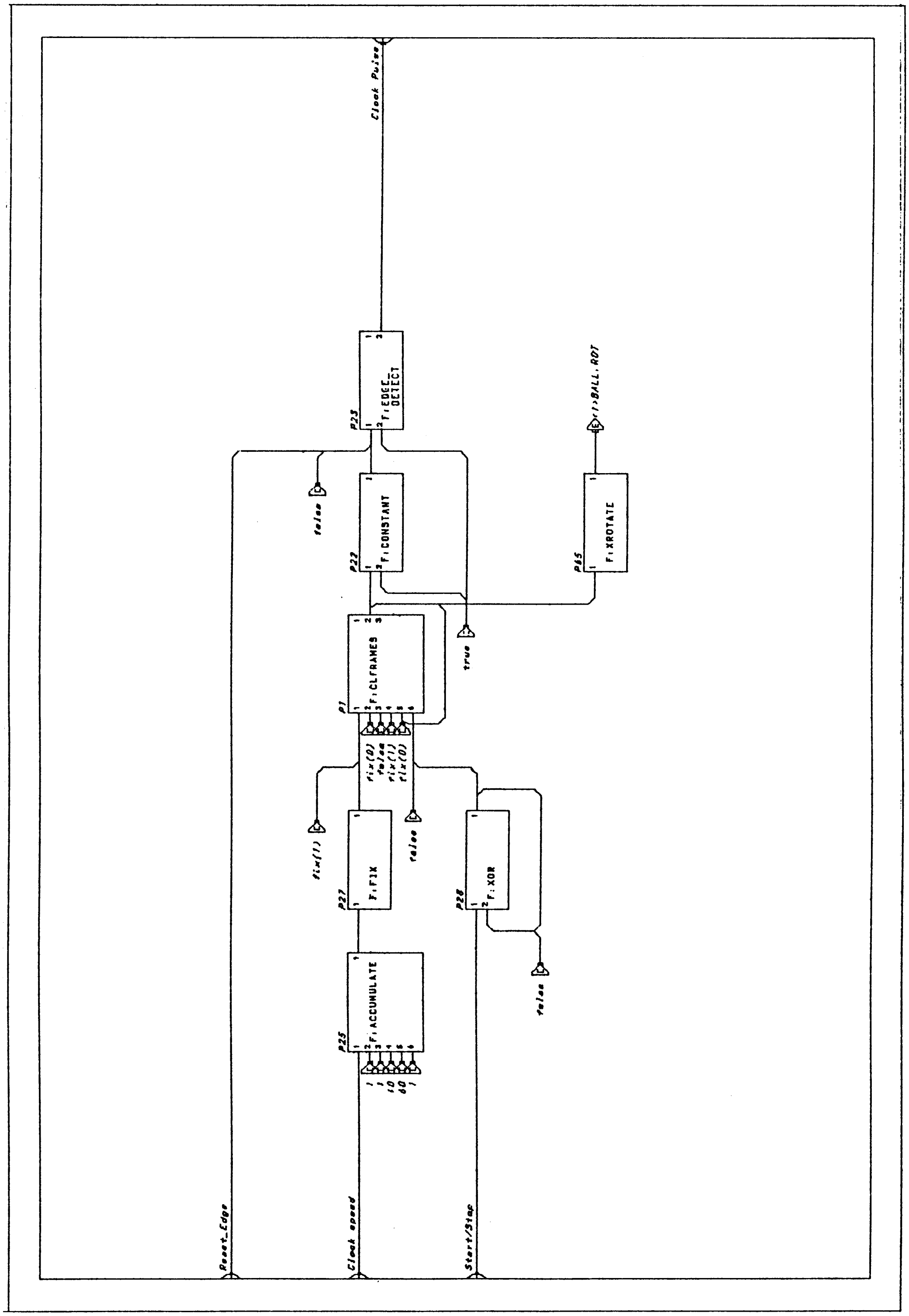


Figure 2. COLLISION.FUN (Sheet 3 of 6)

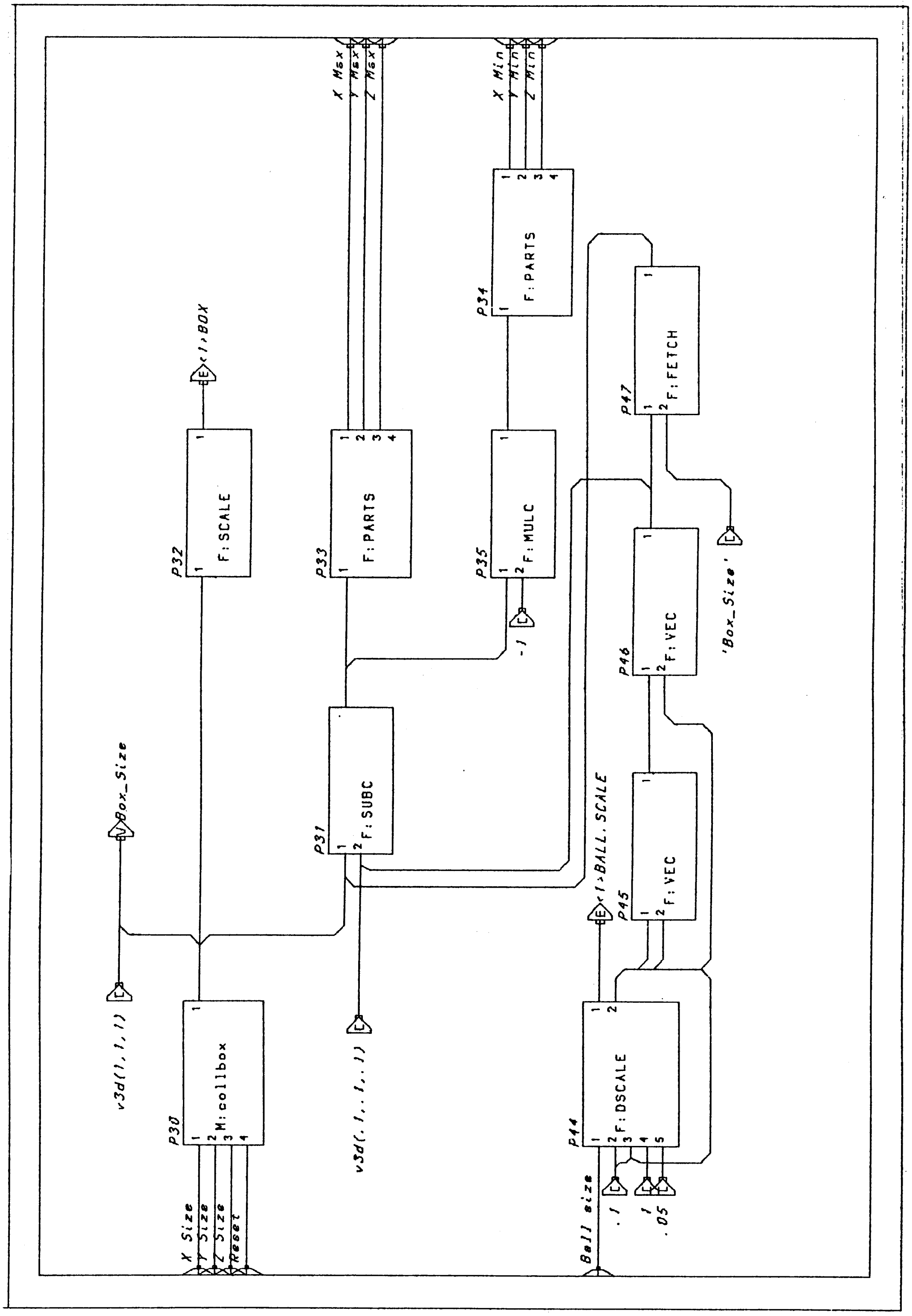


Figure 2. COLLISION.FUN (Sheet 4 of 6)

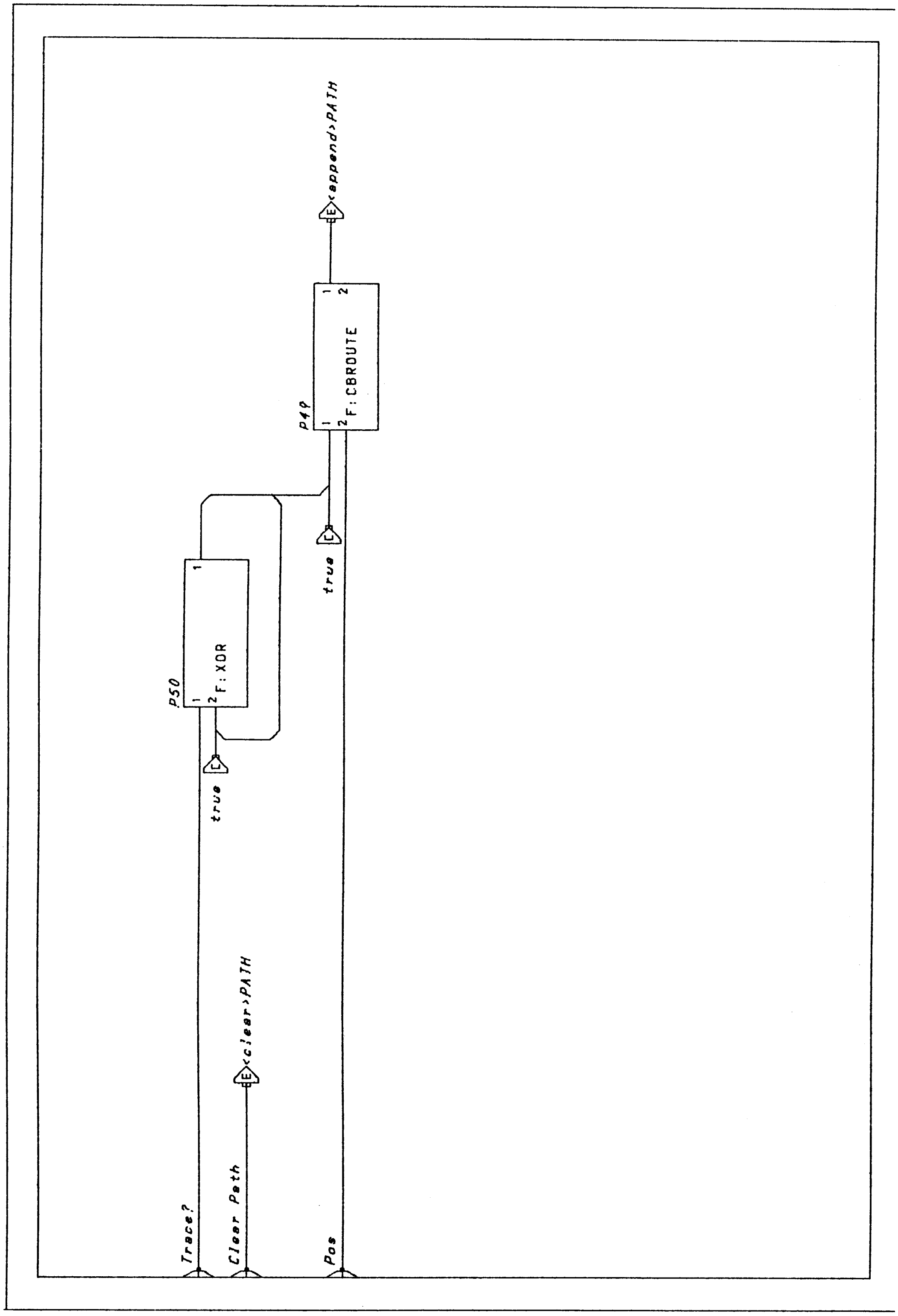


Figure 2. COLLISION.FUN (Sheet 5 of 6)

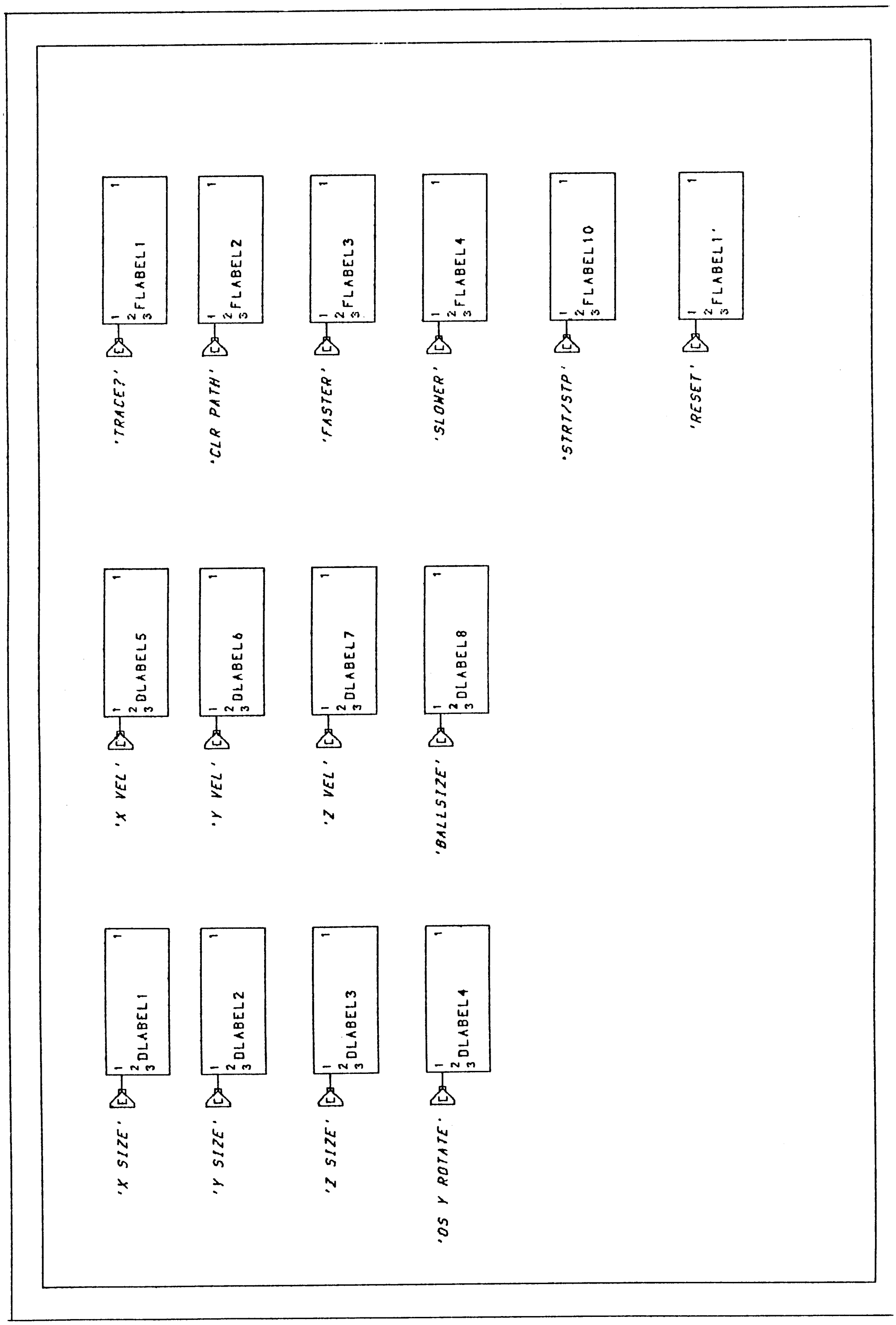


Figure 2. COLLISION.FUN (Sheet 6 of 6)

PROJECTN.300

Programmed by: Neil Jon Harrington
Software Support Specialist
Evans & Sutherland
P.O. Box 8700
Salt Lake City, Utah 84108

Created: July, 1982
Last update: February, 1985

Demonstrate X,Y, and Z planar projections using Matrix_3x3 command. The vector list data node for SPHERE, which is referred to in this structure, is not included in this file.

```
INIT DISP;
DISP Projection;

Projection := BEGIN_S
                CHAR SCALE .65;
                FONT Complex_Roman;
                INST Isometric_View;
                WINDOW x=-7.2:7.2 y=-7.2:7.2;
                INST Front_View,Side_View,Top_View;
              END_S;

Front_View := BEGIN_S
                VIEWPORT HOR=-1:0 VERT=-1:0;
                LOOK AT 3,2,0 FROM 3,2,-12 THEN Object;
              END_S;

Side_View := BEGIN_S
               VIEWPORT HOR=0:1 VERT=-1:0;
               LOOK AT 0,2,3 FROM 12,2,3 THEN Object;
             END_S;

Top_View := BEGIN_S
              VIEWPORT HOR=-1:0 VERT=0:1;
              LOOK AT 3,0,1 FROM 3,12,1 THEN Object;
            END_S;
```

```
Isometric_View  := BEGIN_S
                     VIEWPORT HOR=0:1 VERT=0:1;
                     WINDOW x=-7:9 y=-7:9;
Rot :=               ROT 0;
                     ROT X -30;
                     ROT Y 40 THEN Object;
                   END_S;

Object := BEGIN_S
            SET COLOR 240,1;
            SCALE 8 THEN WS_Gnomon;
            SET COLOR 0,0;
            INST Globe,Xplane,Yplane,Zplane;
          END_S;

Globe := BEGIN_S
Rot :=     ROT 0;
           SCALE 1.5;
           SET COLOR 0,1 THEN Sphere;
           SET COLOR 120,1;
           SCALE 1.5 THEN Os_Gnomon;
         END_S;

Xplane := BEGIN_S
            TRAN 5,0,0;
            INST Xprojection_Matrix;
            ROT Y -90;
            INST Square;
            LABELS -2.5,-2.5 'YZ Plane';
          END_S;

Yplane := BEGIN_S
            TRAN 0,5,0;
            INST Yprojection_Matrix;
            ROT X 90;
            INST Square;
            LABELS -2.5,-2.5 'XZ Plane';
          END_S;

Zplane := BEGIN_S
            TRAN 0,0,-5;
            INST Zprojection_Matrix,Square;
            LABELS -2.5,-2.5 'XY Plane';
          END_S;

XProjection_Matrix := MATRIX_3X3  0,0,0
                                  0,1,0
                                  0,0,1 THEN Globe;
```

```
YProjection_Matrix := MATRIX_3X3   1,0,0
                                   0,0,0
                                   0,0,1 THEN Globe;

ZProjection_Matrix := MATRIX_3X3   1,0,0
                                   0,1,0
                                   0,0,0 THEN Globe;

Square := VEC n=5 3,3 -3,3 -3,-3 3,-3 3,3;

WS_Gnomon := BEGIN_S
               TEXT SIZE .05;
               SET CHARACTERS Screen_Oriented;
               FONT Triplex_Roman;
               LABELS
                 1.1,-.05 'Wx'
                 -.05,1.1 'Wy'
                 -.05,-.05,1.1 'Wz';
               VEC ITEM n=5 P 0,.8,0 L 0,0,0 L .8,0,0
                 P 0,0,0 L 0,0,.8;
               TRAN .8,0 THEN Xarrow;
               TRAN 0,.8 THEN Arrow;
               TRAN 0,0,.8 THEN Zarrow;
             END_S;

Xarrow := ROT z -90 THEN Arrow;
Arrow := SCALE .025,.2,.025 THEN Pyramid;
Zarrow := ROT x 90 THEN Arrow;

OS_Gnomon := BEGIN_S
               CHARACTER SCALE .0375;
               SET CHARACTERS Screen_Oriented;
               FONT Triplex_Roman;
               LABELS
                 1.1,-.05 'Ox'
                 -.05,1.1 'Oy'
                 -.05,-.05,1.1 'Oz';
               WITH PATTERN 1 1 LEN .1
               VEC ITEM n=5 P 0,.8,0 L 0,0,0 L .8,0,0
                 P 0,0,0 L 0,0,.8;
               TRAN .8,0 THEN Xarrow;
               TRAN 0,.8 THEN Arrow;
               TRAN 0,0,.8 THEN Zarrow;
             END_S;

Pyramid := VEC BLOCK ITEM n=10
   P 1,0, 1 L -1,0,1 L -1,0,-1 L 1,0,-1 L 1,0,1 L 0,1,0 L 1,0,-1
   P -1,0,-1 L  0,1,0 L -1,0,1;
```

PROJECTN.FUN

Programmed by: Neil Jon Harrington
Software Support Specialist
Evans & Sutherland
P.O. Box 8700
Salt Lake City, Utah 84108

Created: July, 1982
Last update: February, 1985

Demonstrate X,Y, and Z planar projections using Matrix_3x3 command. The vector list data node for SPHERE, which is referred to in this structure, is not included in this file.

```
{ Code generated by Network Editor 1.07 }
{ PROJECTN }
{ Frame-Prefix Macro-Prefix  }
{ Frame1:M2$F1_ }
M2$F1_P1:=F:MULC;
M2$F1_P2:=F:MULC;
M2$F1_P3:=F:MULC;
M2$F1_P4:=F:XROTATE;
M2$F1_P5:=F:YROTATE;
M2$F1_P6:=F:ZROTATE;
CONN M2$F1_P1<1>:<1>M2$F1_P4;
CONN M2$F1_P2<1>:<1>M2$F1_P5;
CONN M2$F1_P3<1>:<1>M2$F1_P6;
SEND 200 TO <2>M2$F1_P2;
SEND 200 TO <2>M2$F1_P1;
SEND 200 TO <2>M2$F1_P3;
{ Frame1:M1$F1_ }
{World Space Rotations}
M1$F1_P2:=F:CMUL;
M1$F1_P3:=F:CONSTANT;
CONN M2$F1_P5<1>:<2>M1$F1_P2;
CONN M2$F1_P4<1>:<2>M1$F1_P2;
CONN M2$F1_P6<1>:<2>M1$F1_P2;
CONN M1$F1_P2<1>:<1>M1$F1_P2;
CONN M1$F1_P3<1>:<1>M1$F1_P2;
SEND M3D(1,0,0 0,1,0 0,0,1) TO <2>M1$F1_P3;
SEND M3D(1,0,0 0,1,0 0,0,1) TO <1>M1$F1_P2;
```

```
{ Frame1:M3$F1_ }
M3$F1_P1:=F:INPUTS_CHOOSE(13);
M3$F1_P2:=F:ROUTE(12);
CONN M3$F1_P1<1>:<2>M3$F1_P2;
SEND TRUE TO <1>M3$F1_P1;
SEND TRUE TO <2>M3$F1_P1;
SEND TRUE TO <3>M3$F1_P1;
SEND TRUE TO <4>M3$F1_P1;
SEND TRUE TO <5>M3$F1_P1;
SEND TRUE TO <6>M3$F1_P1;
SEND TRUE TO <7>M3$F1_P1;
SEND TRUE TO <8>M3$F1_P1;
SEND TRUE TO <9>M3$F1_P1;
SEND TRUE TO <10>M3$F1_P1;
SEND TRUE TO <11>M3$F1_P1;
SEND TRUE TO <12>M3$F1_P1;
{ Labels:F2_ }
SEND 'RESET' TO <1>FLABEL11;
SEND 'OS ROT' TO <1>FLABEL2;
SEND 'WS ROT' TO <1>FLABEL1;
SEND 'OBJ ZROT' TO <1>DLABEL7;
SEND 'OBJ YROT' TO <1>DLABEL6;
SEND 'OBJ XROT' TO <1>DLABEL5;
SEND 'VIEWZROT' TO <1>DLABEL3;
SEND 'VIEWYROT' TO <1>DLABEL2;
SEND 'VIEWXROT' TO <1>DLABEL1;
{ Frame1:F1_ }
F1_P2:=F:CROUTE(2);
F1_P3:=F:MULC;
F1_P4:=F:MULC;
F1_P5:=F:MULC;
F1_P6:=F:XROTATE;
F1_P7:=F:YROTATE;
F1_P8:=F:ZROTATE;
F1_P9:=F:CMUL;
F1_P10:=F:MULC;
F1_P14:=F:CONSTANT;
CONN M1$F1_P2<1>:<1>Isometric_View.Rot;
CONN M1$F1_P3<1>:<1>Isometric_View.Rot;
CONN F1_P2<1>:<2>F1_P9;
CONN F1_P2<2>:<1>F1_P10;
CONN F1_P3<1>:<1>F1_P6;
CONN F1_P4<1>:<1>F1_P7;
CONN F1_P5<1>:<1>F1_P8;
CONN F1_P6<1>:<2>F1_P2;
CONN F1_P7<1>:<2>F1_P2;
CONN F1_P8<1>:<2>F1_P2;
CONN F1_P9<1>:<1>Globe.Rot;
CONN F1_P9<1>:<2>F1_P10;
CONN F1_P9<1>:<1>F1_P9;
CONN F1_P10<1>:<1>F1_P9;
CONN F1_P10<1>:<1>Globe.Rot;
CONN F1_P10<1>:<2>F1_P10;
CONN M3$F1_P2<1>:<1>F1_P2;
CONN M3$F1_P2<2>:<1>F1_P2;
CONN M3$F1_P2<11>:<1>F1_P14;
CONN M3$F1_P2<11>:<1>M1$F1_P3;
CONN FKEYS<1>:<13>M3$F1_P1;
CONN FKEYS<1>:<1>M3$F1_P2;
CONN DIALS<1>:<1>M2$F1_P1;
CONN DIALS<2>:<1>M2$F1_P2;
CONN DIALS<3>:<1>M2$F1_P3;
CONN DIALS<5>:<1>F1_P3;
CONN DIALS<6>:<1>F1_P4;
CONN DIALS<7>:<1>F1_P5;
CONN F1_P14<1>:<2>F1_P10;
CONN F1_P14<1>:<1>F1_P9;
CONN F1_P14<1>:<1>Globe.Rot;
SEND FIX(2) TO <2>M3$F1_P1;
SEND FIX(1) TO <1>M3$F1_P1;
SEND FIX(1) TO <13>M3$F1_P1;
SEND FIX(1) TO <1>M3$F1_P2;
SEND M3D(1,0,0 0,1,0 0,0,1) TO <2>F1_P14;
SEND M3D(1,0,0 0,1,0 0,0,1) TO <1>F1_P9;
SEND M3D(1,0,0 0,1,0 0,0,1) TO <2>F1_P10;
SEND 200 TO <2>F1_P3;
SEND 200 TO <2>F1_P4;
SEND 200 TO <2>F1_P5;
```

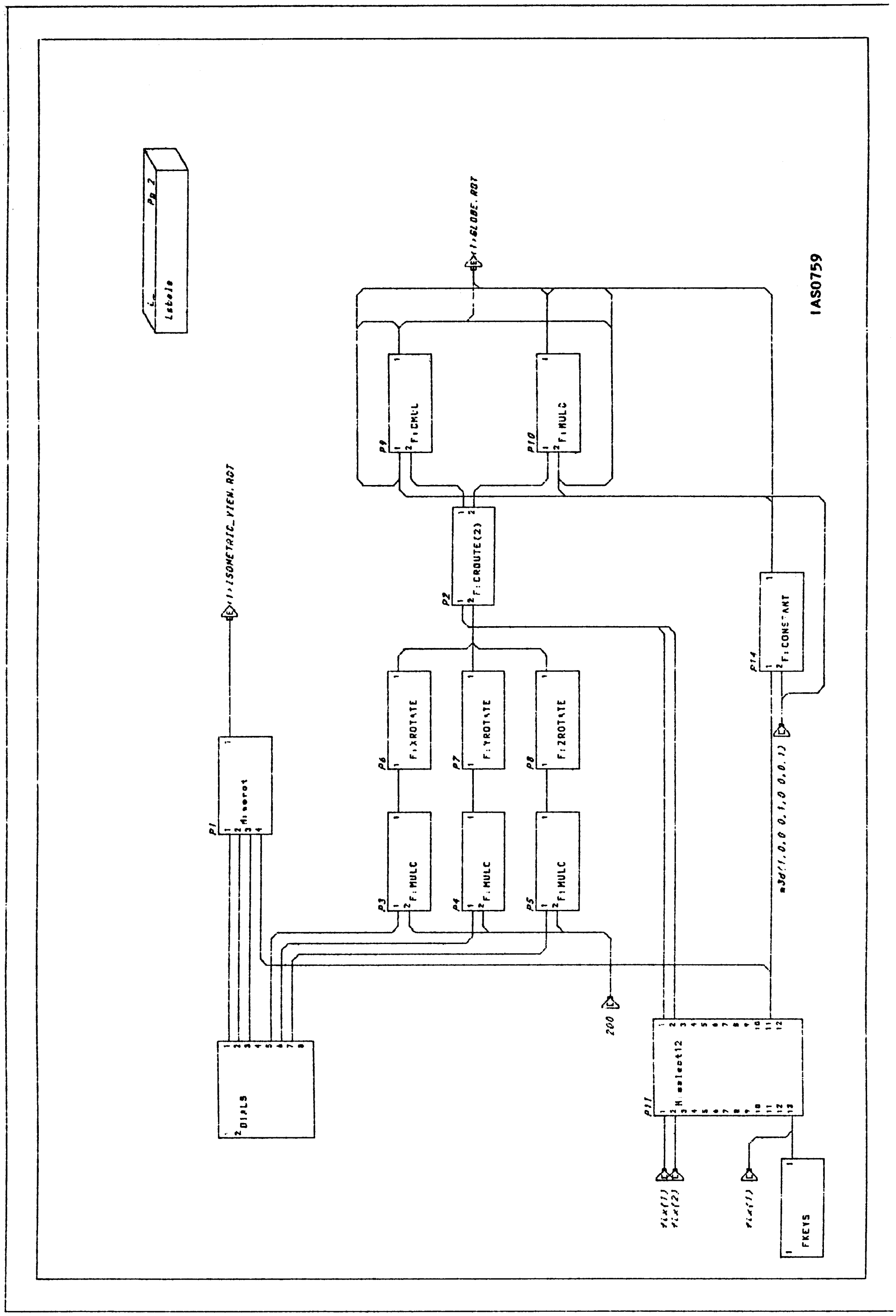


Figure 3. PROJECTN.FUN (Sheet 1 of 2)
(Function Network for PROJECTN.300)

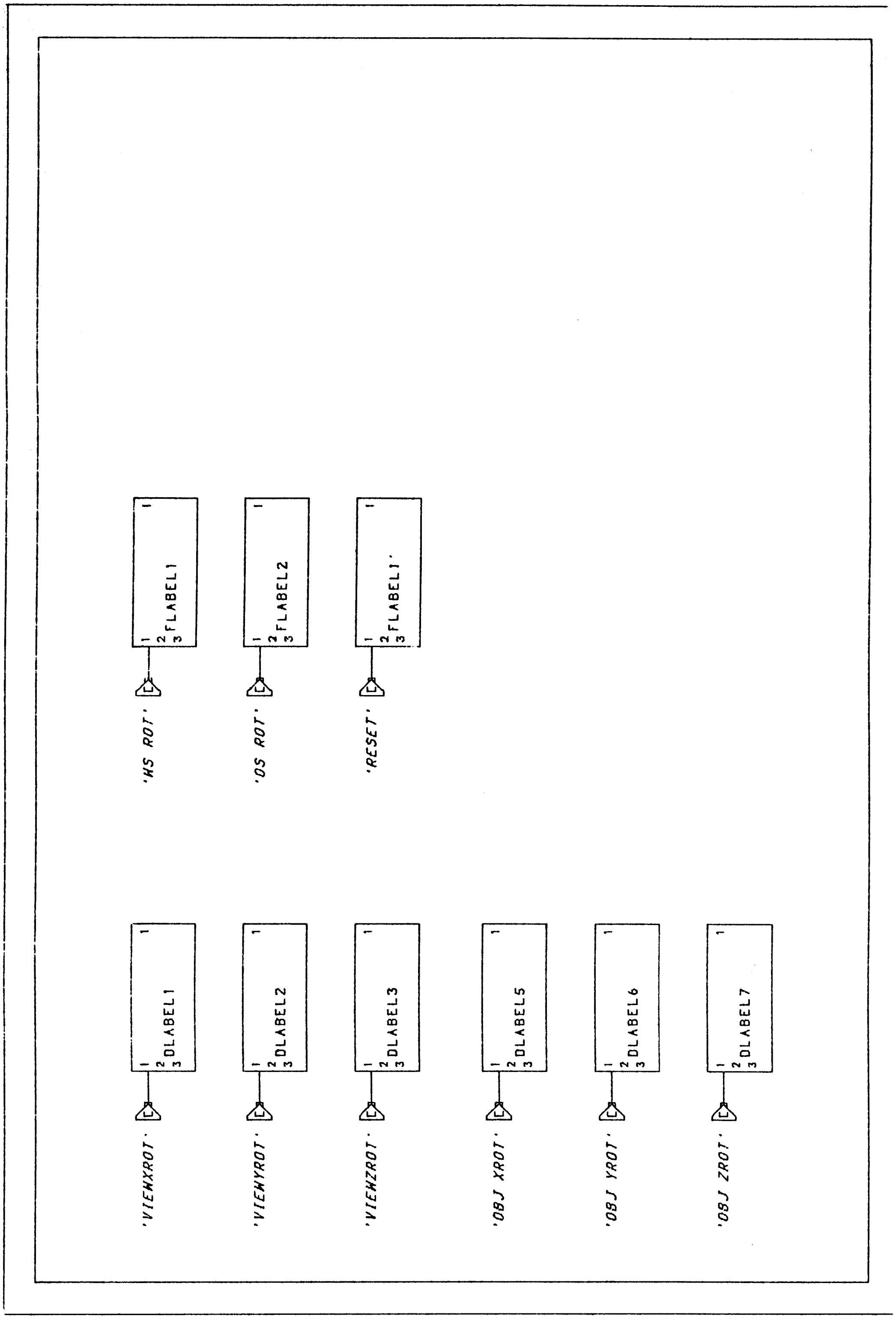


Figure 3. PROJECTN.FUN (Sheet 2 of 2)

TRISQUARE.300

Programmed by: Neil Jon Harrington
Software Support Specialist
Evans & Sutherland
P.O. Box 8700
Salt Lake City, Utah 84108

Created: December, 1983
Last update: February, 1985

Demonstration to transform four pieces from an equilateral triangle to a square and vice versa. Can be done either manually with the dials or automatically by starting a clock. The control network is in TriSquare.fun.

```
INIT DISP;
DISP TriSquare;

TriSquare := BEGIN_S
               WINDOW X=-5:5 Y=-5:5;
               TRAN -2,2;
Rot :=         ROT 0;
               SET COLOR 0,1 THEN Part1;
               TRAN 1,-1.268;
P2_Rot :=      ROT 0;
               SET COLOR 90,1 THEN Part2;
               TRAN -1,-1.732;
P3_Rot :=      ROT 0;
               SET COLOR 180,1 THEN Part3;
               TRAN -1.5,.866;
P4_Rot :=      ROT 0;
               SET COLOR 240,1 THEN Part4;
             END_S;

PART1 :=   VEC n=5 0,.4641 -.5,-.4019 -.2857,-1.5151 1,-1.268 0,.4641;

PART2 :=   VEC n=5 0,0 -1.2857,-.2471 -1,-1.732 1,-1.732 0,0;

PART3 :=   VEC n=5 0,0 -.2142,1.1135 -1.5,.866 -2,0 0,0;

PART4 :=   VEC n=4 0,0 1.2858,.2475 1,1.732 0,0;
```

TRISQUARE.FUN

Programmed by: Neil Jon Harrington
Software Support Specialist
Evans & Sutherland
P.O. Box 8700
Salt Lake City, Utah 84108

Created: December, 1983
Last update: February, 1985

Network to control the structure created by TriSquare.300.

```
{ Code generated by Network Editor 1.07 }
{ TRISQUARE }
{ Frame-Prefix Macro-Prefix  }
{ Clock Motion:F2_ }
{ first in que ---> }
F2_P13:=F:EQC;
F2_P14:=F:XOR;
F2_P15:=F:CLFRAMES;
F2_P16:=F:BROUTEC;
F2_P17:=F:SYNC(2);
CONN FKEYS<1>:<1>F2_P13;
CONN F2_P13<1>:<1>F2_P14;
CONN F2_P14<1>:<2>F2_P14;
CONN F2_P14<1>:<6>F2_P15;
CONN F2_P15<2>:<5>F2_P15;
CONN F2_P15<3>:<1>F2_P16;
CONN F2_P16<2>:<2>F2_P15;
CONN F2_P16<2>:<1>F2_P17;
CONN F2_P17<2>:<4>F2_P15;
CONN F2_P17<2>:<2>F2_P17;
SEND FIX(1) TO <2>F2_P13;
SEND FALSE TO <2>F2_P14;
SEND FIX(-1) TO <2>F2_P17;
SEND FIX(1) TO <2>F2_P17;
SEND FIX(179) TO <2>F2_P16;
SEND FIX(0) TO <5>F2_P15;
SEND FIX(1) TO <4>F2_P15;
SEND FALSE TO <3>F2_P15;
SEND FALSE TO <6>F2_P15;
SEND FIX(179) TO <2>F2_P15;
SEND FIX(6) TO <1>F2_P15;
```

```
{ Labels:F3_ }
SEND 'STRT/STP' TO <1>FLABEL1;
SEND 'JOINT 3' TO <1>DLABEL3;
SEND 'JOINT 2' TO <1>DLABEL2;
SEND 'JOINT 1' TO <1>DLABEL1;
{ Frame1:F1_ }
F1_P1:=F:ACCUMULATE;
F1_P2:=F:ACCUMULATE;
F1_P3:=F:ACCUMULATE;
F1_P4:=F:ZROTATE;
F1_P5:=F:ZROTATE;
F1_P6:=F:ZROTATE;
CONN F1_P1<1>:<1>F1_P4;
CONN F1_P2<1>:<1>F1_P5;
CONN F1_P3<1>:<1>F1_P6;
CONN F1_P4<1>:<1>Trisquare.P2_Rot;
CONN F1_P5<1>:<1>Trisquare.P3_Rot;
CONN F1_P6<1>:<1>Trisquare.P4_Rot;
CONN DIALS<1>:<1>F1_P1;
CONN DIALS<2>:<1>F1_P2;
CONN DIALS<3>:<1>F1_P3;
CONN F2_P15<2>:<1>F1_P4;
CONN F2_P15<2>:<1>F1_P5;
CONN F2_P15<2>:<1>F1_P6;
SEND 180 TO <5>F1_P1;
SEND 180 TO <5>F1_P2;
SEND 180 TO <5>F1_P3;
SEND 200 TO <4>F1_P1;
SEND 200 TO <4>F1_P2;
SEND 200 TO <4>F1_P3;
SEND 0 TO <2>F1_P1;
SEND 0 TO <3>F1_P1;
SEND 0 TO <6>F1_P1;
SEND 0 TO <2>F1_P2;
SEND 0 TO <3>F1_P2;
SEND 0 TO <6>F1_P2;
SEND 0 TO <2>F1_P3;
SEND 0 TO <3>F1_P3;
SEND 0 TO <6>F1_P3;
```

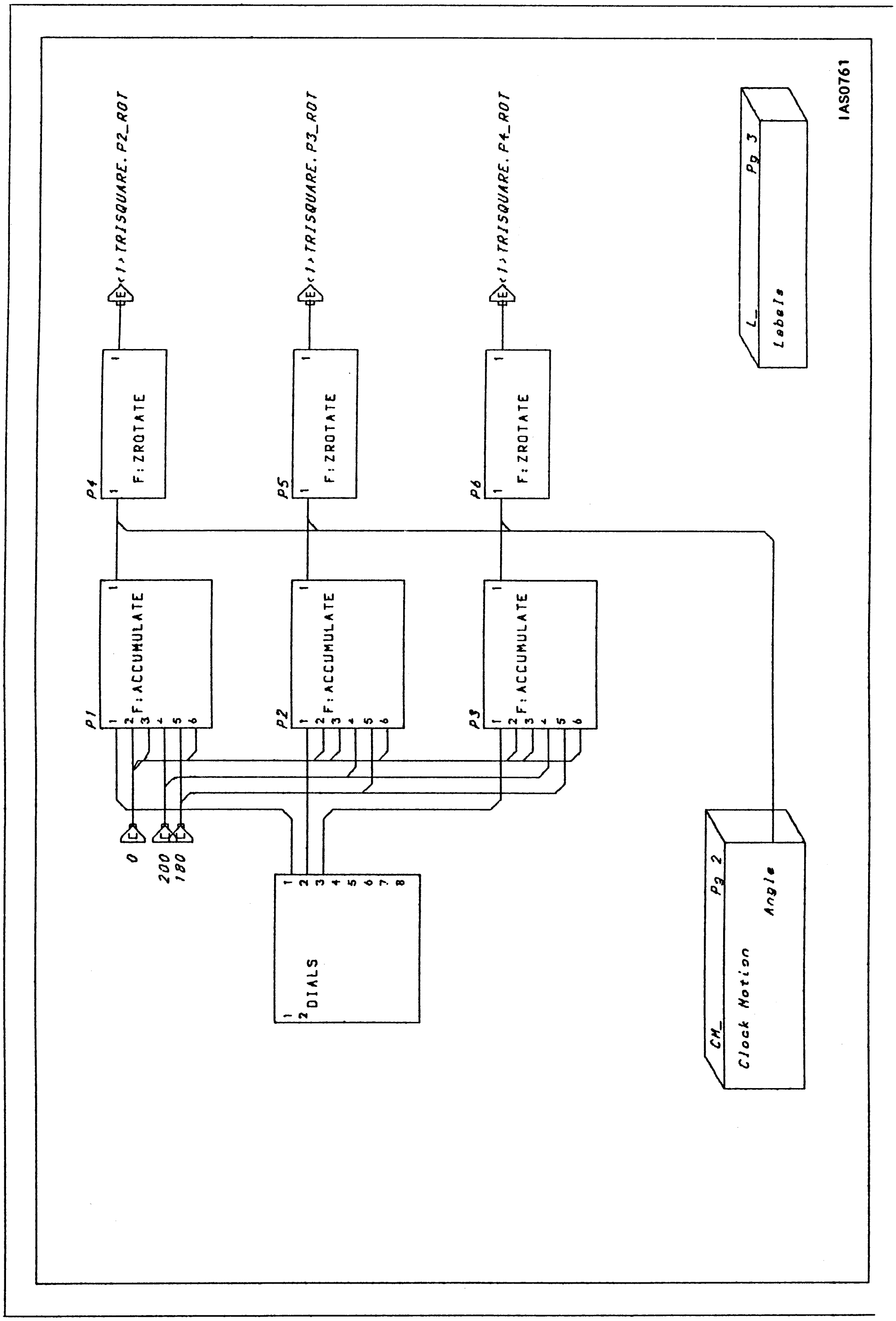


Figure 4. TRISQUARE.FUN (Sheet 1 of 3)
(Function Network for TRISQUARE.300)

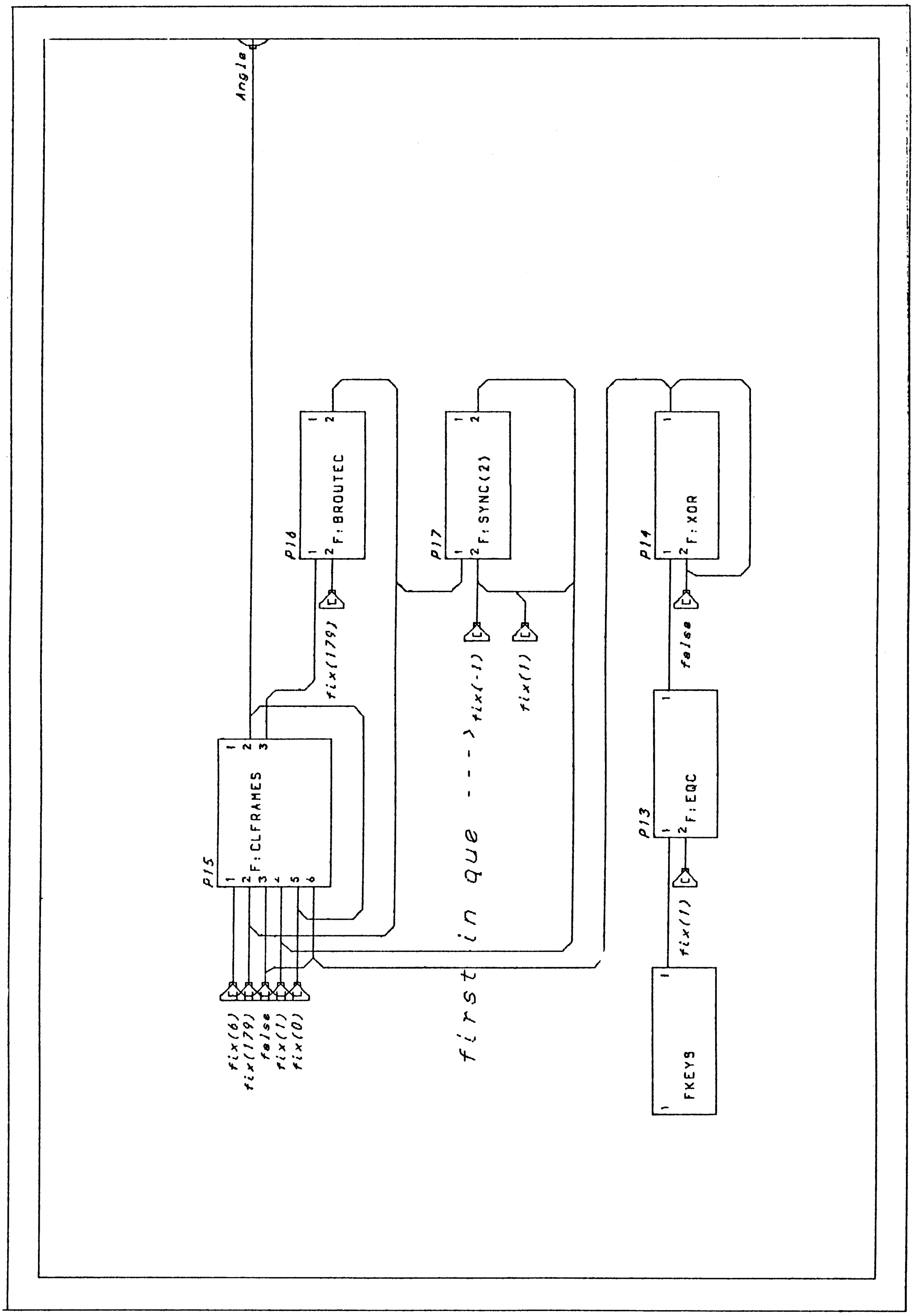


Figure 4. TRISQUARE.FUN (Sheet 2 of 3)

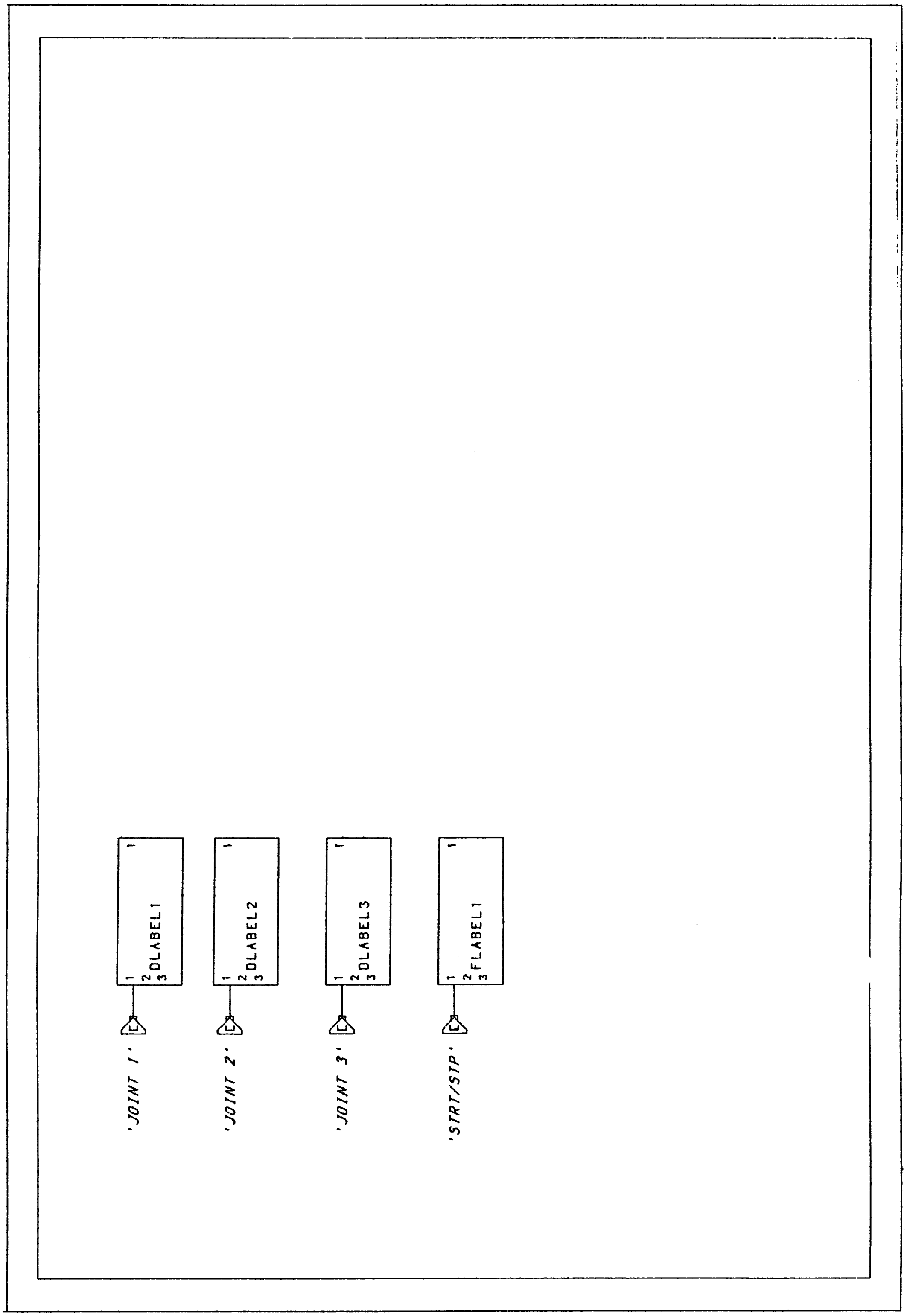


Figure 4. TRISQUARE.FUN (Sheet 3 of 3)

PROGRAM SetRate (Input,Output);

Programmed by: Neil Jon Harrington
Software Support Specialist
Evans & Sutherland
P.O. Box 8700
Salt Lake City, Utah 84108

Created: November, 1984
Last update: February, 1985

PS 300 Set Rate programming example using the GSRs. Pascal version of BLKLEVF.FOR created originally by A. Kerry Evans. To run this program compile it and link it with the Pascal GSR library.

This program creates a PS 300 display structure that has many SET RATE nodes cascaded by offsetting the starting time of each SET RATE node. This structure is another way to create an animated sequence on the PS 300. A function network is not needed, since the PHASE attribute value is modified by the DISPLAY PROCESSOR as a function of the number of times a SET RATE node is traversed.

```
CONST
   DTheta  = 0.1745329;  { PI/18 }
   %INCLUDE 'gsrlib:ProConst.pas/nolist'

TYPE
   %INCLUDE 'gsrlib:ProTypes.pas/nolist'

VAR
   Theta   : REAL;
   Tran    : P_VectorType;
   I       : INTEGER;
   Vecs    : P_VectorListType;
   Front   : P_VectorListType;
   Name    : P_VaryingType;

%INCLUDE 'gsrlib:ProExtrn.pas/nolist'


PROCEDURE ErrHnd ( Error : INTEGER);
   BEGIN
      WRITELN  ( 'Error: ',Error:3);
   END; { ErrHnd }
```

```
PROCEDURE Calc_Wave;
   VAR
       I,J  : INTEGER;
       VecNum  : INTEGER;
       VecNum2   : INTEGER;

   BEGIN
       VecNum  := -1;
       FOR I := 0 TO 49 DO BEGIN
           VecNum  := VecNum + 2;
           VecNum2  := VecNum + 1;

           Vecs[VecNum].v4[1] := I/50;
           Vecs[VecNum].v4[2] := 0.8*EXP(-0.02*I)*COS(Theta-0.2513274123*I);
           Vecs[VecNum].v4[3] := 0;
           Vecs[VecNum].v4[4] := 1 - I/150;

           Vecs[VecNum2].v4[1] := Vecs[VecNum].v4[1];
           Vecs[VecNum2].v4[2] := 0;
           Vecs[VecNum2].v4[3] := 0.5;
           Vecs[VecNum2].v4[4] := Vecs[VecNum].v4[4];

           FOR J := 1 TO 4 DO
               Front[I+1].v4[J] := Vecs[VecNum].v4[J];

       END; { FOR I }
   END; { Calc_Wave }

BEGIN
   PAttach  ('LogDevNam=tt:/PhyDevTyp=async',ErrHnd);
   PInitC  (ErrHnd);
   PInitD  (ErrHnd);
   Tran.v4[1]  := -0.5;
   Tran.v4[2]  := 0;
   Tran.v4[3]  := 0;
   PTransBy  ('Sine_Wave',Tran,'Inst',ErrHnd);
   PInst  ('Inst','',ErrHnd);
   Theta  := -DTheta;
   FOR I := 10 TO 46 DO BEGIN
       Theta  := Theta + DTheta;
       WriteV  (Name,'VecList',I:2);
       WRITELN  (Name);
       Calc_Wave;
       PBeginS  (Name, ErrHnd);
           PSetR ('',1,35,FALSE,I,'',ErrHnd);
           PIfPhase ('',TRUE,'',ErrHnd);
           PVecBegn ('',100,TRUE,FALSE,3,P_Sepa,ErrHnd);
           PVecList (100,Vecs,ErrHnd);
           PVecEnd (ErrHnd);
           PVecBegn ('',50,TRUE,FALSE,3,P_Conn,ErrHnd);
           PVecList (50,Front,ErrHnd);
           PVecEnd (ErrHnd);
       PEndS  (ErrHnd);
       PIncl  (Name,'Inst',ErrHnd);
   END; { FOR I }

   PDisplay  ('Sine_Wave',ErrHnd);
   PDetach  (ErrHnd);
END. { SetRate }
```

GLOSSARY OF TERMS

This glossary contains definitions of terms that might be unfamiliar to a novice graphics programmer. It also contains terms that are specific to the PS 300 Graphics Systems.

Active Input -- An active input (or active queue) is one of two types of function input queues. Data arrive at an active queue and are input to a function on a first-in-first-out basis as soon as the function is activated. The data on active queues are consumed by the function.

Arithmetic Control Processor (ACP) -- This is a subsystem in the PS 300's Display Processor. The ACP includes a microprocessor that performs arithmetic and logical functions on data in Mass Memory. The ACP traverses display trees, performs matrix multiplication (rotation, scaling, and windowing), applies the matrix to the data nodes, and outputs the transformed coordinates of the data to the Pipeline Subsystem.

Aspect Ratio -- This ratio is the relationship of height to width. The aspect ratios of windows and viewports must be the same, or objects will appear distorted on the screen.

Attribute (Attribute node) -- An attribute is a characteristic that can be applied to data in a manner similar to a transformation. Attributes, unlike transformations, do not affect the location, orientation, size, or vector definitions associated with an object. They are non-matrix operations. They set and change characteristics of the displayed image, such as depth clipping, enabling picking and blinkir j, intensity, or the color of an object. An attribute command creates an operation node. This node changes a bit setting in the ACP save-state block, rather than affecting the contents of the current transformation matrix. Attribute nodes normally accept a Boolean value and/or integers.

B-spline -- A B-spline is a mathematical representation of a curve which approximates and interpolates a specified set of points.

Backface Removal -- Backface removal is an intermediate step in hidden-line removal. It removes all polygons facing away from the viewer.

Block-normalized Vectors -- When the components of all coordinate locations (vectors) that comprise an object have a common exponent, they are said to be block-normalized vectors.

Break Key -- A break key is used to send a break sequence to the host system. The break key for PS 300/host communications (using the VT-100 Terminal Emulator) is defined by the user in the SETUP mode of the keyboard.

Calligraphic -- Calligraphic is a term used to describe a method of displaying dots, characters, and lines on a CRT screen. It is also referred to as random stroke. In contrast to the raster display of a typical TV screen, the electron gun traces the displayed lines from endpoint to endpoint and does not scan each screen location every refresh cycle.

Character Font -- A character font is a set of characters of a particular style. A standard 128 ASCII character set is provided with the PS 300 graphics system as the Standard Font. Different fonts can be used instead of, or as a supplement to, the standard font, by using the BEGIN_FONT ... END_FONT and CHARACTER FONT commands.

Characters Node -- A characters node is a data node that consists of a single string of up to 240 characters. These nodes are created by the CHARACTERS command.

Clipping -- Clipping is a viewing operation that removes from the screen display lines or parts of lines that are outside of the viewing area (window). If lines were not clipped, they would wrap back onto the display.

Command Interpreter (CI) -- The PS 300 command interpreter is a system function that is responsible for accepting a stream of tokens (messages) until it has enough to act on (a complete command).

Command Mode -- The Command mode is also referred to as CI mode or local command mode and is one of three types of communication modes available on any style PS 300 keyboard. Command mode implies that data entered locally (as opposed to data received from the host) are to be routed to the command interpreter. Command mode displays the "@@" prompt on the screen.

Communication Modes -- There are three communication modes available with the PS 300 terminal emulator; Terminal Emulator mode (TE), Command mode (CI), and Interactive or Graphics mode (KB).

Compounded Renderings -- A rendered object that has had another rendering operation applied to it is said to be a compounded rendering.

Conditional Referencing -- Conditional referencing is the referencing of data only when certain conditions are met. Conditional referencing is set up in a display tree by creating SET nodes which set any of fourteen conditional bits. IF nodes are placed further down in the display tree to test for the condition set above. Data is referenced if the condition is met.

Constant Input -- A constant input queue is an input of a function where only the last message entered in the queue is input to the function. A message in a constant input queue will be used and reused until another message is queued on that input to replace it.

Contrast -- Contrast is the difference in intensity from brightest to dimmest (near to far) of the lines that compose an image. Contrast is used to impart the illusion of depth in an object.

Control Dials -- The Control Dials Unit is one of the PS 300's interactive devices. It consists of a bank of eight dials which can be programmed to control the orientation of objects displayed on the screen. The dials send out numbers which are used as input to function networks. The numbers are converted through the network to matrices which update translation, rotation, and scale nodes in a display tree.

Control Key -- The Control Key (CTRL) on the PS 300 keyboard generates a control sequence and is used in conjunction with other keys. Control Key sequences are used in PS 300/host ANSI control and escape sequences. The character '↑' is used to represent the control value of a key (the sequence that is generated when the Control Key is pressed and a second key is pressed).

Coordinate System -- All mathematical information that the designer enters to create an object must be given in terms of a three-dimensional coordinate system. A coordinate system is a way of specifying a three-dimensional space in which objects can be modeled. The coordinate system used in programming the PS 300 is a left-handed Cartesian coordinate system, usually referred to as the "world coordinate system."

Coplanar -- For polygons, coplanar denotes that polygons have the same plane equation. This equation is used in the PS 340 system to associate inner and outer contours.

Cross Sectioning -- Used in programming the PS 340, this rendering operation makes use of a defined sectioning plane to create a cross section of an object. When this operation is used, both sides of the object are discarded and only the slice defined by the sectioning plane remains.

Current Transformation Matrix (CTM) -- When a series of transformations are applied to graphical data, the matrices are concatenated. This means that each matrix is pre-multiplied to a matrix called the current transformation matrix. The current transformation matrix contains the accumulation of all transformations that are to be applied to graphical data and preserves the order in which they are to be applied.

Data Node -- Data nodes are terminal nodes for the branches of a display tree. Data nodes can contain vector lists, polygons, curves, and text. They are represented as squares in a display tree.

Data Structure -- See **Display tree.**

Data Structuring Commands -- These commands build or affect display trees. They create structures that are stored in Mass Memory for later execution by the Display Processor. Data structuring commands are not "saved" as files on the PS 300.

Data Tablet -- The data tablet is one of the interactive devices, commonly used with the PS 300. It is a flat board used for inputting data, for pointing, and for picking items from a graphics display. A data tablet typically provides two-dimensional positioning data and is used in conjunction with a stylus or a puck.

Demultiplexing -- Demultiplexing is a communication operation where one input accepts data from many sources, all of which have the same destination. A demultiplexing operation can be performed by a port or a function.

Depth Clipping -- Depth clipping is a viewing operation that clips from the display objects or parts of objects which extend beyond the Z (or depth) plane of a viewing area. This is also known as Z-clipping.

Depth Cueing -- This operation imparts an illusion of depth to the image of a three-dimensional object by decreasing the intensity of lines as they "recede" into the distance (i.e., in positive Z).

Diffuse -- In shading polygons, this attribute is used to specify the proportion of color contributed by diffuse reflection versus that contributed by specular reflection.

Display -- As a verb, display refers to the visibility of a graphical object on the graphics terminal screen. As a noun, display is used to refer to the terminal screen, (i.e., PS 300 Display). "Display" is also used as a modifier; display tree, display structure, etc.

Display Data Structure -- See **Display tree.**

Display List -- The display list contains the names of all display trees that are currently being traversed for display. Whenever anything is displayed, its name goes on the display list.

Display Processor (DP) -- The Display Processor traverses display trees in Mass Memory and processes the data for display. The Display Processor transforms the data to be displayed; performs clipping, perspective projection, and viewport mapping.

Display Tree -- A display tree represents the structure of an object in mass memory. Display trees are a hierarchical ordering of instance nodes, operation nodes, and data nodes. They contain primitives and the transformations and attributes which are applied to them. The Display Processor traverses the display tree of any object in the display list.

Distributed Graphics -- Distributed graphics allows the graphical portion of an application to be performed by the graphics system, without burdening the host computer.

Exposure -- Exposure is a shading parameter that controls the overall brightness of an object displayed on the PS 340 raster display.

Field-of-view Angle -- The field-of-view angle is the angle at the apex of the viewing pyramid used to define a perspective viewing area. This angle is used as a parameter of the FIELD_OF_VIEW command.

Flat Shading -- Flat shading is applied to objects on a PS 340 raster display. This process uses color, one light source, and depth cueing to shade the polygons in the object accordingly. Flat shading can produce objects that simulate a faceted surface.

Frustum -- This is a section of a viewing pyramid that is obtained by slicing through the pyramid parallel to the base. The frustum encloses a portion of the world coordinate system. Any objects contained in the frustum will be displayed in perspective projection on the screen.

Function -- A function is a procedure that performs one or more operations by accepting input, processing input, and sending output. The PS 300 has three types of functions: intrinsic functions, initial function instances, and user-written functions.

Function Instance -- A function instance is a specific case of an intrinsic function that is created and named by the user when the need for a particular function arises. A function instance has a set of input sources and output destinations, specified by the user. Function instances are combined into function networks.

Function Keys -- The PS 300 Function keys are the top row of keys on any style PS 300 Keyboard. These keys are number F1 through F12. The character generated by any function key is dependent on the mode of the keyboard. Function keys are generally used to provide inputs to function networks. They are also used in the SETUP mode of the keyboard.

Function Network -- A function network is a collection of interconnected function instances. Function networks are the programmed path between an interactive device and an interaction node in a display tree, or between the PS 300 and the host. One end of a function network is usually connected to a node in a display tree and the other end to the port associated with an interactive device or the host computer.

Geometry -- The geometry of an object is the location in the coordinate system of the points that define the object.

Graph Key -- The GRAPH key is a toggle key on the left-hand keypad of any style PS 300 keyboard. Pressing the GRAPH key displays or blanks pictures on the PS 300 screen.

Graphics Command Language -- A graphics command language is the user-interface to a graphics system. It is a high-level set of commands and instructions specifically designed for graphical operations.

Graphics Control Processor (GCP) -- The GCP serves as the central controller for the PS 300. It provides the interfaces to devices external to the system and manages all internal system communications. The GCP also controls the display trees in Mass Memory and initiates the display defined by these structures.

Graphics Support Routines (GSRs) -- The GSRs are a collection of Pascal routines or FORTRAN calls that pre-parse and package data on the host. There is a GSR routine that corresponds to almost every PS 300 command.

Hardcopy Key -- The HARDCOPY key on the left-hand keypad of any style PS 300 keyboard is used to activate the hardcopy plotter. If no plotter is attached to the system and the key is pressed, an error message is generated.

Hidden Line -- In a line-drawing model, a hidden line is one which would be obscured by surfaces of the model.

Hidden-line Removal -- This PS 340 rendering operation generates a view in which only the unobstructed portions of an object are displayed.

Hierarchical Data Structure -- A hierarchical data structure is a structure that is arranged in such a way that a hierarchical order is maintained between what precedes and what follows any element in the structure. Complex models are designed as a hierarchical structure called a display tree.

Hierarchy -- A hierarchy is a principled organization of components. The organizing principle will vary depending on the relationship between components which the hierarchy is designed to show.

I/O Subroutines -- The PS 300 I/O Host-Resident Subroutines (PSIOs) are callable FORTRAN subroutines designed to aid in host/PS 300 communications.

Identity Matrix -- An identity matrix is composed of ones and zeros, with the ones running in a diagonal (top left to bottom right). Multiplying by an identity matrix is the equivalent of multiplying by one: nothing changes. The current transformation matrix (CTM) starts out as an identity matrix each time a display tree is traversed.

Illumination -- This attribute is used with the PS 340 raster system to specify light sources applied to a shaded object.

Immediate Action Commands -- These commands cause an immediate result when received by the system. Immediate action commands perform such actions as initializing, naming, and displaying data.

Initial Function Instance -- An initial function instance is a function that is automatically instanced for use upon system initialization. Such a function may be system-connected into an initial function network or user-connected to a user function network. Examples of initial function instances are DIALS, TABLETIN, PICK, HOST MESSAGE. Unlike intrinsic functions, initial function instances are not preceded by "F:" and are not named by the user.

Initial Structure -- Initial structures are structures which are loaded into memory with the PS 300 firmware. When the PS 300 is initialized, two initial structures are loaded. CURSOR defines the cursor as an 'X', and PICK_LOCATION identifies the pick-sensitive area as the center of the cursor.

Inner Contour -- The inner contour of a polygon represents a cavity, hole, or protrusion site in an object.

Instance -- An instance is a specific reference to any nameable entity. There is an INSTANCE command which creates instance nodes in the display tree. Intrinsic functions must be "instanced" (uniquely named) before they can be used in a function network using the NAME:= function_name; command.

Instance Nodes -- Instance nodes group primitives, transformations, and attributes into single-named entities. They are represented as triangles in a display tree.

Interactive Devices -- Interactive devices (also referred to as peripheral devices) provide programmable interactive capabilities that allow an operator to interact with graphical data on the screen. These devices include, but are not restricted to, the control dials, a data tablet, the keyboard, and function buttons.

Intrinsic Functions -- Intrinsic functions are the set of nameable functions that are provided for constructing function networks. These functions have the 'F:' prefix and must be instanced (i.e., given a unique user-defined name) before they can be used in a function network.

Labels -- Labels are data nodes that consist of a block of several character strings.

Left-hand Rule -- The left-hand rule is a mnemonic for the direction of rotation around an axis in the PS 300's world coordinate system. Point the thumb of your left hand in the positive direction of any axis, and your fingers will curl in the direction of positive rotation.

Level-of-detail -- Level-of-detail settings are attributes that allow data to be conditionally referenced for display based on a level-of-detail setting. A SET node is created in a branch of a display tree to set the level-of-detail to a certain value. An IF node created lower down in the structure tests for the condition set above and displays data only if the condition is met.

Line Generator System (LGS) -- The LGS is the final subsystem in the Display Processor. After data have been processed in the Arithmetic Control Processor and the Pipeline Subsystem, the LGS converts the X,Y, and intensity information into analog signals that drive deflection and intensification circuitry in the PS 300 Display.

Line Local Key -- The LINE LOCAL key (located on the left-hand keypad) is used in conjunction with other keys to access the keyboard communication modes on the VT-100 style PS 300 keyboard.

Local Action -- Local action is the cumulative result of the operations of functions in a function network. Generally, a local action determines how an interactive device affects an image on the screen.

Look At/From -- The terms "look at" and "look from" in viewing operations are used to establish a line of sight in the world coordinate system. The PS 300 uses the line of sight to perform a matrix operation which transforms the coordinates of an object to produce the correct "view" on the screen. All points in the world coordinate system are translated and rotated to place the "from" point at the world coordinate system origin and the "at" point on the positive Z axis.

Mass Memory -- Mass Memory is memory in which display structures are stored and managed by the Graphics Control Processor. These structures are accessed by the Display Processor through a dedicated port. Mass Memory also stores function instances, function connections, and function messages.

Memory Alert -- There is a memory alert display area at the bottom of the PS 300 Screen. This is connected to a system function that alerts a user by displaying the amount of existing memory whenever available memory drops below an acceptable level.

Message -- A message is data input to and output from function instances.

Message Data Type -- A message data type is the specific data type associated with a message to or from a function instance. Message data types include: Boolean, character, character string, integer, real, 2D vector, 3D vector, 4D vector, 2D, 3D, 4D position vector, 2D, 3D, 4D line vector, 2x2 matrix, 3x3 matrix, 4x4 matrix, and pick list.

Modeling -- Modeling is the process of defining graphical primitives and applying modeling transformations to them. These transformed and untransformed primitives are used as parts of complex models.

Modeling Transformations -- These transformations move primitives to a new location in the coordinate system or deform primitives to create new shapes. There are three modeling transformations: rotation, translation, and scaling.

Multiplexing -- A multiplexing operation can be performed by a port or a function. Multiplexing is a communication operation where one input accepts data from a single source and distributes the data to various destinations.

Naming -- Naming is the process by which the user identifies a particular command or group of commands. Once a name has been given, all data specified by the command(s) are referenced for use by referring to the assigned name. The name of entity is equivalent to its "address" in memory.

Non-uniform Scaling -- Non-uniform scaling consists of scaling an object by different amounts in different dimensions.

Normal -- Normals are used with shaded renderings and are given with each vertex of the polygon specified N X,Y,Z. The shaded-rendering process interpolates between these normals when rendering the polygon to generate a smooth shaded image.

Null Object -- A null object is created when a name is referenced that has not previously been defined.

Object Space Rotation -- An object is said to rotate in object space when it rotates around its own set of axes.

Operation Nodes -- Transformations and attributes are represented by operation nodes in a display tree. They are represented as circles. Operation nodes can be used as points of interaction with a model. They can receive new values from interactive devices such as dials or the data tablet. Operation nodes which are set up as interaction points are shown as double circles in a display tree.

Orthographic Projection -- Orthographic projection is the two-dimensional projection of a three-dimensional object in which lines that are parallel in the object always appear parallel, without regard to relative distance from the eye. This form of projection is also called parallel projection.

Outer Contour -- The outer contour of a polygon represents a face of an object.

Parallel Projection -- See orthographic projection.

Perspective Projection -- Perspective projection is a viewing projection that allows spatial relations (distance and position) of three-dimensional objects to be represented as they might appear to the eye. Parallel lines in the object appear to converge with respect to relative distance or depth from the eye position.

Pick Identifier -- A pick identifier (pick ID) is a user-assigned name that allows data to be reported as "picked".

Pick List -- A pick list is the information returned when a pick occurs. A pick list consists of an integer that represents the element of a vector list or a character in a string that was picked, and a list of pick identifiers. A pick list can optionally contain the picked coordinate location.

Picking -- Picking is the ability to indicate with a pointing device such as a stylus or light pen a displayed object or a portion of an object which is oriented in any manner. When some part of the displayed image is picked, the PS 300 generates a pick list which identifies the element picked.

Picking Location -- The picking location is a region within a viewport. If a pick-sensitive entity (line, character, or dot) is within the pick location, it may be reported as having been picked. When the PS 300 is initialized, the pick location is established by the Initial Structure PICK_LOCATION as being the center of the displayed cursor.

Pipeline Subsystem (PLS) -- The PLS is a subsystem in the Display Processor. The PLS accepts transformed coordinate data from the Arithmetic Control Processor and performs clipping, perspective division, and viewport mapping on the data to be displayed. The processed data are then output to the Line Generator Subsystem in the form of X,Y locations and intensity values.

Pixel -- A pixel is a picture element. It is the smallest element which can be displayed on a raster display device.

Polygon -- A polygon is a closed-plane figure defined by the coordinates of its vertices. The edges of the polygon are defined by lines that connect those vertices. In the PS 340, a polygon must have at least three vertices and no more than 250, all of which must lie in the same plane.

Polygon Clause -- This part of the POLYGON command defines an individual polygon or face of an object by specifying the coordinates of its vertices.

Polygonal Definition -- The polygonal definition of an object specifies the association of multiple points as parts of separate polygons.

Primitive -- This is the simplest object in a graphical data base. It consists entirely of points and lines or planes. The points specify the geometry of the primitive, the lines or planes specify the topology.

Quality Level -- Quality level is a shading parameter used with the PS 340. It controls the number of pixels over which filtering is applied.

Raster -- Raster is one technique used for producing an image on a CRT screen. Raster images are generated with an intensity controlled, line-by-line sweep across the screen, in contrast to calligraphic displays that trace only the displayed lines, dots, or characters.

Raster System -- The raster system is an option for the PS 340 system that consists of a printed circuit card which outputs static images to a pixel raster display. The raster system can be used as an "image buffer" to display host-generated images or it can display "shaded images" derived locally from PS 340 polygonal models.

Real Time -- The term "real time" is applied to high-performance computer graphics systems which allow the operator to interact with a displayed object with no perceptible delay. For example, if a car is displayed on the screen and a dial rotates the car around the vertical axis, real time interaction gives the illusion that the car actually rotates with no perceptible delay as the dial is turned.

Refresh Buffer -- The refresh buffer is a memory buffer that temporarily stores graphical data which has been processed for display.

Rendering Node -- A rendering node is an operation node created with the PS 340 SOLID_RENDERING or SURFACE_RENDERING command.

Rendering Operations -- Rendering operations are performed with the PS 340 on polygonal objects to remove hidden line segments, perform backface removal, section an object relative to a sectioning plane, obtain a cross section, or display shaded objects on a raster screen.

Right-hand Rule -- The right-hand rule for polygons states that if you point the thumb of your right hand towards the center of a polygonally defined object and rotate your fingers towards your wrist, the direction that your fingers move indicates the order in which the vertices of that polygon should be listed in the polygon clause.

Rotate -- An object that is rotated around any of the three axes (X, Y, Z) in world coordinate space is said to rotate "in" that axis. When an object is rotated in X, for example, X values do not change; Y and Z do.

Rotation Angle -- The rotation angle is the angle (degrees) of rotation around a particular axis.

Rotation Matrix -- A rotation matrix is a 3X3 matrix used to perform a rotation on an object. The PS 300 uses the sine and cosine of the angle of rotation to create the matrix, then applies the matrix to the coordinates of the points which define the object.

Scale -- To scale an object is to apply a factor to any or all dimensions of an object. Scaling may or may not be proportional in all dimensions (X, Y, and Z). If the scale is applied in all dimensions, this is uniform scaling. A different scale factor applied in different dimensions is known as non-uniform scaling.

Scaling Matrix -- The PS 300 creates a 3X3 scaling matrix which multiplies the coordinates of the points which define the object by the scale factor. This determines the new coordinates of the scaled object.

Screen-oriented Characters -- Screen-oriented characters are not affected by ROTATE and SCALE nodes that are applied to the object of which they are a part. Screen-oriented characters maintain their size and their front-facing orientation when other data is transformed.

Sectioning -- Sectioning is a PS 340 rendering operation which cuts away parts of polygons that extend beyond an arbitrarily positioned plane called the sectioning plane. This plane passes through the object to divide the object into two pieces. When sectioning is performed, the affected polygons are reconstructed so that they do not extend beyond the sectioning plane; one piece is removed while the other remains displayed.

Shaded Renderings -- Shaded renderings are PS 340 operations which are used on the raster screen to draw the surface of a polygonally defined object. Shaded operations include wash shading, flat shading, and smooth shading.

Shading -- Shading is the process of drawing the surface of a polygonally defined object. "Flat" shading uniformly fills the interior of the polygons so that each polygon is recognizable. "Smooth" shading fills the polygons in a non-uniform manner to give the appearance of curvature to the object surface.

Smooth Shading -- Smooth shading is a rendering operation applied to objects on a PS 340 raster display. The color of a polygon is varied across its surface, affected by the normals at the polygon's vertices, the direction and color of various active light sources, the polygon's attributes (both color and highlights), and depth cueing. Objects that simulate a curved surface can be produced with smooth shading.

Soft Edges -- An edge declared with the "S" specifier in the polygon clause of the POLYGON command is a soft edge. Soft edges are invisible in hidden-line renderings except when they make up part of the profile of an object or a silhouette.

Solid -- A solid is a polygonal object that encloses a volume of space. In a solid, every edge of every polygon must coincide with an edge of a neighboring polygon.

Solid Fill -- Solid fill is the shading of the interior of polygons. This, along with hidden line removal, makes an object appear solid.

Solid Model -- Solid models are representations of physical objects within a computer so that not only computer-generated pictures, but also physical characteristics, such as center of gravity and moments of inertia, can be generated from the computer model.

Specular -- This PS 340 polygon attribute is used in shading polygons to adjust the concentration of specular highlights.

Sphere of Influence -- Sphere of influence defines the influence one node has on other nodes in the display tree. In general, any node in a hierarchical branch has influence over nodes below it in the same branch. Spheres of influence are established and maintained by instance nodes.

String -- A string is a sequence of characters and spaces enclosed in single quotation marks. Strings can be displayed as text or can be used as inputs to function instances.

Structuring -- Structuring using the BEGIN_STRUCTURE ... END_STRUCTURE; command allows commands that must otherwise be named to be grouped without naming the individual commands. It provides a method of applying transformations without using the explicit APPLIED TO form of the command.

Surface -- A surface is a polygonal object that does not enclose a volume of space.

Term Key -- The TERM key is a toggle key on the left-hand keypad of any style PS 300 keyboard. Pressing the TERM key displays or blanks terminal emulator text (characters received from the host or local communication functions) on the PS 300 screen.

Terminal Emulator -- The terminal emulator is a feature available over standard interface lines that allows the PS 300 to be used as a host terminal. With the RS-232, RS-449, DMR-11AE, DEC Parallel, and other ASCII interfaces, the PS 300 emulates a DEC VT-100 terminal. With the IBM 3278 interface, the PS 300 emulates an IBM 3278 terminal.

Toggle -- A toggle feature on a key or button means that pressing it once activates a feature or function and pressing it a second time deactivates the feature or function.

Topology -- The topology of an object is the manner in which points specified in the object's geometry are connected with lines or planes.

Transformed Data -- Transformed data is the matrix or vector-list representation of transformation operations in a display tree.

Translate -- To translate an object is to relocate it in world coordinate space. An object which is translated in X is moved in the X direction. An object translated in X and Y is moved some distance in the X direction and some distance in the Y direction.

Traverse -- This is a process in which the ACP steps through the display tree in Mass Memory to retrieve data and operation specifications necessary to generate a picture.

Uniform Scaling -- In uniform scaling, an object is scaled by the same amount in all dimensions.

Variable -- A variable is a storage place for updating values for use in function networks.

Vector -- A vector is a coordinate location that may or may not be the endpoint of a line.

Vector-normalized Data -- When the components (X, Y or X, Y, Z) of a single coordinate (vector) location share a common exponent, they are said to be vector-normalized.

Viewing Area -- This is a three-dimensional region of world coordinate space in which objects can be viewed. A viewing area in which objects are viewed orthographically is created by the WINDOW command. The EYE and FIELD_OF_VIEW commands create a viewing space for perspective projections of an object.

Viewing Pyramid -- A viewing pyramid defines a portion of world-coordinate space for viewing objects in perspective. The actual viewing area is shaped like a frustum. The pyramid is completed by extending the converging sides of the pyramid until they meet. This point, the apex of the pyramid, is the eye point of the viewer.

Viewing Transformations -- Viewing transformations are matrix operations which specify whether displayed objects appear in perspective or orthographic projection. Viewing transformations also specify a point to look from and a direction to look in the world coordinate system.

Viewport -- A viewport is the area of the PS 300 display screen in which pictures are displayed. Multiple viewports can be displayed on the same screen so that the same object may be viewed from different vantage points, different objects can be viewed, text can be displayed, etc. The viewport specification is a ratio and proportion calculation, unlike viewing transformations which are matrix operations. Viewport specifications are not concatenated with the current transformation matrix.

Viewport Mapping -- Viewport mapping is the projection of data from the world coordinate system to a viewport.

Wash Shading -- Wash shading is applied to objects on a PS 340 raster display. It produces an object with area-filled colored polygons. Wash shading ignores normals, light sources, all lighting parameters, and all depth cueing parameters.

Window -- A window is the three-dimensional area of the world coordinate system in which data is visible. A window can impose an orthographic or a perspective view on objects within it. A window is identical to a "viewing area."

Working Storage -- Working storage consists of large contiguous block of PS 340 mass memory needed to create renderings. Working storage must be explicitly reserved with the RESERVE_WORKING_STORAGE command.

World Coordinate System -- The world coordinate system is the three-dimensional space which the programmer uses for designing and modeling objects. It is a left-handed Cartesian coordinate system.

World Space Rotation -- An object is said to rotate in world space when it rotates around any of the world coordinate system axes, as opposed to one of the object's own axes.

World-oriented Characters -- Characters that are world-oriented are transformed along with an object of which they are a part.

Z-clipping -- See depth clipping.

AN INTRODUCTION TO DATA-DRIVEN PROGRAMMING METHODOLOGY

FOR PS 300 USERS

by

A.L. DAVIS

IAS #115

Evans & Sutherland Computer Corporation
P.O. Box 8700, 580 Arapeen Drive
Salt Lake City, Utah 84108

PREFACE

An original design objective of the PS 300 family was to provide a way to program a number of parallel activities that may be occurring at the same time. Our experience at Evans & Sutherland with our other products (PS1, PS2, and MPS graphics systems) showed that considerable application program complexity resulted from handling such activities as:

- polling multiple interactive devices to determine which ones have changed and apply the necessary modifications to graphical data structures.
- gathering interactive device event records, sorting through them, and applying modifications where necessary.
- performing updates from interactive device information in such a way that responsiveness to operator interaction is predictable.

Such tasks were often performed by complex programs whose results were still less than satisfactory -- responsiveness to operator interaction is hard to guarantee in a time-sharing environment. An application program in such an environment may have little or no control over the scheduling mechanism of the programs that are to be executed.

It was recognized that such needs were not unique in the computer industry. Programming languages designed to address these needs (e.g., Ada, Concurrent Pascal, Modula 2) were emerging. These languages, however, required that the application be written in one of the languages. Such a requirement would tend to limit the market for the PS 300 and was deemed unacceptable.

Instead, a mechanism was sought that would allow an application to be partitioned into an application-specific piece with no real-time response requirements and a graphics-specific piece, where real-time response is required.

The mechanism selected was one that would enable inherently parallel operations to be programmed independently and would also allow the graphics-specific portions of an application to be programmed without regard for the particular language that the application was written in. The function network facilities provided in the PS 300 thus evolved.

Recognizing that these function network facilities bore a striking resemblance to the data-flow concepts and theory that had been the subject of research for a number of years, we attempted to incorporate some data-flow terminology and concepts into the PS 300. Also, since the terminology, concepts, and data-flow programming methodology would be new to almost all PS 300 users, Dr. Alan L. Davis was contracted to write a document introducing some of the data-flow concepts, theory, and proper programming practices that might prove useful in programming the PS 300's function network facilities. This document is the result of his efforts.

It is not the intent of Evans & Sutherland to provide an entirely new programming methodology and environment for the development of application programs. We believe that this can best be done with standard programming languages, in typical program development environments. But we do feel the PS 300's data-driven methodology is well-suited for graphics-specific operations, local handling of interactive devices, and other paraliel operations. Its capability, flexibility, and responsiveness will enable sophisticated graphics programs to be written more simply and with predictable results -- regardless of the particular host computer or operating system.

It is unusual for Evans & Sutherland to distribute a document written by a single individual, as we have done with **An Introduction to Function Network Programming for PS 300 Users**. It has been a pleasure for us here at Evans & Sutherland to be able to support Al Davis in this effort. He has presented an introduction to data-driven methodology in a very readable, yet thoroughly informative manner. I hope that the reader will find his efforts as informative and useful as we have.

M.W. Mantle
Interactive Systems Engineering

ABSTRACT

The PS 300 allows users to specify a variety of interactive methods by writing function net programs. Function net programs have a rather different semantic base than conventional (or von Neumann) sequential programming languages.

Function net programming is based on data-driven semantics. The advantage of this is that the user may partition a program in a much more natural way than would be possible using a conventional programming language.

The change in the semantic base does, however, require a slightly different way of thinking and the acquisition of a new programming methodology. Fortunately, experienced programmers will find the new methodology to be simpler and less restrictive than the von Neumann methodology with which they are currently familiar.

The purpose of this document is to present a self-contained tutorial discussion of data-driven programming in general, and PS 300 function net programming in particular. The hope is that by reading and understanding this material, the PS 300 user will be able to write well-formed, stylish, function net programs which will be efficient, easy to test and debug, and easy to modify.

TABLE OF CONTENTS

TABLE OF FIGURES

1. INTRODUCTION

The purpose of this document is to provide a self-contained introduction to data-driven programming and act as a tutorial on proper programming style. As such this document will not contain the usual references to other manuals and/or papers. For the reader who wants to dive head-first into the deeper reaches of the subject, an annotated bibliography is provided which if faithfully pursued should do at least two things:

- Provide substantive material on the key issues of general net theory, distributed asynchronous programming, and data-driven programming.

- Keep the zealous reader busy for a very long time.

This document takes a rather pragmatic view in that the goal is to produce effective PS 300 programmers capable of writing efficient, clean code. Theoretical issues will therefore be well-disguised in a practical environment.

This document is organized into two major parts. The first is a general discussion of the properties of data-driven programs and proper programming style. The second part is a specific analysis of the PS 300 function net language and a discussion of proper PS 300 function net programming style.

There is some overlap in the content of the two parts. This is intentional; the acquisition of a new programming methodology for a language with an extremely unconventional semantic base is almost always confusing and frustrating. It is hoped that the reader can make this transition more smoothly by first understanding data-driven concepts in general and then seeing them applied to PS 300 function net programs.

This document assumes prior programming experience, but not necessarily prior PS 300 experience. In fact, it is hopefully the case that you are reading this before you have the "enlightening" experience of wondering what on earth should be done with your function network, which has currently gone off the wall in some unforeseen and definitely unplanned manner.

The experienced PS 300 programmer may often find the suggestions contained in this document a bit inefficient. The only justification for this is that it should be clear to most modern programmers that good style and raw efficiency of the resultant program are often not perfectly compatible. If a programmer knows all the nitty gritty details of the operating system, the compiler, and the hardware itself, then it is possible, by cheating just a wee bit, to write many wondrous programs which are fast and/or small.

Unfortunately for such people, operating systems, compilers, and hardware change as systems evolve and many "tricks" need constant updating in order to keep programs functioning in the intended manner. These changes are often substantial and, in a world where programs are created and modified by an entire community, such tricks are usually an impediment to orderly progress.

However, data-flow programming is less restrictive than strictly sequential programming. Using function net programming can result in faster and more natural programs which are easier to create, debug, modify, maintain, pass on to others, and so on.

1.1 FUNCTIONS, FUNCTION NETS, AND THE PS 300

The PS 300 system can be thought of as an interactive eye through which an object called the model is viewed. The model is created on a host machine and is represented as a database of essentially graphical information. The PS 300 eye can be used to view the model from an arbitrary position. The key feature of the PS 300 is that eye movement can be controlled by twisting knobs and dials rather than by a program running on the host machine. The result is a zoomed, scaled, or clipped view of the model.

Since users will likely want to define their own customized functions for the controls, some interface must exist to allow them to do this. The interface is in fact a type of programming language known as function nets.

A function net program or function network can be viewed as a program which takes values generated by a knob (or a set of knobs) and produces graphical transformations which modify how the model is displayed on the CRT. Usually a single knob is insufficient to specify all of the desired transformations, and several knobs need to be used. A simple example would be to use three knobs to specify the X, Y, and Z position of the eye and a fourth knob to indicate the depth of the viewed object.

It is easy to imagine numerous scenarios which would need even more controls. Sometimes two sets of knobs might be desired where each set's actions are considered to be independent from the influence of the other set. At other times complex interdependencies of individual controls are desired.

In addition, it is natural to view the controls as being used in an unordered and asynchronous manner.

The PS 300 function net language was designed to fulfill these needs in a direct and natural fashion. It is possible to support such a rich and complex variety of interactions using sequential programming languages such as FORTRAN or Pascal in conjunction with an interrupt mechanism which could supply changing control values. But this "von Neumann" approach is somewhat unnatural because the sequence specified by the sequential program does not reflect the additional control imposed by the interrupt facility.

In particular, the need for complicated interdependencies and independencies of the PS 300 controls does not map cleanly and clearly onto the sequential program structure ot a language like Pascal. This lack of clarity causes errors, delays, and in the worst case may even hide an otherwise obvious solution to a programming problem.

Fortunately, function net programs do not suffer from these problems. These programs can in fact clearly and succinctly represent arbitrary mixtures of dependence and independence from a number of PS 300 controls changing in complete asynchrony with each other.

The primary reason for this power and flexibility of function net programs is that their semantics are based on a model entirely different from languages such as Pascal. This basis is known as data-driven or data-flow semantics (the terms are used synonymously both here and in the general literature).

A more detailed discussion of data-driven programs and semantics will appear later. The essential idea is that in data-driven programs, the arrival of the operands at a function causes that function to be activated and send output values to other functions. Thus, data "drives" the computational process. Hence the name "data-driven."

In a data-driven program, there is no program counter or other central control device to indicate "what should be done next." Any number of functions may be active at any particular point in time. Thus, data-driven programs can support arbitrary amounts of concurrency or parallelism. (If the hardware can support the available concurrency, the program may execute much faster than its sequential counterparts. This is a future possibility for PS 300 successors, but the main use right now of this concurrency structure is that it permits a more natural and manageable program structure.)

A function network is presently programmed textually, but it's easier to think of it as a directed graph structure. The graph is composed of directed arcs and vertices. Each arc carries data from the "producer" vertex to the "consumer" vertex. This directed graph represents the function network that is executed by the PS 300 hardware to transform the model database into a viewable object.

The complete set of PS 300 functions can be found in the **PS 300 User's Manual**. These functions are characterized in Chapter 5 of this document. Before returning to the PS 300 function nets specifically, the next chapter describes important aspects of data-driven programming in general.

2. DATA-DRIVEN PROGRAMMING

2.1 BASIC CONCEPTS

The simplest and easiest way to understand data-driven programs is to view them as directed graphs. These graphs are composed of two basic entities: arcs and vertices. Arcs carry tokens of data in the direction indicated by the arrowhead. Arcs can be thought of as leaving from the output ports of vertices and then going to the input ports of other vertices. Vertices can be viewed as functions which consume data tokens arriving at their input ports and produce data tokens at their output ports.

The directed graph program is actually a network of these vertices and arcs, where the network topology is whatever the programmer creates. These Function Graph Networks will hereafter be referred to as FGNs.

A large variety of data-driven FGN languages and notations exist, but it is not the purpose of this document to enumerate them here. The interested reader is referred to the bibliography for pointers to the various descriptions of these schemata.

Data-driven FGN's have been created as high-level programming languages, machine languages for custom hardware, meta-languages which are interpreted by a variety of hardware architectures (the PS 300 FGN language is an example of this), or as modeling notations for processes ranging from office information systems to chemical flow in a nuclear power plant.

Figure 2-1 shows a simple FGN example that models the inventory control system at a hospital. In this example the arcs carry data tokens which correspond to forms. The process modeled in this diagram could as easily be implemented by computers or turned into a set of job descriptions performed by people. In a computerized implementation, the forms could be formatted digital messages or files.

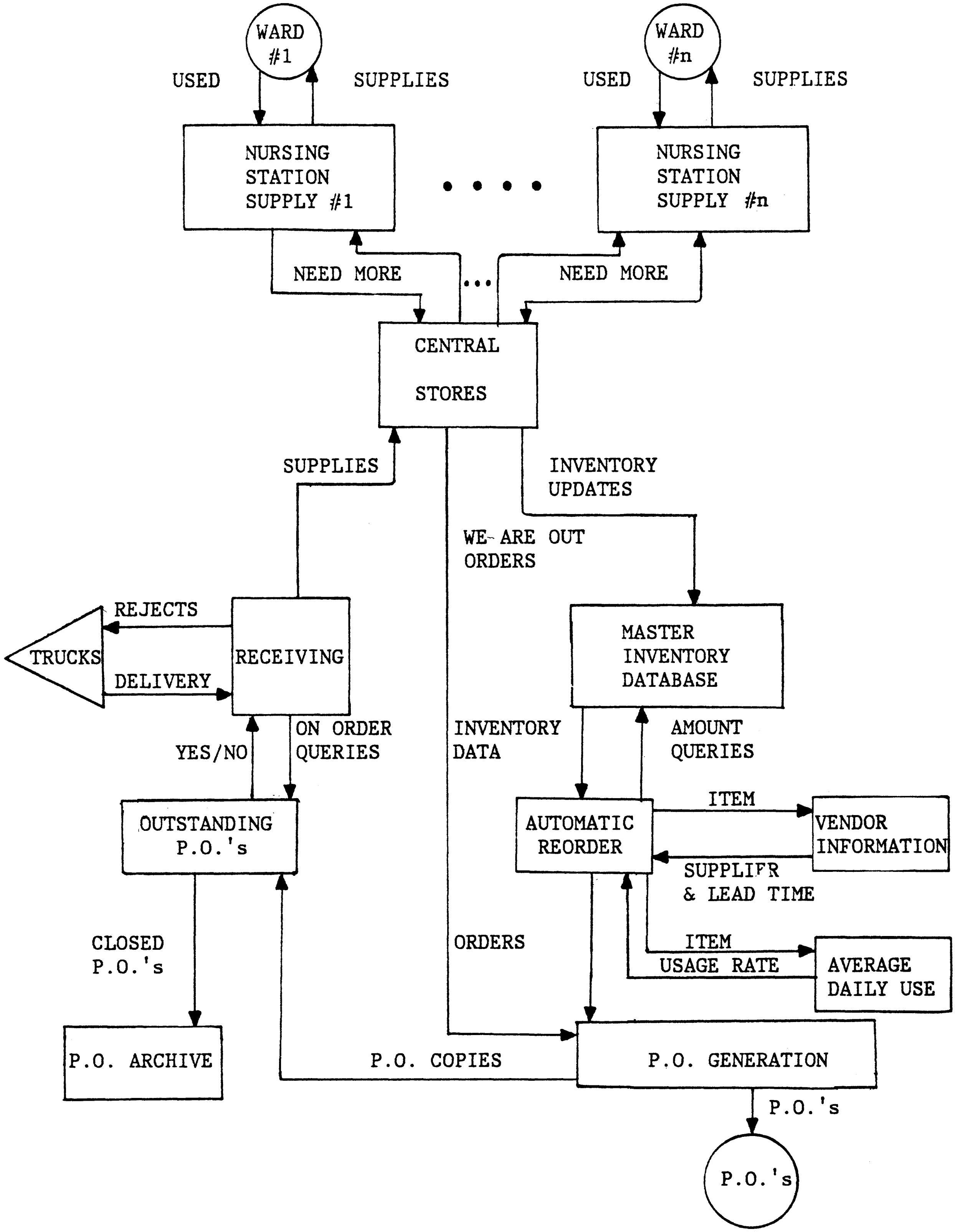


Figure 2-1. Simple Hospital Inventory FGN

Note the inherent clarity in the diagram, which is probably much clearer than a formal specification, a Pascal program, or an English language description of the process. The primary reason is that the graph is visual. It quickly and succinctly indicates the structure of the process whereas the alternative textual representations must be analyzed before the structure can be understood.

For example, in the FGN diagram of the hospital system it is easy to see that price never seems to be a factor and an automatically reordered item never gets questioned. Note also that the arc labels or inscriptions never specify function but serve more as comments do in conventional programs to describe the nature of the transmitted information.

For any particular "flavor" of FGN, there will be a set of atomic vertices corresponding to the system's functions. Vertices may or may not have special shapes which mean something. And there may be several categories of arcs. The data tokens transmitted may vary from simple numbers to multi-dimensional matrices or other highly complex structures.

How these FGN components behave determines data typing, statement typing, and so on. The choice of a particular set of vertices, arcs, tokens, and rules for their use creates a specific FGN language with its associated syntactic and semantic structure.

The syntax with which a programmer describes an FGN may be the graph itself or a textual equivalent. Choice of syntax has a major impact on clarity and user friendliness, but does not directly affect the expressive power of a particular FGN language.

In data-driven programs, the arrival of the necessary set of tokens at the input ports of a vertex causes that function to be activated. When a vertex can be activated, it is said to be fireable. When it is active, it is said to be firing and when it has completed its action, it is said to have fired. The set of inputs which are needed to make a given vertex fireable is called the firing set of that vertex.

2.2 CONJUNCTIVE AND DISJUNCTIVE VERTICES

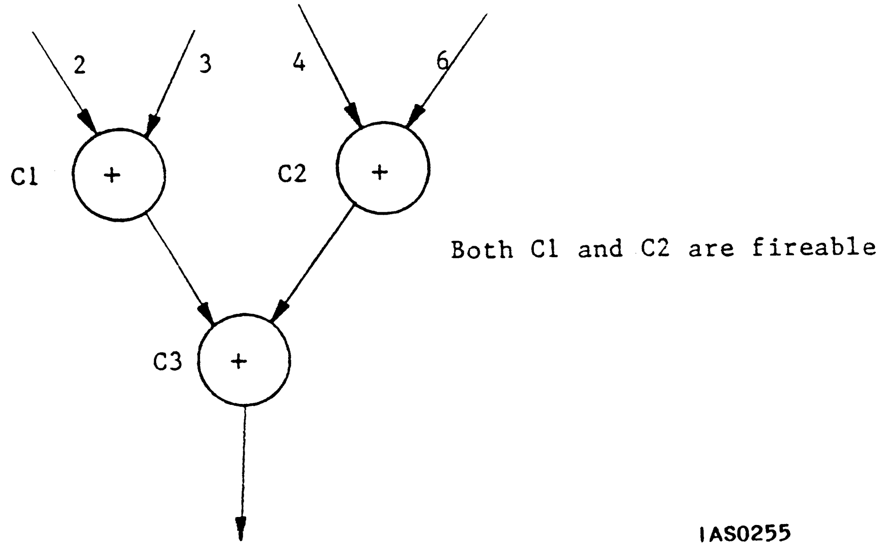
Consider a particular vertex type which simply adds two integers to produce an output integer. Such a vertex needs both of its inputs in order to do anything meaningful. Hence the firing set of this addition vertex is all inputs. When the firing set requires all input ports to have a token then the firing set is termed conjunctive.

Non-conjunctive firing sets are called disjunctive. This distinction turns out to be important and will be discussed shortly.

The terms conjunctive and disjunctive can also be used to describe the output ports of a vertex. If, after firing, a vertex places a token on all of its outputs then such a vertex is output conjunctive. If not, then it is output disjunctive. Output disjunctive functions are often said to behave under a disjunctive output rule. Similarly an input disjunctive function behaves under a disjunctive input rule. Similar meanings apply for conjunctive input and output rules.

Function types which are conjunctive for both input and output are almost trivial to use correctly, and that makes life easy for a programmer. Unfortunately, the need exists for disjunctive vertices as well. For the moment, the discussion will stay in the nice conjunctive domain in order to present some important and fundamental FGN properties.

In data-driven networks, each vertex can be viewed as an autonomous processing element. As such it is not necessarily synchronized with other vertices in the network. A network of vertices behaves as an asynchronous processing ensemble. Due to the fact that each vertex is an independent entity, these networks contain high levels of concurrency. Figure 2-2 shows a simple network of addition functions.



IAS0255

Figure 2-2. Simple Spatial Concurrency Example

At time 1 both vertices C1 and C2 are fireable, independent of each other. This is indicated in the graph by the lack of an arc connecting vertices C1 and C2. Actions in FGNs are sequenced only by the existence of data dependency, which is indicated by an arc. If vertex C1 produces a value needed by vertex C3, then C1 must fire before C3. This is the case in Figure 2-2. No other sequencing mechanism is necessary.

C1 and C2 are independent -- there is no arc connecting them to indicate data-dependency. In this example, C1 and C2 fired between time 1 and time 2 but it may not have been possible to determine the order in which they fired. If the time frame had been divided more finely (with points 1.1, 1.2, and so on), then a different sequence of events may have been observed. Here are three such possible sequences:

- C1 and C2 fired between time 1.n and 1.(n+1). In this case C1 and C2 still appear to have been active simultaneously. However, if the time line was viewed with an even finer grain set of observations, then this may or may not continue to be the case.

- C1 may have fired between time 1.3 and 1.4 and C2 may have fired between 1.5 and 1.6. In this case the firing of C1 definitely preceded the firing of C2.

- C2 may have fired between 1.1 and 1.5 and C1 may have fired between 1.7 and 1.8. If this were the case then we have observed at a fine enough grain to have seen, at times 1.2 through 1.4, that C2 was firing but had not delivered its output token until time 1.5. In this case, regardless of the duration of C2's firing, C2 fired before C1.

- C1 fired between time 1.1 and 1.6, and C2 fired between time 1.3 and 1.9. In this case the independent firings overlapped in real time.

What is important to note is that regardless of which of the four possibilities did actually take place, the result of the program is the same. The actual firing order of independent vertices (those not connected by a directed path of arcs) is unimportant. In fact, trying to figure out which independent vertices fire first in real time is an impediment to proper thinking with respect to FGN languages.

This property of concurrency is an important difference from traditional languages, where exactly one instruction is active in any particular time step. This exactly-one-instruction-at-a-time style is called a total ordering. Totally ordered programs -- sequential programs -- clearly do not contain any concurrency structure.

FGN programs represent partial orderings of the actions represented by the vertex functions. Vertices which are independent are unordered, while vertices which are connected by arcs or paths are totally ordered.

Vertices C1 and C2 in the diagram are concurrent operations. In this document, such instances of concurrency due to vertex independence will be called spatial concurrency. (Other terms for such structures which may also be seen in the general literature are horizontal concurrency or parallelism.)

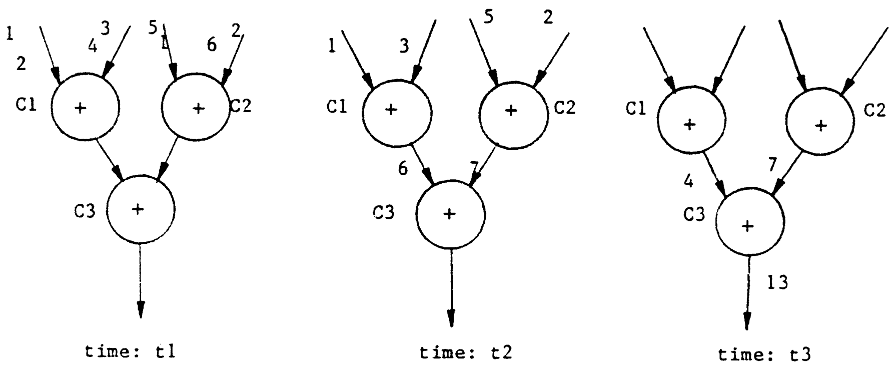
As an aside, it is worthwhile to note the differences between the nature of concurrency represented by the program and the concurrency which is exploited at execution time. It is interesting that to date there has been very little relationship between runtime and program concurrency. Typically, a concurrent program written in a language such as Concurrent Pascal has been executed on a one-instruction-at-a-time machine. Conversely, concurrent execution on machines such as the CRAY-1, IBM 360/91, and others has been done for totally ordered programming languages such as FORTRAN.

It is not the purpose of this document to point out the large number of ways that things could be done better. It will suffice to point out that concurrency in programs often leads to a more natural expression of a solution to a problem, allows inherently more efficient algorithms to be discovered, and may be more efficiently executed on computers of the future which may well be able to take advantage of program concurrency. Execution concurrency, on the other hand, is determined by the machine a program runs on. It is useful primarily for speed.

The focus of this paper will be on program concurrency, since the PS 300 programmer cannot intelligently control the actual execution speed of the PS 300 hardware by creating any special concurrency structure.

The nature of the arcs in a FGN program has not been examined very closely. They have been viewed simply as a register capable of storing a single input token. But if an arc was a queue of indefinite length, it would be capable of holding an arbitrary number of tokens.

The example in Figure 2-3 shows the same FGN addition network as before but with several tokens queued on the arcs.



where t1 is before t2 which is before t3

Figure 2-3. Queue-arcs Permit Pipelining

At time t2 all three vertices are fireable. The queues allow the FGN program to represent a pipelined execution style on streams of input tokens. Pipelining contains a form of concurrency -- vertices which are linked together in a line can fire at the same time. Vertices further down the pipeline will be firing as a result of tokens which were sent by earlier functions at the same time the earlier functions produce new tokens. This concurrency is obviously different from spatial concurrency, in much the same way that electric circuits in parallel are different from circuits in series.

Pipelined concurrency will be called temporal concurrency because it occurs with linear streams of time-ordered, sequenced tokens. (The general literature may also use terms such as vertical concurrency, temporal parallelism, or just simply pipelining. The choice of terms is not particularly important; spatial and temporal concurrency will be used here.)

Figure 2-3 shows that if the arcs behave like queues, then FGNs nicely represent an arbitrary mixture of both temporal and spatial concurrency. In some sense FGN programs of this form are maximally concurrent program representations. This is due to the fact that actions in such programs are sequenced only by the availability of their input data (operands).

The argument could be carried further to claim that FGNs contain only the sequencing constraints required by the algorithm and therefore allow the most natural procedural representation of the algorithm possible. The programmer only needs to worry about the things which are required to be sequenced -- this is essentially what the programmer specifies when an arc is drawn. Spatial concurrency does not need to be specified explicitly and therefore just falls out for free.

This is a simplification, but experience has shown that certain amounts of concurrency seem to just appear in a FGN program. This is certainly more natural than worrying about what should be concurrent and what must be sequenced, and then specifying both. Unfortunately, the ease and utility of a programming representation is affected by other factors besides sequencing. But with respect to sequencing the argument is valid.

Each vertex of a FGN program acts as an independent processing site which operates in an isolated, self-controlled, and self-timed fashion. When the firing set is present the vertex fires and sends out its results on the output arcs.

It does not matter in any functional sense whether or not the firing is slow or fast. It also does not matter whether or not the outputs are sent in any particular order or whether the inputs are used in any particular order. An instance of firing is considered complete only after all of the firing set tokens have been consumed and all the results have been sent. As long as the firing time is finite, the functional properties of a FGN program are not affected.

This self-timed behavior is yet another freedom that FGN programmers have that is not typically available in conventional programming languages. The benefits of these freedoms are many and will be subsequently enumerated in some detail.

Often in an FGN program there will be various types of inscriptions scattered around the graph. In Figures 2-2 and 2-3 the inscriptions C1, C2, and C3 were used to label the individual addition vertices. These labels had no semantic value and served only as comments. (Often comments can be used in the normal sense to describe in natural language some aspect of the FGN.) Labels could also be attached to arcs to describe the nature of the data tokens which pass over that arc.

In some FGN languages, certain types of inscriptions may also have semantic roles. For example, the + inscription in the vertices C1, C2, and C3 indicates that these vertices perform an addition. Other possibilities might indicate arc data types, input or output ports, and so on.

The discussion so far has taken a fairly purest view of FGN programs. Any particular FGN language specifies certain vertex, arc, and inscription semantics to create a hopefully useful programming language. These choices may in fact destroy some of the nice properties discussed in this section if they are made carelessly. In addition, restrictions may be made on FGN topology to disallow certain types of undesirable behavior. Two examples of such topological restrictions might be:

- Only one arc may be connected to a particular input port of a vertex. This would prevent non-deterministic merging of data tokens at that port.

- If input and output data ports are typed, then it would make sense to disallow an arc to connect ports of incompatible types.

2.3 DISJUNCTIVE VERTICES AND DECISIONS

FGN operation would be simple if all vertices had conjunctive firing rules and conjunctive output functions. Unfortunately if this were the case, important program constructs like decisions would be difficult.

Conditional execution (that is, the FGN equivalent to an IF-THEN-ELSE statement in a conventional programming language) is an essential aspect of any usable programming language. In data-driven languages, activities are triggered into action by the arrival of the firing set tokens. In order to conditionally select function A there are two mechanisms which can be used:

- Send a token to A which says do not do anything if the condition holds to not perform A. Such a token might be called an omission token. If A was to be selected then a regular or "commission" token would be sent. The advantage of this approach is that firing rules could remain conjunctive and analysis of a FGN program would be made easier. The disadvantage is that a second class of tokens -- omission tokens -- have to be created, transmitted, observed, and so on. This may consume valuable resources without directly producing any desired value as a result.

- More typically FGN language designers conditionally route tokens to vertices. The advantage of this method is that vertex firings are always done with real data (so no omission tokens have to be created and resources are not unnecessarily consumed). The disadvantage is that conditional routing makes disjunctive vertex types necessary.

Discussion of the first mechanism -- using omission tokens -- will not be pursued. The second mechanism -- conditional routing with disjunctive vertices -- will prove more useful for PS 300-style programming.

2.4 SYMMETRIC DECISION STRUCTURES

Let us define two vertices: DISTRIBUTE (DIST) and SELECT (SEL). Figure 2-4 shows them.

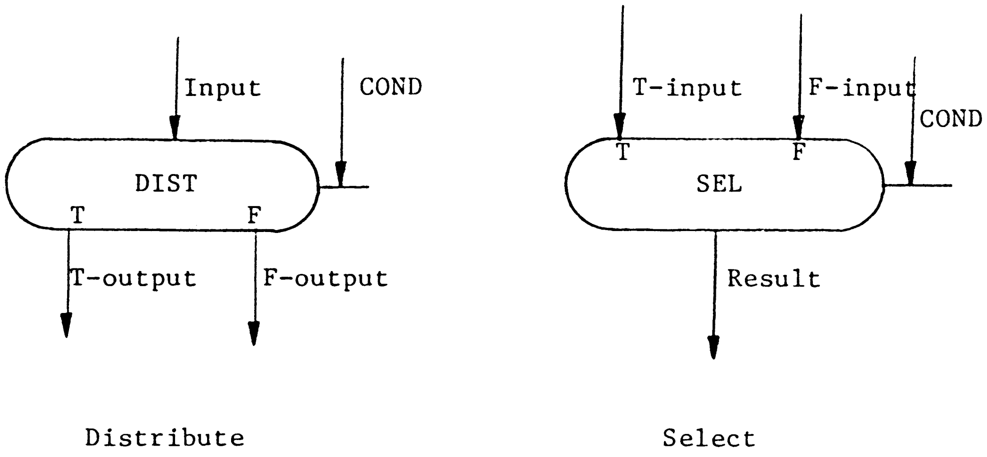


Figure 2-4. DISTRIBUTE and SELECT Cells

The arcs are labeled for reference. The COND arc carries a Boolean value. The arcs labeled T carry the token produced by DIST or used by SEL if COND is true. F indicates the paths used if COND is false.

Both DIST and SEL simply pass the input to the output unchanged. DIST conditionally distributes its input to the proper output arc (one-to-many distribution). Conversely, SEL conditionally selects an input token to be passed to its output (many-to-one selection).

SEL has a disjunctive firing set and, since there is only one output anyway, a conjunctive output set. COND is always needed for SEL to fire, but only one of the other two inputs is needed (depending on the value on COND).

DIST, on the other hand, has a conjunctive firing set and a disjunctive output set, where the COND input specifies the output arc which will receive the output token.

Figure 2-5 shows a FGN program which adds 1 to a number A if it is greater than zero and subtracts 1 from A if it is less than or equal to zero.

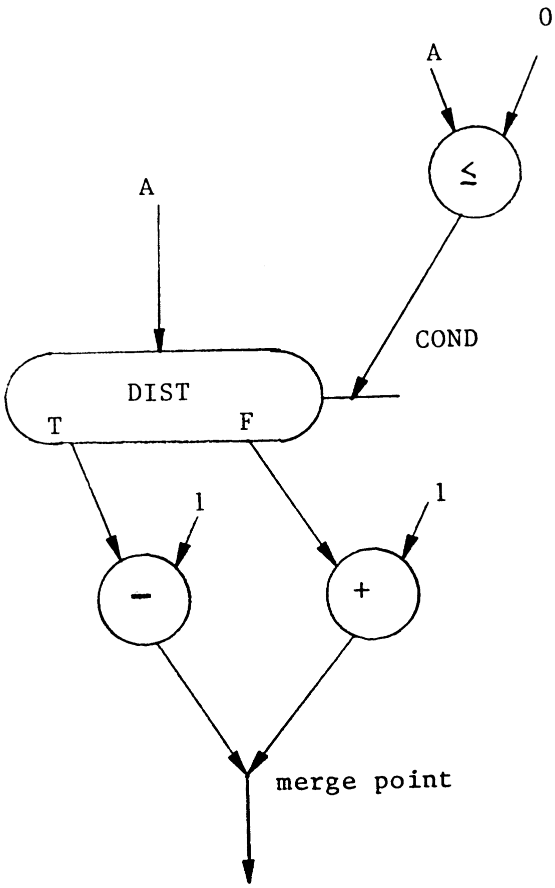


Figure 2-5. FGN to Computer IF A <1 THEN A - 1 ELSE A + 1

Assume for now that the constants 0 and 1 are tokens which are never destroyed -- as soon as they are consumed during the firing of the vertex, they reappear. The output arcs from the subtraction and addition vertices are simply merged together into a single result arc.

This merge works fine if exactly one value of A is put onto each of the two A arcs. No knowledge of vertex firing speeds is necessary to understand what the result will be. If, however, two identical streams of values are sent into the two A arcs then the output order will be affected by the speeds of the two parallel paths which contain the add and subtract vertices.

Here's how that might work. Let's say the first token gets routed to the add vertex. If the add vertex fires much slower than the subtract vertex in the other path, it will still be processing that first token as the second token gets routed to the subtract vertex and causes it to fire. The first output would thus result from the second input -- the output order would differ from the input order. If order was important in the network, that would introduce a problem.

To insure the correct output order, you would need to take the execution speeds of each parallel path into account, but that only complicates matters. A much better programming technique would be to disallow merged arcs and the non-determinism they can introduce.

By placing a SEL at the merge point, which gets a copy of the conditional value, the resultant FGN (shown in Figure 2-6) is once again nice and well-behaved even with the pipelined behavior resulting from streams of tokens.

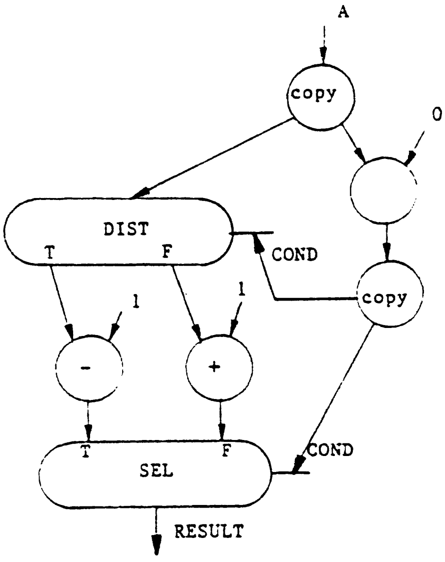


Figure 2-6. Well-Behaved IF Example Under Pipelining

Note from the diagram that a new vertex type called COPY has been introduced. COPY takes an input token and produces two identical tokens on each output. Also note the symmetry of the decision structure in this example. The single stream fanned out to many streams at the DIST vertex, which were then brought back into a single stream at the other end with a SEL. Without becoming unnecessarily formal, it will suffice to say that these symmetric decision structures are always well-behaved.

There are a number of ways to incorporate well-behaved decision structures into FGN languages (it is the assumption here that nobody is all that interested in non-well-behaved mechanisms -- which in fact are even easier to invent). Some possibilities are:

- Incorporation of high-level vertex types which implicitly contain symmetric decision structures. The simplest would be an IF-THEN-ELSE statement box, but better choices certainly exist.

- Allowing DIST and SEL type vertices to be used directly by the programmer. Checking for legal decision structure could then be done by the compiler in a fairly straightforward fashion. The main problem with this approach is that if the value of the token on the COND arc is an illegal value, then something weird may happen. In general, this scheme only works with certain error-handling mechanisms since the compiler has no way of anticipating run-time token values.

- Hope that the programmer gets it right and forgo proper design of the compiler and vertex type set. This is clearly the worst choice but unfortunately has been the most common approach.

Note that the PS 300 function net language has a function (F:SWITCH) which is exactly the same as the DIST function shown here. There is no single PS 300 function equivalent to the SEL function, but the SEL function can be programmed as a small network using F:SWITCH and several F:SYNC functions with an appropriate initial marking.

2.5 ITERATIONS

Iterations are also performed differently in data-driven sytems than in sequential programs. In a data-driven iteration an initial set of tokens arrives at the net which performs the iteration. If something indicates the iteration is to run, these tokens are then sent through the part of the net which corresponds to the body of the iteration.

The body produces a set of partial results of the iteration which are fed back around into the iteration the same way the initial set was, and the cycle repeats itself. The problem is that if pipelined tokens can enter the iterative network, the feedback stream and the initial stream must not be interleaved -- this would introduce indeterminacy.

This separation of new tokens from iterated tokens can be done either explicitly when the program is written or dynamically at run time. There are a number of ways to do each, one of which is presented here. It should also come as no surprise that the method presented here is in fact the one that will prove directly applicable to PS 300 function net programs.

With this method, the programmer provides a decision structure to keep the two streams separate. This means that a set of initial tokens is selected (via a SEL function). If the predicate evaluates to TRUE (meaning do the body of the iteration), then the tokens are distributed (via a DIST function) to the body. New initial sets are blocked from the iteration to allow the feedback token sets to iterate until the predicate becomes FALSE, indicating that the iteration is done. The tokens are then distributed out of the iteration net as results. A new initial set is then enabled to be selected which will start up the next instance of the iteration.

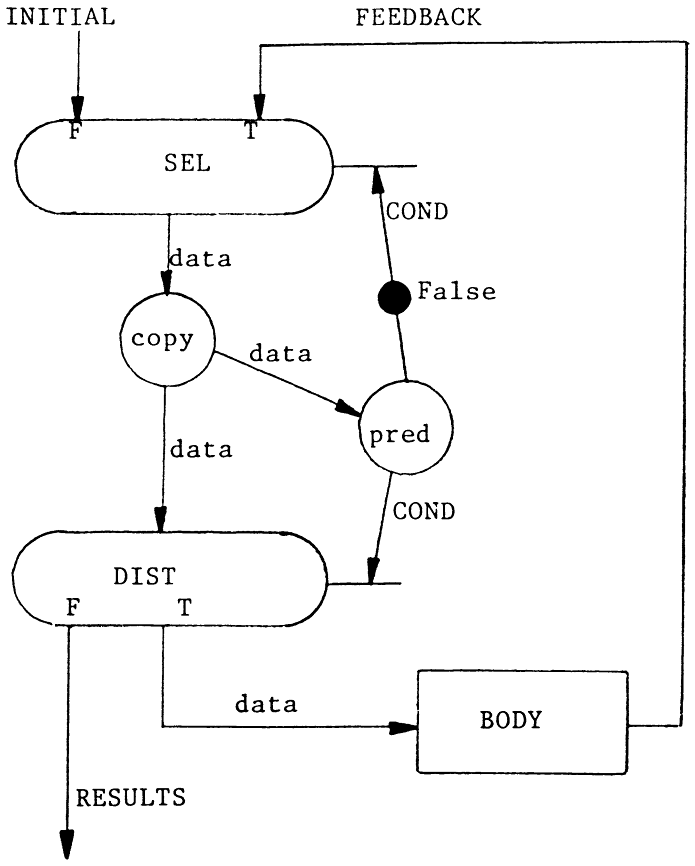


Figure 2-7. Simple WHILE Type Iterative Structure

The detailed scenario of the actions performed by this iterative structure is:

1. When an INITIAL token arrives, SEL fires and the data is copied and sent to PRED. The predicate then fires to decide whether or not the iteration is to be started. Let's assume that the predicate produces a TRUE, meaning that it does start.

2. The TRUE is copied and sent to SEL and DIST. At this point, even if a new INITIAL token were to arrive, the SEL vertex could not fire because its condition arc would contain a TRUE. Until a FEEDBACK token arrives nothing can happen. It is precisely this blocking action of the SEL vertex that prevents tokens on the INITIAL and FEEDBACK arcs from getting mixed up.

3. DIST fires and the data tokens are passed through the BODY of the iteration, which in turn produces a set of new values. These are passed back to the top into the T input of SEL. SEL already has a TRUE token on its COND line, and so the recirculated values are passed on as in step 1.

4. This process continues until FALSE appears on the COND line of SEL. Then the data is passed out the RESULTS side of the DIST vertex. Note that when this happens, FALSE is left on the COND line of SEL. This is identical to the initial configuration of the FGN -- the network is ready to begin a new set of iterations with new initial values. This simple network can thus respond to a pipelined stream of tokens entering it.

The tokens which are present on the arcs of an FGN program initially are called the initial marking of the graph. There clearly are some restrictions on initial markings if a particular FGN program is to operate properly. In the previous example, for instance, if the initial value placed on the COND line of the SEL vertex was a TRUE, then no vertex of the program would ever be fireable and the net would be dead forever.

The theory of initial markings for a particular graph topology is extensive. The next chapter will discuss the properties that determine what sort of initial markings are allowable (that is, which initial markings will lead to well-behaved FGN programs).

The iteration control mechanism in the example illustrates how FGN loops can be constructed. As was pointed out with decision structures, other possibilities certainly exist. Higher or lower level vertex types can be used, but the basic style would be the same.

One iterative technique, however, is rather different. This style has been called unraveled iteration or U-iteration. U-iteration allows an arbitrary number of instantiations of the same iteration to proceed concurrently. A new instance is started for every token arriving on the INITIAL line. This method does not need to block new INITIAL tokens while the current iteration is active, as was the case in the previous example.

U-iteration can be visualized from Figure 2-5 with only a minor change. By breaking the feedback arc and grafting an identical but new version of the net at the breakpoint every time a new initial token comes along, the graph grows dynamically as long as it needs to. This allows full pipelining of the iterative structure. Without waiting for a set of initial tokens to finish looping through an iteration, a later set can begin.

An additional mechanism needs to be provided at runtime to make sure the output order of RESULT tokens is correct for U-iteration, but as this paper concentrates on linguistic mechanisms rather than execution requirements we will leave the discussion of iteration at this point.

2.6 CALLS

The mechanism to do the equivalent of a CALL, be it recursive or not, is similar to CALL mechanisms in conventional languages. Semantically, however, there is an additional freedom with data-driven CALLs that is not available in languages like Pascal or FORTRAN.

In sequential programming, arguments or formal parameter values typically are supplied from a single site such as the calling subprogram. A data-driven CALL vertex is activated by the arrival of its firing set just like any other vertex. These firing set tokens can be supplied from several concurrently active subprograms if that is what is desired.

Most FGN CALL mechanisms behave like a CALL-by-value in a language like Pascal. This is due to the fact that tokens are best thought of as values. Some FGN language designers have created CALL mechanisms which are more like CALL-by-reference or CALL-by-name, but in these cases they have also made compromises to the basic data-driven semantics of their languages. For our purposes here the CALL-by-value view will be the most productive.

2.7 FAN-IN AND FAN-OUT

We have seen two rather different views of fan-out in FGN programs. Fan-out is a vertex output property. If a vertex has two output arcs then it has a fan-out of two.

If a vertex has a fan-out greater than 1, and the outputs all (conjunctively) receive tokens when the vertex fires, then concurrency is inherently created. It does not matter whether the tokens are copies of a single value or are different values. The result -- tokens flowing down different paths at the same time -- is a fundamental illustration of concurrency.

The other form of fan-out is when only part of the output paths receive tokens. This disjunctive fan-out is a fundamental indication of a decision structure, and an example is the DIST vertex shown in the previous two figures.

Fan-in, on the other hand, refers to the number of input arcs a vertex has. Fan-in also has two fundamental characterizations:

- Synchronization. Conjunctive fan-in at a vertex with a conjunctive firing set is an indication that concurrent activities are being synchronized at that vertex. All of the tokens must be there before any are allowed to move on, and when they do move, they all move at the same time.

- DECISION termination. Disjunctive fan-in, as we have seen, is the finish for what began earlier as the result of a decision structure.

Fan-in is a way of merging tokens. There is an additional type of merge that is quite different in nature from synchronization or decision termination, and that is arbitration.

Arbitration is a type of token-merging where tokens arrive on a number of input arcs and are passed out on a single output arc on a first-come-first-served basis. Arbitration only arises when concurrent activity is allowed (such as with data-driven programming) and as such is not seen in conventional programming languages.

This can be an inherently non-deterministic operation and in most FGN languages it is. If non-determinacy is allowed, then two or more arcs going to the same input port is usually an implicit indication of a non-deterministic arbiter.

However, non-deterministic program behavior is usually considered to be an undesirable property. Figure 2-8 shows two forms of non-deterministic arbitration (implicit and explicit) and an arbiter that can be used for deterministic operation.

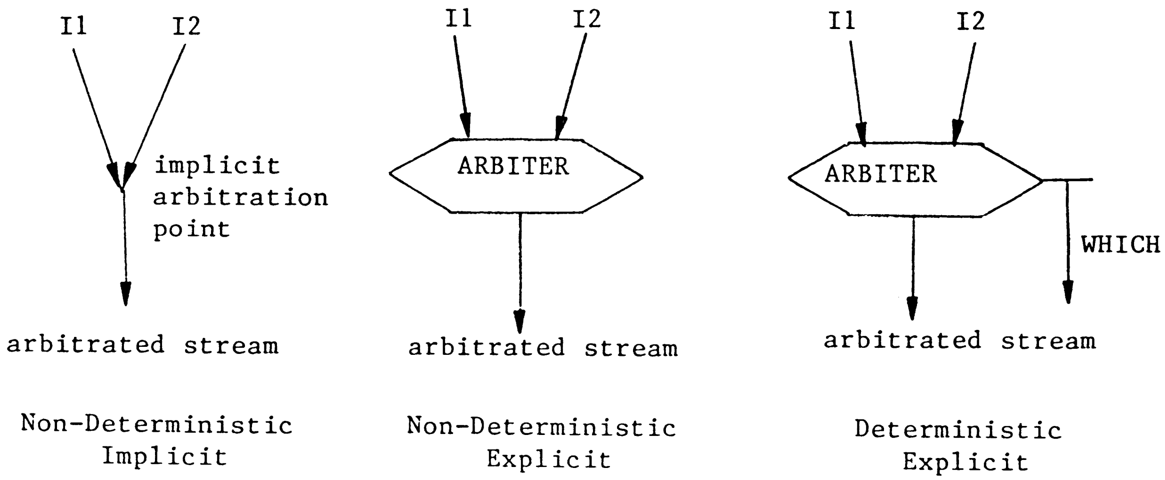


Figure 2-8. 3 Styles of Arbiter Indications

In general, programmers are cautioned against using non-deterministic arbiters of either type. Such usage almost always turns out to be a major mistake. Unfortunately most FGN languages do not have primitives for deterministic arbitration. There is a way around this problem, however, if token types are flexible enough -- we'll discuss this in a moment.

The key to deterministic arbitration is to create another token stream (WHICH in Figure 2-8) that indicates which input port (I1...In) produced the token placed on OUT. This identifies the input token so it can later be matched with its output in the proper order.

An example will help to illustrate this point. Suppose that three independent processes need to access a very expensive resource. In order to avoid duplicating the expensive resource, the three processes share it. Let the processes be called P1, P2, and P3 and let us assume that they query a huge data base created by the World Knowledge Corporation (WKC), which contains everything known about everything. Each of the processes gives the WKC data base system (WKC-DBS in the diagram) a single word and receives in return a token describing all that is known about the query.

The FGN program to do this is actually quite simple and is shown in Figure 2-9.

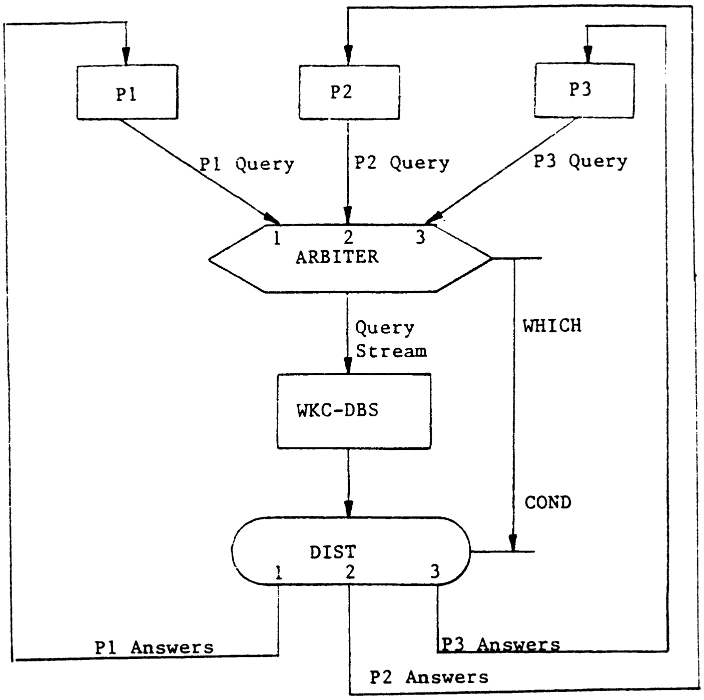


Figure 2-9. Deterministic Arbiter used to Share an Expensive Resource

In this example any of the three processes can at any time send out a QUERY token, which is merged by the ARBITER and sent to the WKC-DBS. The REPLY token is sent to the DIST. DIST knows from the token on the ARBITER's WHICH arc what process to send the REPLY token to. (Note that in this case the semantics of the DIST vertex have been expanded somewhat -- distribution is determined by an integer rather than a Boolean value.) Topologically this network can be thought of as an order-preserving, parallel-to-serial-to-parallel FGN program.

In the hypothetical language used for the previous example, a special vertex (with a WHICH arc) allowed you to match parallel input paths with parallel output paths in a deterministic way.

In FGN languages lacking a special vertex that allows deterministic arbitration, the programmer can build something similar by creating token values on the QUERY n lines which are two-element lists. The first element is the process number (which in the example was created by the arbiter), and the second element is the actual QUERY token. The only danger with this mechanism is that the language compiler must allow syntactically non-deterministic programs to be compiled. The burden is then on the programmer to insure that the program's behavior is deterministic. Clearly the best approach is to use another method that would not burden the programmer, since that load is usually a heavy one to begin with!

2.8 ERRORS

Errors are a perennial problem in any programming methodology, but in FGN programs the programmer must diagnose errors rather differently than he would in traditional programs. With sequential programs, a programmer would usually run the program and, if it does not work, run it again and observe the actions more closely until the problem is found. With partially-ordered programs (that is, FGN's) this does not necessarily work.

For example, if the problem is caused by some relative-timing error between two independent pieces of the FGN, then this timing discrepancy may not be repeatable due to the concurrent nature of the two pieces of code. It is therefore paramount to program in a way that prevents programs which exhibit this type of behavior.

One of the aims of this document is to aid prospective PS 300 programmers to develop such a style. The key is to never design programs which rely on runtime speeds of topologically independent subgraphs (vertices which aren't connected by arcs). Synchronization, arbitration, and decision structures, if used as described above, will usually insure that proper program structures are designed.

Still, the art of FGN language design is new. There are some bad FGN languages out there, which may lull the programmer into thinking that intrinsically dangerous FGN programs will work. If you are unfortunate enough to get a bad language as your vehicle, be very careful to analyze the decision, synchronization, arbitration, and other fan-in and fan-out structures provided by the particular language.

Fortunately, there are many aspects of FGN languages which make them easier to modify and debug than conventional languages. The lack of global variables, GOTO statements, and so on, make them inherently side-effect free.

The influence of one thing on another is also easy to see -- just follow the input paths backwards and you will find all of the actions which can affect the token values.

Runtime debugging facilities also affect the process regardless of the language. We will not elaborate on this topic, however, since the theme here is primarily language-oriented. Suffice it to say that there is no reason why FGN languages should not have good runtime debugging environments. The only thing that can negatively influence these claims is an improper choice of vertex types; this topic will be discussed later.

There is one additional point that deserves discussion, and that is how can errors be detected and represented. When a vertex firing creates an error there are a number of things that may have caused it. Two of the three possibilities are:

- Incorrect token values

- Incorrect token type. This would not be the case in a strongly typed FGN language where type-checking was performed at compile-time.

Dealing with these errors is straightforward -- the previous discussion indicates how they may be handled. The other possibility, which is unique to data-driven programs, is not an error of commission (as the previous ones are) but an error of omission. That is, the vertex should have fired but could not because something was left out. Errors of omission are always caused by disjunctive fan-in or fan-out. The cure for this problem is properly structured decisions, iterations, and arbitrations, which have been previously discussed.

With errors of commission it is important that something be done, because if nothing is done then the output arcs of an error firing vertex will either:

- Receive incorrect values, which will propagate and cause other errors. In the process, they will mask the site where something initially went wrong,

- Or receive no tokens. This will essentially turn an error of commission into an error of omission and will only add to the confusion.

A number of things can be done when an error is noticed at run-time; the two most common and useful strategies are:

- To suspend execution and allow the programmer to examine the input tokens of the vertex in error and possibly find what caused the error.

- To place special error-valued output tokens on the output arcs (and possibly into an error-log file, which holds the identification of the error vertex and the values of the input tokens which caused the problem). The advantage to this scheme is that the FGN can keep running, propagating the error tokens which are passed through as NOP (no-op) indications to the vertices they meet on the way to this final destination. This allows concurrent pieces of the program to continue on so that as much as possible can be tested before the programmer needs to dive in and find the problem.

2.9 CONSTANTS

Constants are a special topic. Typically there are two ways that constants have been treated. These methods are depicted in Figure 2-10.

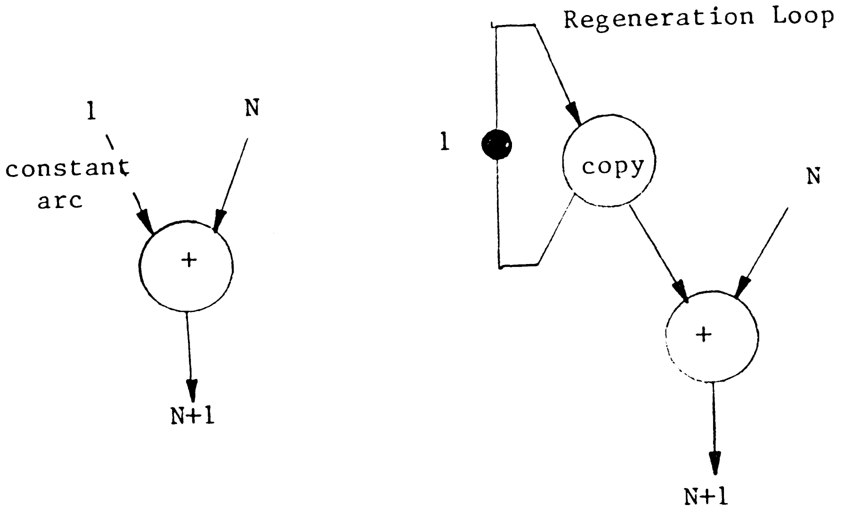


Figure 2-10. Constant Specification Methods

In the constant arc case, the arc is viewed as always containing a token with the constant value. The firing set of vertices which contain a constant arc is consequently reduced by 1. This is no problem unless a vertex has all constant arcs as inputs. In that case the vertex would be firing continually, which could spell disaster for the program.

There is fortunately a simple solution to prevent vertices with all constant inputs. At compile-time, execute the always fireable vertex and place the results on the output arcs which are now of constant type. The advantage of the constant arc approach in that no unnecessary vertex firings occur, but the scheme requires the ability to handle two arc types: constant and regular.

The other approach is called regeneration and involves making a copy of the constant value as it moves on to its destination and passing that copy back around to reprime the constant copy vertex. This scheme requires that the loop arc be initially marked with the constant value. The advantage of this scheme is that only one arc type is needed but the disadvantages are:

- Extra vertex firings are needed to regenerate the same value.
- Initial markings are required.
- Initially fireable vertices are also inherent in this scheme. They must be found and executed even though no tokens have been delivered into the loaded network.
- Since regeneration is a closed loop net, the regeneration part may fire an arbitrary number of times causing an unbounded supply of constant tokens to queue on the feedback loop. This consumes both storage and processing resources, which is clearly not a particularly clever idea.

Other constant mechanisms exist, but these are the two types most frequently found in existing FGN languages.

2.10 I/O

The final basic topic to be covered is I/O. I/O always seems to be a mess, and most of the I/O problem arises from poor language design. Here we view I/O as information which either comes into the program or leaves it. The formatting problem is the same as with conventional languages, but there is one important difference which deserves discussion.

Data-driven I/O should be pushed into activity by its firing set just as any other vertex is activated. This means that vertices that perform I/O functions should be initiated by tokens, and that tokens should be returned to indicate completion. If this is done, then the nice self-timed behavior is preserved. Unfortunately many FGN language designers do not understand this, and so I/O presents a problem in some languages.

For input, a prompt can be given and the input data is the indication of completion of the input. For output the data itself initiates the task and a dummy token indicating that the output operation completed should be the response.

If this style is adhered to then inputs are easy to direct to the proper part of the FGN program. Outputs which may be generated concurrently will appear in the proper order on a shared-output device. Sharing I/O devices by concurrent I/O operations can be structured similarly to the shared resource example shown in Figure 2-9.

If input is viewed (albeit dangerously) as a token appearing on an arc whenever it gets there, then the programmer must carefully analyze the token-flow possibilities of the FGN to see that all possible actions will in fact permit proper program behavior.

Similarly, if output is viewed as a token sink, the programmer must also be careful to analyze the FGN to insure that errors of omission do not result. In general these methods are not all that bad as long as I/O resources are not shared. If they are shared, the I/O actions should be encapsulated in a program piece which allows proper sharing and synchronization of the atomic actions. As usual a good programming style can go a long way toward curing the problems caused by careless language design.

This concludes the discussion of FGN programming basics. A thorough understanding of the topics presented in this section is necessary before proceeding.

3. NET PROPERTIES

In conventional programming languages, certain structural properties of a program are important. These properties involve control structures such as properly nested decisions or iterations, but also involve scope rules and variable usage. Creating programs that contain these properties is what good programming style is all about.

The advantages of such programs are widely touted in the general literature, and it will suffice here to say that good programming style concerns itself with the entire programming spectrum from human efficiency to runtime speeds; and from program design through testing, modification, and on to the point where future enhancements can be made efficiently by a totally new programmer community.

FGN programs also have properties which should be the goal of proper programming style. Due to the nature of data as tokens in FGN programs, the proper variable structure, scope, and usage of conventional programs have no direct FGN counterpart. The issue of proper use of branch statements, for instance, simply does not come up in FGN programs. Properly nested decisions and iterations are equally important in FGN programs as they are in languages like Pascal or FORTRAN.

In addition, however, some new program properties must be understood before a FGN programmer can develop a good programming style. These result from the intrinsic concurrent and self-timed nature of FGN program actions. Some of these properties are influenced by improper FGN language design. Whether the language itself makes things easier or harder, it is important for a programmer to understand these properties and strive to create programs which contain them.

Some reasonable questions to ask with respect to a given FGN program are:

- Does the program continue to run until it provides output tokens?
- If the program works for a single token on its firing set arcs, will it always work as well if pipelined streams of tokens are sent to it?

- Does the program always produce the same answers for a given set of input values?
- Is it possible for mysterious things to happen which cause the program to behave differently from time to time?

The properties described in this section are the key to providing the answers to these questions.

3.1 PERSISTENCE

It is important that tokens on arcs do not randomly disappear. If they did, it would be extremely difficult to predict program behavior. Most FGN tokens are defined to have the property of persistence. A token is persistent if, when one is placed behind another token at the tail of an arc, it persists on the arc until it is consumed at the other end of the arc by the firing of a vertex. This is necessary to maintain locally-controlled action at the vertices of an FGN program.

Persistence is threatened if the producer of a token can remove from an arc a token that it has previously placed there. That could change the firing set of another vertex dynamically. This could be disastrous -- the receiving vertex could think it was fireable and start to fire, only to discover that a token has been removed.

Another threat to persistence is allowing tokens to be overwritten by later tokens. This creation and destruction of an unused data value may not be disastrous -- the program structure may not care if values are overwritten. But if the algorithm DOES care, then there is a problem. Persistence is a necessary condition to insure deterministic behavior for a FGN network.

3.2 LIVENESS

It is important that, once started, a FGN program does not destroy its tokens in a mysterious way which causes no vertex to be fireable. For example, let us consider the program shown in Figure 3-1.

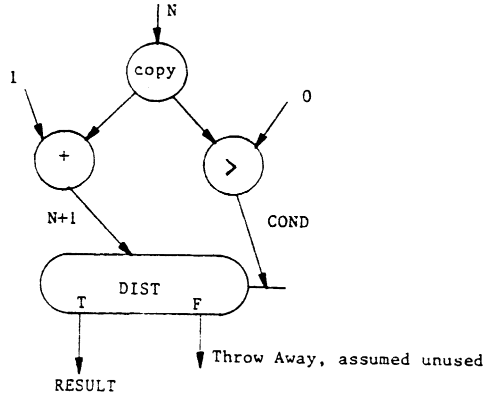


Figure 3-1. Example of FGN Which May Die

The program is a silly one, but it illustrates the point. The programmer expects the program to take positive values of N and produce a result that is the value of N plus 1. The problem is what happens when N is not positive. In this case, the value of N is thrown away and no result ever comes out of the program.

Clearly, the problem in Figure 3-1 is that the decision structure is not symmetrically closed. The parallel arcs T and F that leave DIST need to be merged somehow.

In data-driven languages this problem is an insidious one due to the self-timed nature of the semantic model. Data-driven programs, waiting for the firing set to arrive -- not for a time to arrive -- is how things happen. It is not particularly valid to say that something should have happened "by now".

Certainly there are some pragmatic things which could be done, but in a huge net of a million vertices containing a great deal of concurrency, the analysis of the total state of the system is an extremely counterproductive activity. In fact, the view of distributed control concurrent systems in terms of total system state is generally not productive.

Partially ordered, self-timed systems are nice because they permit analysis of parts of the system which are known to be independent from the influence of other parts. There is no need to make the analysis combinatorically intractable by worrying about the state of the entire system.

In general, it is easy to ascertain whether or not a FGN is live if all of the vertices are conjunctive for both input and output. The analysis proceeds as follows:

- Assume the output arcs of the FGN are live.
- If this is true then all of the vertices which generate tokens on these arcs are live, too.
- Since they are live, all of the input arcs to these vertices are live.
- Repeat the last two steps until all that's left are input arcs into the FGN.

If the input arcs that you have determined are live are the firing set of the FGN, then the FGN itself is live. If however, the firing set of the FGN contains more inputs than you can determine are live, the FGN is not live.

The liveness problem is always associated with vertices that are disjunctive for input or output. This fortunately is easier to check than you may imagine. If a disjunctive vertex is part of a subnet which has a symmetric decision structure (as previously discussed), then the FGN is live. A symmetric decision structure has conjunctive inputs and outputs. Inside any subnet, if it is symmetric, the tokens will always flow to produce conjunctive outputs regardless of the disjunction pattern.

You may be tempted to make assumptions about what decision paths can be taken in a given structure. These assumptions are always seeds for disaster. Always assume that the decision may be made in any direction, and symmetrically close the decision structure. Your program will perhaps contain a few more vertices but it will also contain less surprises. (Like airline travel, programming is one of those activities that is made less pleasant by surprises.)

3.3 BEING WELL-ORDERED

The property of being well-ordered applies to arcs. In this discussion we assume persistent tokens. A well-ordered arc delivers tokens to their destination in the same order they were placed on the arc.

Hence, queues are the only storage techique which makes sense for FGNs, with the exception of how to deal with constants. The queues can be of length 1, n, or indefinite. If the queues are of finite length, then the execution environment must be capable of delaying placement of a token on an arc which is full. The PS 300 employs arcs of indefinite length, limited only by the amount of available mass memory.

3.4 SAFETY

An additional net property which is often seen in network schemas but rarely has a direct application to FGN programming schemas is safety. A network is safe if there is no chance for a token to be placed on an arc which is full. The advantage of safe nets is that the execution environment would be relieved of the duty of forcing vertices which were about to place a token on a full arc to wait. Safe nets are designed so this just cannot happen.

Unfortunately, the ways in which safety can be guaranteed in FGN languages varies with the language so much that a full explanation here would be too lengthy. Suffice it to say that safety is only a necessary property if the arcs have a fixed capacity, and if the execution environment is not capable of delaying output placement to a full arc.

3.5 CLEAN

If a FGN is clean, then it exhibits no history-dependent behavior. After doing its work, a clean FGN returns exactly to the way it was with exactly the same initial markings. So if a set of input values produces a certain set of output values, then the same input set will always produce the same output values. In addition, a clean FGN is safe to use with pipelined or streamed inputs.

There is a theoretical topological analysis which can be made to determine whether or not an FGN is clean, but an easier nuts-and-bolts method is to:

- Take a given FGN and its initial marking.

- Put exactly one token of the proper type (if strong typing is a feature of the particular FGN language being used) on each arc of the FGN's firing set.

- Run the net until no more vertices are fireable.

- Remove all of the tokens from the output arcs of the FGN.

- Compare the resultant marking with the initial marking.

If the markings are the same then the FGN with the associated initial marking is clean -- otherwise it's "dirty". The term dirty stems from the residual garbage tokens (which affect future net behavior) which get trapped in a dirty net.

(The FGN must be live for this procedure to have much practical value, but it works in any case as a test for being clean.)

Note that the clean property applies to a FGN with an initial marking. Constant arcs and an empty initial marking are correctly contained within the scope of this procedural test.

Note also that vertices which when fired produce random output values are assumed to be absent from such a net, if the functionality claim previously made is to be true. If such randomness was present, non-deterministic behavior is usually considered to be different from the non-deterministic behavior exhibited by the garbage token non-determinacy nets which are caused by nets which are not clean.

Finally note that only nets containing loops, iterations, or circular directed paths have a non-trivial clean test. Circular paths in FGN programs, as in electrical circuit structures, are the primary indicator that static storage (in this case, the initial marking) is present.

3.6 WELL-BEHAVED

A FGN program is defined to be well-behaved if and only if:

- tokens are persistent

- arcs are well-ordered

- the FGN and its initial marking is live, safe, and clean.

The goal of proper FGN programming is to create well-behaved FGN programs.

3.7 OUTPUT FUNCTIONALITY

Determinacy has been mentioned several times. With FGN programs, determinacy has a slightly different flavor. Deterministic program behavior has traditionally meant that if an atomic history of program activity was kept, then the history would be repeated if the program was run again with the same input values.

Due to the concurrent and self-timed nature of FGN programs and the fact that they represent partially-ordered computations rather than totally-ordered ones, the repeatability of an atomic history does not make any sense.

Usually the property which is used instead is output functionality. A FGN program is output functional if, for a given initial marking, identical input streams produce identical output token streams.

The repeatability of program behavior is still the essence of this property, but the focus has changed from the atomic history to the "big picture" -- the order and values of the input and output tokens. The question becomes, "Is my FGN deterministic and repeatable overall" -- without worrying about the ordering of small operations inside the FGN.

The next problem is how to insure that a FGN program is output functional. If the program contains a vertex which, say, outputs random values regardless of its firing set values, then life is indeed hard.

Usually programs which contain randomness of this type are not supposed to be output functional. Fortunately most programs are supposed to be output functional, and if such random actions are contained in the program then they are encapsulated inside a program fragment which completely hides this randomness.

An example of such behavior would be a program which randomly assigns pairs of inputs to three different addition vertices called ADD1, ADD2, and ADD3. This program is shown in Figure 3-2.

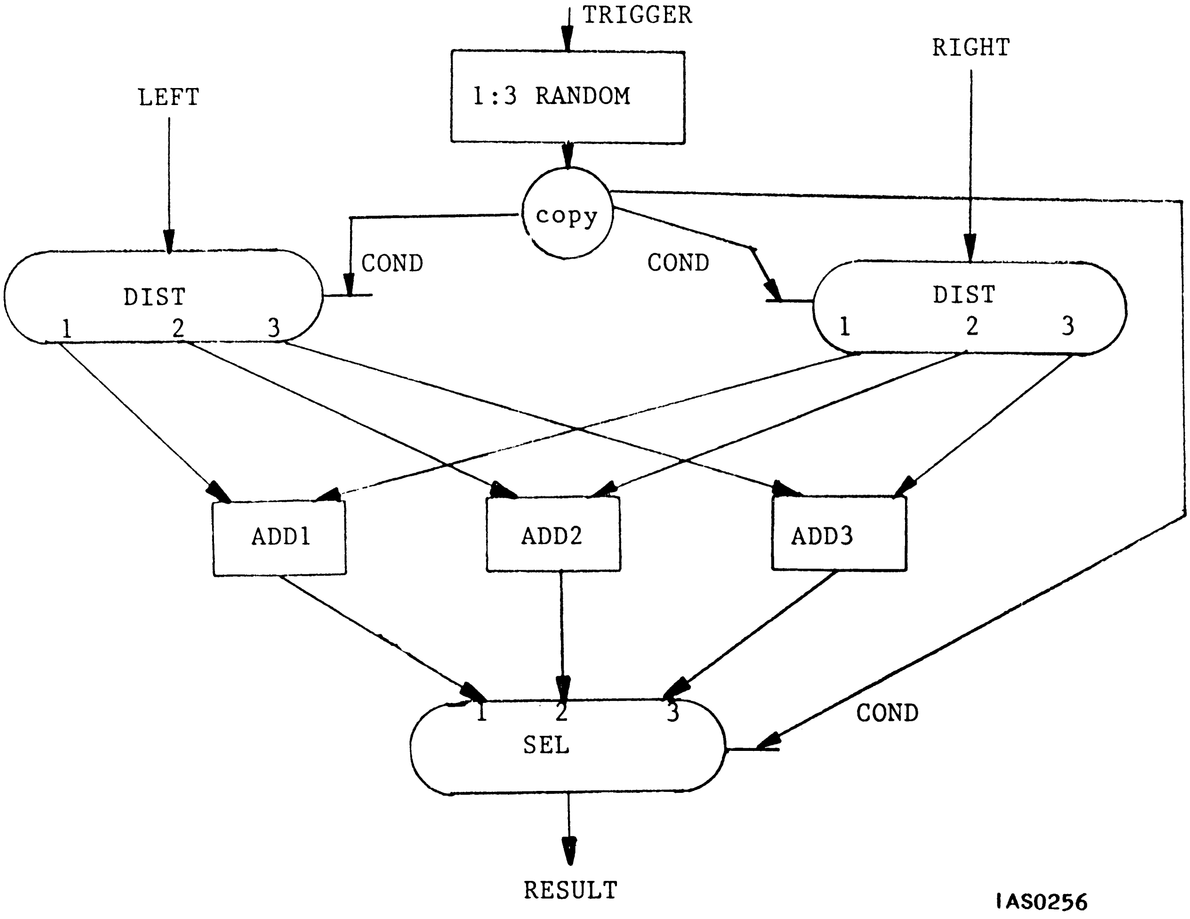


Figure 3-2. Encapsulation of Random Behavior

The vertex 1:3 RANDOM, regardless of the trigger value, generates at random either a 1, 2, or 3. This causes the LEFT and RIGHT operands to be sent to a random adder but the results are SELected in the same fashion as they were DISTributed. This makes the entire FGN program appear as if it contained no random action. Such random behavior may be called transparent randomness.

If a given FGN contains no randomness, no asymmetric decision structures, all iterations are well-nested, tokens are persistent, I/O is symmetric, and the FGN is live, clean, AND well-behaved then it will be output-functional. A formal proof of this fact is lengthy and somewhat complex but has been done. The above criteria are both necessary and sufficient conditions to establish output functionality.

3.8 SOME FINAL REMARKS ON PROPERTIES

As with all properties of "nice" programs, there is a big difference between practice and theory. This is as true for FGN programming as it is for Pascal.

A common scenario is for the programming methodology prophets to expound endlessly on the merits of the new religion while presenting their principles via laborious arguments and trivial program examples. The real programmers, on the other hand, scoff at the prophets with a claim that if that much analysis was required for each line of real code, then no useful program could ever be written. There is however an element of truth in both arguments, and quality and productivity can result from a proper balance of the two views. Programmers can create complex, well-structured programs without laboriously analyzing every atomic action in great detail.

In general, good programmers understand the principles of good programming practice and the properties which their programs should contain. In the past 30 years the art of programming has come a long way, and there is no reason to go back to the dark ages and start all over with FGN languages.

If the PS 300 function network programmer can keep the principles and properties discussed in this section in mind, the knowledge will go a long way toward developing an efficient, surprise-free programming style. While the ideas presented here may seem new to programmers who have previously used sequential languages, the work on concurrent programming is actually over 20 years old. It is a mature discipline which is new only in that it has been applied to practical programming environments only recently.

4. SPECIFIC FGN LANGUAGE REFINEMENTS

The discussion has thus far presented all of the basic concepts which, if understood, will provide a firm foundation for high-quality FGN programming. It has not listed specific DO's and DON'Ts because each particular FGN language is a bit different; a complete list would be impossible. It would be easy enough at this point simply to describe the PS 300 function networks and be done with it. But the philosophy here has been to present general concepts, which will then allow FGN programmers to develop their own practical principles list.

There are only two dangers in continuing the general discussion:

- Boredom. If this gets you then please skip the remainder of this chapter and proceed to the PS 300 stuff. If and when you get in trouble, then come back and read it (it will still be here) -- who knows it may help!

- Confusion. It may be confusing to some readers who find it difficult to assimilate abstract information without sitting down in front of a terminal and doing something concrete. If this is the case, then go do something and come back. It will not help conceptually, but it may relieve some frustration.

In general, it is useful to analyze a particular FGN language in terms of that language's types of tokens, arcs, and vertices with their associated semantics. These specific choices are termed refinements in this document.

By analyzing these refinements and how they are related to general FGN principles and properties, the programmer should be able to understand how the particular language being used fits into the general framework of data-driven languages. The programmer should also understand what practical programming restrictions should be applied in order to produce high quality code that is easy to debug, modify, understand, and hand off to others without significant danger.

4.1 ARC REFINEMENTS

Typical arc refinements include typing. If a particular FGN language allows different types to be mixed in a confusing way, the programmer must develop restrictions to prevent this confusion. In particular, usage must be restricted so that only well-ordered programs can be created.

In some FGN languages type-checking occurs at the vertices at runtime. If an input token is not compatible with a vertex it has been sent to, then some mechanism must recognize that and indicate that there is an error. Possible mechanisms for such error handling have been discussed.

In other languages, the arcs are considered to be strongly typed, in which case type checking can be done at compile time. A less sophisticated error-handling mechanism will be required for these languages.

In cases where the language has well-ordered arcs with limited capacity, the programmer must create safe programs, or the execution system must prevent an output token from being placed on an arc until the arc can receive it. Otherwise tokens may get lost -- and that would violate the persistence property.

Some systems use a single arc to represent composite token-carrying paths. This is similar to routing bundles of arcs. The advantage is more simplified graph structure. Usually languages which allow composite arcs also have vertex types which compose and dissect the bundles where needed. This is not an issue that concerns the programmer, since it is a syntactic nicety which either exists or is absent in a given FGN language.

4.2 TOKEN REFINEMENTS

What types of tokens does the language allow? Possibilities are:

- Simple integers, reals, and characters
- Complex tokens such as arrays, variable-length strings, list structures, and so on.
- Streams of tokens, which may or may not require beginning- and end-of-stream indicators.

If the language allows any type of token to hit any vertex then either the compiler or execution environment should be able to check if things go wrong and provide a sufficiently powerful error mechanism which will allow the programmer to find and correct the trouble spots.

Without this error-handling mechanism, the burden is on the programmer to carefully check all FGN paths and insure that the wrong type token does not get routed to a vertex which it will cause to go crazy. Crazy vertices are a sad thing to observe. Crazy programmers, while being somewhat easier to observe, are driven even crazier by crazy vertices.

The main property that should be guaranteed by the system is persistence. If this is not done by the language and its runtime environment, then the programmer must write programs which cause tokens to exist as if they were persistent.

4.3 VERTEX REFINEMENTS

There are diverse possibilities for vertex refinements. Choosing them is similar to choosing machine op-codes and language statement types. They reflect the designer's view of what is needed for a particular set of target applications or program styles. Deciding on vertex refinements is more of an art -- or a matter of the designer's personal style -- than a science. (A sort of Murphy's Law corollary states that no matter what the designer did decide will be viewed by every other programmer as the wrong decision. This is healthy, since complaining is a fun way to spend time when you can not think of anything else that you would rather do.)

Vertex types may be indicated by their shape, by a combination of shape and inscription, or even by their position in the graph. This is syntax stuff and except for ease of use, does not affect behavior of the program. Some languages only allow atomic vertices to be placed -- this inhibits hierarchical program design except for the equivalent of doing subroutine calls.

Other languages allow for non-atomic vertices to be defined. These non-atomic vertices are essentially "black box" encapsulations of the FGN which define its behavior. By opening the doors of the box, the substructure can be examined or created. Usually there is a one-to-one correspondence between the input and output arcs to the box and the arc structure of the FGN which defines the box.

Another difference is whether or not vertices have fixed numbers of input and output ports. This is not really something the programmer needs to worry about, as it is more a reflection on the flexibility of the compiler and the execution environment.

Copying is usually provided for either explicitly, using a "copy" vertex (this is the method used in the examples presented in this document). Or it may be implicit, indicated by multiple arcs leaving an output port of the vertex producing the value which is to be copied.

All that a programmer needs to be concerned about with copying is how to do it in the specific FGN language he's working with.

It is important to note which vertices do not behave in an output-functional manner -- those which contain "state" and therefore exhibit a history-dependent behavior, or those that generate random output values. These vertices are not inherently worthless, but they do cause non-deterministic behavior in the programs which contain them. If complete determinacy is desired, then all non-functional vertices must be encapsulated as shown in Figure 3-2 or via some similar technique which preserves output functionality.

It's also important to note vertices which have disjunctive input or output. These vertices must be used in a symmetric decision structure fashion in order to preserve a well-behaved network.

4.4 INSCRIPTIONS

Inscriptions can be used without affecting program function to name arcs and provide comments. These serve simply as ordinary comments do in von Neumann languages to aid in the readability of the code. Some inscriptions are used to type vertices, arcs, tokens, or act as global constants which can be assigned by the compiler to the appropriate symbol. Such inscriptions are not a need for concern, as they are just another syntactic method for specıfying what could be represented in other ways.

In some FGN languages, vertices may contain inscriptions which are programming instructions. It is important that the programmer write these mini-programs in a way that insures well-behaved function networks. The advantage of this approach is that it allows the programmer to describe actions that are normally thought of as a text string by simply writing the text string. Algebraic expressions are an example.

4.5 TOPOLOGY RESTRICTTONS

Some languages do not allow more than one arc to end at the same vertex input port. This is because a program that contains such merging is less likely to be well-ordered. The only time using this kind of merge will result in well-behaved programs is when token arrival on the merged arcs is restricted to guarantee well-ordered arrival at the port -- that is, if the input arcs are mutually exclusive. This means that if only one of the parallel arcs can contain a token at any given time, then no "race" between tokens in parallel arcs can occur. The resultant programs will still be well-behaved. If such a merging is allowed, the best advice is to NOT use the capability. If you must use it for some reason, be careful to restrict arcs to avoid token races.

Other topology restrictions usually apply to strongly-typed languages where the compiler or editor (even better) will complain if you assign an arc to the wrong type of vertex port.

4.6 ERROR HANDLING

Many error mechanisms exist and they must be evaluated with respect to the properties described in the previous section.

4.7 INITIAL MARKINGS

Some languages do not permit initial marking. If the one you are using does, be careful to specify initial markings which result in well-behaved programs. Be even more careful if the editor and compiler do not do "property checking" -- if they don't check programs before they execute to see they are well-ordered, safe, clean, correctly typed, and so on.

4.8 TEXT vs. GRAPHS

So far, this document has assumed that the programs are specified as graphs. This is the clearest and most natural way to specify FGN programs. Unfortunately, graphical editors are less commonly available and require a more sophisticated terminal than text editors, which have been around for years. Using a text editor won't affect programming methodology -- you'll still have to pay attention to the FGN properties that have been discussed. It will make actual programming more laborious, though. A graphical editor allows you to SEE how the vertices in a graph are connected. It isn't nearly as easy to visualize a FGN program disguised as a list of text commands.

It is always possible to provide a textual description of a graph via a set of arc and vertex statements which contain connection information. If property checking is done by the system you are using then life is nice. If it is up to you to do it, then in times of doubt draw pictures!

Fortunately, graphical specification is becoming more prevalent. System designers are discovering the merits of systems which prevent errors and are beginning to stop designing systems which promote error-creation. After all, making mistakes is easy enough, so why should people work hard to make it even easier?

A more philosophical issue is that there probably is no good reason why program structure should be indicated in the same way that poetry is, i.e. by some hokey indentation structure. A better view of program and poem similarity is based on the fact that both words start with a "P"

4.9 SOME FINAL REMARKS

At this point the diligent reader who has taken the time and effort to digest this unfortunately lengthy treatise has all of the right tools to grab a FGN language and do things properly. At least this will be the case after a moderate amount of practice.

The function net language of the PS 300, quite frankly, contains features which will not make your life as easy as it could be. With all FGN principles in mind, however, you'll be able to create surprise-free PS 300 function net programs.

5. PS 300 FUNCTION NET LANGUAGE

5.1 INITIAL STRATEGY

This document is not intended to be a PS 300 function net language reference manual, so the details of what each vertex type does is not included here. The definitive document for that is the **PS 300 User's Manual**, and the relevant chapter is Chapter 7, "Local Actions". If you have not already done so, you should go read that chapter prior to continuing with your study of this document.

What follows are some very general tips on how to create well-behaved programs using the PS 300 command language. If your understanding of general FGN properties is rather complete, then the remainder of this section will come merely as a review of what you already know, or perhaps a confirmation of what you expect to be true.

If there are some issues discussed in this section which you do not expect, then perhaps your understanding in this area is still a bit w^ak. This may be due to the fact that you have little experience with a new programming model or to a failure on the author's part to describe it in an way that is understandable to you, given your particular programming experiences and preferred terminology.

At any rate, you should list the areas in which you are weak and then perhaps go back in this document and reread the areas which discuss these topics.

5.2 GENERAL PS 300 FUNCTION NET ISSUES

The first thing that you will notice about the PS 300 function net language is the large number of vertex types -- or functions -- which have been defined. Some of them are not absolutely necessary but have been provided as a convenience, to permit more direct implementations of a desired solution. The actions which the functions perform are specialized to allow the interactive controls to modify the model for viewing.

It is not a general-purpose programming language, so some of the issues which have been previously presented in this document are not directly applicable to PS 300 programming. **They are, however, all relevant.**

Conjunctive and disjunctive input and output rules were presented in the guise of a decision structure which was hopefully similar to the types of program decisions made in sequential programming languages. Several PS 300 functions have disjunctive structure and all of the issues which apply to decision structures apply to them, even though their use may not be in a conventional IF-THEN-ELSE decision.

Often, there are standard values for certain types of graphical transformations. The PS 300 functions which are likely to incorporate these "default values" allow the programmer to specify the default in a particularly easy manner. In such cases the input port, if left disconnected, will always have the default input value. This is an entirely safe way to incorporate default constants.

Of course if the programmer wants to use a special value, then simply by using the appropriate input port, the new values can be specified. The programmer should not confuse this type of default constant specification with the constant queues type of arc or input port. Input ports which are labeled with a C can be thought of as being driven by an arc, whose queue length is 1.

Constant queue inputs, labeled C in the manual and hereafter referred to as C-inputs, are used by many of the PS 300 functions. C-inputs are a potential source of problems to the PS 300 programmer. The C-input defines the equivalent of a token register at the C-input port. As such the token which it contains is not removed when the vertex function fires. This is not a problem, as such a mechanism is similar to the well-behaved constant mechanism discussed previously.

The problem arises because C-input tokens can be overwritten at anytime, simply as the result of another token arriving at the C-input port. This clearly violates the persistence principle. Without persistent tokens, remember, some FGN programs may not be well-behaved.

For example, if a C-input value is sent to a function and is not used before another value arrives and overwrites it, then the effect is the same as if the first token was never generated at all. In cases where every generated value is supposed to be used, this disappearance would clearly be contrary to the programmer's intent.

Fortunately, there is a solution to the problem. This involves creating a backsignal from the user of the C-input token to the producer of the token which indicates that the value has been used. This "used" signal can then be part of the firing condition which will allow a subsequent value to be generated.

In other cases, the C-inputs cause no problem -- the program merely wants the latest value and does not care about the intermediate values of that type. Such an example might be the position of a dial. The program may fire when another dial is moved and the latest value of a second dial is then used to produce a new view on the screen.

The programmer must always keep in mind which of these two kinds of use is desired when using vertices with C-inputs and then use the proper technique to ensure that the desired behavior is created in the program structure.

Currently when an error occurs the PS 300 just stops, and prints out an error message indicating the type of error and the function type which was being executed when the error condition was observed.

Unfortunately the programmer cannot conveniently examine the input tokens which caused the error. Instead, the programmer must examine the program structure in an attempt to visualize how that type of error could have occurred.

Once the program has been corrected, the new net must be reloaded and restarted for a subsequent run. There is currently no provision for incremental error traps and restarts. This unfortunately implies that the program must be run at least one time for every error it contains. This primitive error facility and lack of a runtime debugger implies that it is paramount that program creation be done properly! This sentence should probably be reread a few thousand times.

The most common error is a data token-type mismatch. In the definition of the PS 300 functions, arcs and therefore input ports contain type information. Type checking is done at runtime, and if a non-conformable set of firing set types is observed then BOOM an error condition occurs. The PS 300 function net language can be viewed as a type-free language.

Some programmers have argued for years against the inflexibilities of strongly typed languages while others have argued in favor of strongly typed languages because they are easier to use to create correct programs. Both sides are valid.

It is important to realize that the lack of strong typing with the PS 300 means the compiler cannot predict runtime type-mismatch errors. The programmer should analyze all of the paths in the program graph to insure that consistent typed tokens are provided in all possible cases. This is especially true when certain conditions are present. Types must be conformable under all possible disjunctive output rules if error-free operation is to be the result.

You can make initial markings with PS 300 function graphs. After the program graph is loaded, tokens can be sent to any function input to create the initial marking. The programmer must do this with great caution. If a token which should be an initial marking causes a vertex to fire then the net will be ACTIVE. Usually this causes more firings and so on.

In general, initial markings should not cause any vertex to fire. This is a good practice since it is easier to know what the initial configuration is if nothing changes. It is possible to have vertices which are always fireable, and if this is the case then similar caution must be exercised.

In addition there are a number of functions (the CLOCK function is an example) that will fire at a real-time synchronous rate. This is a violation of the usual data-driven self-timed behavior. Any time such a function is used, it must be used in a manner that insures that the rate and nature of token production will be handled by receiving functions. Any time these real-time tokens go to C-inputs, special attention must be given to whether or not the possible lack of persistence will cause a problem.

A final point: users of the PS 300 specify their graph programs as textual descriptions. This is not a problem, but you are advised once again to draw pictures if you are wondering what is going on. Graphs usually aid your intuition!

5.3 CLASSIFICATION OF PS 300 FUNCTIONS

This section gives a function-by-function analysis in the form of tables. Four classes of functions are defined:

- C.C -- vertices with conjunctive input and output rules. These are typically safe as long as types match and the proper discipline is exercised for constant inputs.

- C.D -- vertices with conjunctive input rules and disjunctive output rules. These vertices need to be embedded in a symmetric decision structure, types must match, and constant inputs must be used properly.

- D.C -- vertices with disjunctive input rules and conjunctive output rules. These vertices need to be embedded in a symmetric decision structure, types must match, and C-inputs must be used properly.

- Sinks -- these functions fire in the normal data-driven way but produce no output which is returned back to the function net program. In some sense these arcs can be viewed as program outputs; behavior is rather simple to control.

  The only problem is that the lack of an output means that if data structure updates can happen concurrently then they must be sequenced if there is a necessary order. This sequencing will appear to be done randomly, so if the intent is that the order matters, then vertices in this class must be sequenced by a directed path through the operations which are to be ordered.

5.3.1 C.C FUNCTIONS

Function	Inputs:Outputs	Cautions
F:Ceiling	1:1	
F:Fix	1:1	
F:Float	1:1	
F:Print	1:1	
F:Round	1:1	
F:Vec	2:1	
F:CVec	2:1	C input discipline required
F:VecC	2:1	C input discipline required
F:XVector	1:1	
F:YVector	1:1	
F:ZVector	1:1	
F:Ceiling	1:1	
F:Fix	1:1	
F:Float	1:1	
F:Add	2:1	Input types must conform
F:AddC	2:1	Input types must conform C input discipline required
F:And	2:1	
F:AndC	2:1	C input discipline required
F:Div	2:1	Input types must conform
F:DivC	2:1	C input discipline required Input types must conform
F:CDiv	2:1	C input discipline required Input types must conform
F:Average	2:2	
F:Mod	2:1	

Function	Inputs:Outputs	Cautions
F:ModC	2:1	C input discipline required
F:Mul	2:1	Input types must conform
F:MulC	2:1	C input discipline required Input types must conform
F:Not	1:1	
F:Or	2:1	
F:OrC	2:1	C input discipline required
F:SinCos	1:2	
F:Sub	2:1	Input types must conform
F:CSub	2:1	C input discipline required Input types must conform
F:SubC	2:1	C input discipline required Input types must conform
F:Xor	2:1	
F:XorC	2:1	C input discipline required
F:Eq	2:1	Input types must conform
F:EqC	2:1	C input discipline required Input types must conform
F:Ge	2:1	Input types must conform
F:CGe	2:1	C input discipline required Input types must conform
F:GeC	2:1	C input discipline required Input types must conform
F:Gt	2:1	Input types must conform
F:CGt	2:1	C input discipline required Input types must conform
F:GtC	2:1	C input discipline required Input types must conform
F:Le	2:1	Input types must conform

Function	Inputs:Outputs	Cautions
F:CLe	2:1	C input discipline required Input types must conform
F:LeC	2:1	C input discipline required Input types must conform
F:Lt	2:1	Input types must conform
F:CLt	2:1	C input discipline required Input types must conform
F:LtC	2:1	C input discipline required Input types must conform
F:Fetch	2:1	C input discipline required
F:Neq	2:1	Input types must conform
F:NeqC	2:1	C input discipline required Input types must conform
F:Concatenate	2:1	
F:CConcatenate	2:1	C input discipline required
F:ConcatenateC	2:1	C input discipline required
F:Delta	2:1	
F:Limit	2:1	
F:CXRotate	3:2	C input discipline required
F:CYRotate	3:2	C input discipline required
F:CZRotate	3:2	C input discipline required
F:Scale	1:1	
F:XRotate	1:1	
F:YRotate	1:1	
F:ZRotate	1:1	
F:CRotate	1:1	
F:CScale	1:1	

Function	Inputs:Outputs	Cautions
F:FOV	4:1	C input discipline required
F:Lookat	3:1	C input discipline required
F:Lookfrom	3:1	C input discipline required
F:Window	7:1	C input discipline required
F:ClcSeconds	6:1	C input discipline required Always fireable real time behavior possible
F:ClFrames	6:1	C input discipline required Always fireable real time behavior possible
F:ClTicks	6:1	C input discipline required Always fireable real time behavior possible
F:Constant	2:1	C input discipline required
F:Fetch	2:1	C input discipline required
F:NOP	1:1	
F:Pickinfo	2:2	C input discipline required
F:Positionline	2:1	C input discipline required
F:Dials	1:8	Always fireable real time behavior possible
F:Keys	0:1	Data source, fires on key hit
Pick	1:2	
TabletIn	3:3	C input discipline required
Errors	1:4	
Memory_Alert	3:1	
Memory_Monitor	3:3	
Message_Display	1:1	
F:Dscale	5:2	C input discipline required

The following functions available with the P5 version of the PS 300 Runtime Firmware are also C.C.

Function	Inputs:Outputs	Cautions
F:NE	2:1	
F:NEC	2:1	C input discipline required
Lineeditor	3:6	C input discipline required
F:DXRotate	3:2	C input discipline required
F:DYRotate	3:2	C input discipline required
F:DZRotate	3:2	C input discipline required
F:Parts	1:4	
F:SqRoot	1:1	
F:XFormData	5:1	C input discipline required
Charmask	2:1	C input discipline required
F:Charconvert	2:1	C input discipline required
F:Color	2:1	
F:Matrix2	2:1	
F:Matrix 3	3:1	
F:Matrix 4	4:1	

5.3.2 C.D Functions

All of these functions should be used in symmetric decision type network structure if streamed tokens can occur. If mutually exclusive decision branches are used then this restriction can be relaxed since the marking will not create races in a mutually exclusive environment. However, even if the well-ordering is thus insured, the programmer must still take care to create live nets.

Function	Inputs:Outputs	Cautions
F:Components	1:5	Unused arcs depend on input type
F:RangeSelect	3:3	Input types must conform 3rd output not sent if not in range
F:Select	2:1	C input discipline required

Function	Inputs:Outputs	Cautions
F:Split	2:4	C input discipline required 2 outputs not sent if no match
F:Switch	2:20	
F:CSwitch	2:20	C input discipline required
F:SwitchC	2:20	C input discipline required
F:Edge_Detect	2:2	C input discipline required This function is simply hard to use as it may produce no outputs for certain firings
Keyboard	1:2	Output used depends on input type

The following functions available with the P5 version of the PS 300 Graphics Firmware are also C.D.

Function	Inputs:Outputs	Cautions
F:AtScale	3:1	C input discipline required
F:Accumulate	6:1	C input discipline required
F:CBRoute	2:1	C input discipline required
F:Broute	2:2	
F:BrouteC	2:2	C input discipline required
F:Limit	3:3	C input discipline required

5.3.3 D.C Functions

All of these functions should be used in symmetric decision structures if streamed tokens can occur. If mutually exclusive decision branches are used then this restriction can be relaxed since the marking will not create races in a mutually exclusive environment. However even if the well ordering is thus insured, the programmer must still take care to create live nets.

Function	Inputs:Outputs	Cautions
F:Boolean_Choose	3:1	C input discipline required
F:Choose	20:1	C input discipline required

Function	Inputs:Outputs	Cautions
F:Matrix	4:1	Input type specifies number of inputs used
F:Timeout	2:1	Real Time Firing whether other input is used Possible disjunctive output also Be careful with this one ALWAYS

The following functions available with the P5 version of the PS 300 Runtime Firmware are also D.C.

Function	Inputs:Outputs	Cautions
F:CMul	2:1	C input discipline required
F:Inputs_choose	N:1	C input discipline required

5.3.4 Sinks

The sink nodes are not necessarily a problem, however certain sequencing constraints, as mentioned previously, may require care in their use.

Sink Nodes:

All 8 DLabel functions (C input discipline required).

All 8 DSet functions (C input discipline required).

All 12 Flabel functions (C input discipline required).

Hostout

Set Conditional_Bit

Set Level_of_Detail

Set Rate

Viewport

Matrix_4x3

Look At

Look From

Sink Node (cont.)

Matrix_4x4

Window

Eye Back

Field_of_View

Set Displays

Set Depth_Clipping

Vector_List

Characters

Matrix_3x3

Rotate

Scale

Matrix_2x2

Character Size

Set Picking

Set Pick Location

ANNOTATED BIBLIOGRAPHY

The following is a list of references for the serious data-driven language student. The format is self-explanatory, with bibliographic information indented below a brief comment describing the publication.

In the following MS thesis, an example of a particular FGN language is used to do a sound processing application.

Author="J. A. Stanek",
School="University of Utah",
Month="September",
Date="September 1979",
Department="Computer Science",
Year="1979",
Title="Exploration of Concurrent Digital Sound Synthesis on a Prototype Data-Driven Machine"

The following report describes a Lisp like FGN language.

Author="R. M. Keller, B. Jayaraman, D. Rose, G. Lindstrom",
Title="FGL - Function Graph Language",
Number="AMPS Technical Memorandum #1",
Institution="University of Utah, Computer Science Department",
Date="July 1980",
Year="1980",

The following discusses semantic issues of a particular FGN model.

Author="R. M. Keller",
Title="Semantics and Applications of Function Graphs",
Number="UUCS-80-112",
Institution="University of Utah, Computer Science Department",
Date="October 1980",
Year="1980",

This report discusses a special purpose machine which was built, and which used an FGN language as its machine language.

Author="A. L. Davis",
Title="The Architecture of DDM1: A Recursively Structured
Data-Driven Machine",
Institution=" University of Utah, Computer Science Dept.",
Date="October 1977",
Number="UUCS-77-113",
Year="1977"

This report discusses the machine language of the previously cited machine.

Author="A. L. Davis",
Title="Data-Driven Nets: A Maximally Concurrent, Procedural,
Parallel Process Representation for Distributed Control Systems",
Institution="University of Utah, Computer Science Dept.",
Date="July 1978",
Number="UUCS-78-108",
Year="1978"

This PhD thesis was one of earliest reference to data-driven languages.

Author="D. A. Adams",
Institution="Stanford University, Computer Science Dept.",
Title="A computation model with data flow sequencing",
Date="December 1968",
Year="1968",
Number="CS117",

This article discusses general data-driven issues.

Author="J. B. Dennis",
Title="Programming generality, parallelism, and computer
Architecture",
Booktitle="Proceedings IFIPS Congress",
Organization="IFIPS",
Publisher="North Holland",
Year="1969",
Pages="484-492",

This report discusses a particular FGN language.

Author="K. P. Gostelow",
Title="Flow of Control, Resource Allocation, and the Proper Termination
Of Programs",
Institution="UCLA Computer Science Dept.",
Number="UCLA-ENG-71790",
Date="December 1971"

This paper discusses a particular low level FGN language.

Author="J. B. Dennis",
Title="First version of a data flow procedure language",
Date="September 1974",
Year="1974",
BookTitle="Lecture Notes in Computer Science",
Organization="SPRINGER-VERLAG",
Pages="362-376",
Volume="19",
Editor="B. Robinet",

This paper describes a high-level FGN language.

Author="W. B. Ackerman, J. B. Dennis",
Title="VAL - A Value-Oriented Algorithmic Language Preliminary
Reference Manual",
Institution="MIT, Computer Science Department",
Number="LCS/TR-218",
Year="1979",
Date="June 1979",

This article discusses general network issues.

Author="T. Agerwala, M. Flynn",
Title="Comments on capabilities, limitations, and correctness of
Petri Nets",
BookTitle="Proc. First Annual Symposium on Computer Architecture",
Organization="IEEE",
Year="1973",
Pages="81-86",
Date="December 1973",

This paper is a theoretical treatment of net languages.

Author="M. Hack",
Title="Petri net languages",
Institution="MIT Laboratory for Computer Science",
Year="1976",
Number="161",
Date="June 1976",

This paper is an excellent net modeling survey.

Author="J. L. Peterson",
Title="Petri Nets",
Journal="Computing Surveys",
Volume="9",
Year="1977",
Number="3",
Date="September 1977",
Pages="223-252",

Petri in some sense was the father of the network ideas. This paper is an excellent introduction to the basic theory.

Author="C. A. Petri",
Title="General Net Theory",
Organization="MIT Project MAC",
Booktitle="Conference on Petri Nets and Related Methods",
Date="July 1975",
Pages="26-41",
Year="1975",

More from the father.

Author="C. A. Petri",
Title="Fundamentals of a theory of asynchronous information flow",
Booktitle="Information Processing 62",
Publisher="North Holland",
Pages="386-391",
Year="1962",
Organization="IFIPS",

This paper is an early theory of nets treatise.

Author="R. M. Karp, R. E. Miller",
Title="Parallel program schemata",
Journal="Journal of Computing and System Sciences",
Volume="3",
Number="2",
Year="1969",
Pages="147-195",
Date="May 1969",

This work represents a thorough mathematical treatment of net ideas.

Author="A. Holt, F. Commoner",
Title="Events and Conditions",
Booktitle="Record of the Project MAC conference on concurrent
Systems and parallel computation",
Organization="MIT Project MAC",
Pages="3-52",
Year="1970",

This is required reading for people trying to break away from the bounds of totally-ordered thinking.

Author="J. Backus",
Title="Can programming be liberated from the von Neumann style?
A functional style and its algebra of programs",
Journal="CACM",
Volume="21",
Number="8",
Pages="613-641",
Date="August 1978",
Year="1978",

Another high level FGN language.

Author="Arvind, K. P. Gostelow, W. Plouffe",
Title="The Id Report: An Asynchronous Programming Language and
Computing Machine",
Institution="Univ. Calif. Irvine Comp. Sci. Dept.",
Date="May 1978",
Number="114A",
Year="1978",

A PhD thesis considered by many to be the beginning of modern data-driven thinking.

Author="J. D. Rodriguez",
Title="A Graph Model for Parallel Computation",
Institution="MIT Project MAC",
Number="TR-64",
Date="September 1969",
Year="1969",

An excellent MS thesis discussing the nature of token streams.

Author="K. S. Weng",
Title="Stream-Oriented Computation in Recursive Data-Flow Schemas",
Institution="MIT LCS",
Year="1975",
Date="October 1975",
Number="MIT/LCS/TM-68",

A seminal article on the theory of partially ordered program issues.

Author="D. Scott",
Title="Data types as lattices",
Month="September",
Year="1976",
Date="September 1976",
Journal="SIAM J. Comput.",
Number="3",
Volume="5",
Pages="522-587"

A chapter of the VLSI bible, dedicated to the issues of self-timed systems thinking.

Author="C. L. Seitz",
Fullauthor="C. L. Seitz",
Title="System Timing",
Booktitle="Introduction to VLSI Systems, Chapter 7",
Publisher="McGraw-Hill",
Year="1979",

Clearly there is more but if you understand these references, you won't need to read the others.

ABOUT THE AUTHOR

Alan L. Davis is an associate professor of computer science at the University of Utah. His current research interests include distributed architecture, graphically concurrent programming languages, parallel program schemata, device integration, asynchronous circuits, and self-timed systems. He has been a National Academy of Science exchange visitor and a visiting scholar in the Soviet Union, as well as a guest research fellow at the Gesellschaft fuer Matematik und Datenverarbeitung in West Germany.

Davis received a BS degree in electrical engineering from MIT in 1969 and a PhD in computer science from the University of Utah in 1972.