Design VC1000 Programming Manual

Contents

Introduction 1
Scope of Manual 2
Drive Overview 2
User Interface 2
Loading and Saving Programs 3

Basic Programming Principles 3
System Variables, User Variables, and Constants 4
System Variables 4
User Variables 4
Constants 5
Integer Math 5
More About States 5
Labels 5
Actions 5
Transitions 5

More Specifics About Working With The Drive System 6
Selecting, Editing, and Adding a State 6
Selecting, Editing, and Adding an Action 6
Selecting, Editing, and Adding a Transition 7
Programming Standards 8

Saving and Retrieving Program Files to Disk 8
Retrieve and Upload a Program Into a Drive 9
Download and Save a Program From a Drive 9

Example—Entering an Application Program 10

Appendix A—The Start Screen 12

Appendix B—Table of Possible Actions 13

Appendix C—Table of Transitions 15

W7988/IL

Figure 1. Design VC1000 Servo Drive

Appendix D—Table of System Variables 17

Appendix E—Useful Formulas for the Design VC1000 19
Derivative Gain 19
Proportional Gain 19
Integral Gain 19
Low Pass Filter 19
Current 20
Velocity 20
Position Counts 20

Appendix F—Example Program 21

Introduction

The Design VC1000 servo drive is preprogrammed at the factory. Normally,you will not need to reprogram

the servo drive. If you wish to reprogram the servo drive, however, this manual provides the necessary information.

Scope of Manual

This programming manual provides information for setting up and programming the Design VC1000 servo drive (figure 1). You should be familiar with basic programming principles, and understand the desired end result. The Design VC1000 servo drive is a “state machine”, with some unique programming requirements. You do not need to be familiar with “state machine” programming to use this manual. Refer to separate instruction manuals for information about wiring and installation information.

Only personnel qualified through training or experience should install, operate, and maintain a Design VC1000 servo drive. If you have any questions about these instructions, contact your Fisher Controls sales office before proceeding.

Drive Overview

The Design VC1000 servo drive is a digital drive designed to run Type 330SA servo actuators. The actuator contains a brushless DC servo motor, which means there are no motor brushes to wear out. The motor commutation is done electronically by the servo drive, using the resolver inside the actuator housing to provide position feedback to the drive.

The servo drive is microprocessor-based, and therefore programmable. There are two programs running in the drive. The first is the operating system which provides for all the input and output functions, calculates the pulse width modulation to the actuator, and performs all the internal operations of the servo drive. This program is stored in ROM chips inside the drive, and is generally known as firmware. When the drive powers up, the firmware is transferred into RAM and executed from there to improve execution speed.

This programming manual focuses on the second program, which is the application, or user programming. This programming enables the drive to perform the unique functions that you require. The program is stored in nonvolatile “flash” memory, which does not require any battery to maintain the program. On power up, this program is transferred into RAM and executed from there to improve speed.

User Interface

This programming manual applies to the drive and user interface in Design VC1000 servo drives delivered since the beginning of December 1999.

To access the drive interface, connect a PC to the drive’s serial interface. A cable with a null modem, or a null modem cable is required to make the connection. The PC must be running a terminal program, such as MicroSoft’s HyperTerminal, which is bundled and distributed with Windows95. The communications protocol is 9600 baud, 8 bits, 1 start bit, no parity.

When power is applied, one of three things will happen, depending on the servo drive’s state when it was shut down.

1. The PC will display the “start screen” shown in figure 2. This must be displayed to access and modify the drive programming, so this is the desired screen display. A copy of this screen is also found in *Appendix A*.

2. If the drive was set to run in the “auto” mode, the drive application program is running and must be stopped. Press the `<Tab>` key to stop the drive program and the “start screen” in step 1 should be displayed.

3. If the drive was left in the “Expert” prompt mode there may be nothing displayed except a flashing cursor. Press the `<Tab>` key to stop the drive program (just in case it was running) and type `sy pr no <enter>` to put the drive into the “Novice” prompt mode. This turns full prompting on, and the “start screen” in step 1 should be displayed.

The “start screen,” should now be displayed. This is called the start screen because everything is accessed through this screen. For example, typing `program display states <enter>` will cause the drive to display the machine states programmed into the drive. Actually, only the first two letters of the command are required. In the previous example, `pr di st <enter>` will also cause the drive to display the machine states programmed into the drive. The command `sy pr ex <enter>` puts the drive into the “expert prompt” mode, which turns off the above menu and limits the prompts displayed. The command `sy pr no <enter>` puts the drive into the “novice prompt” mode, turning the

```
help         (topic)
program      select|add|edit| (state|action|transition) #index
program      delete|display (state|action|transition) #index
program      upload|download|save|load|new|run
variable     name|#index|?
system       comm|prompt|axis|screen|echo (value)
system       io|mu|line|carriage|target|auto (value)
system       save|load|clear
drive #n (flimit|rlimit) (value)
drive #n (current|velocity|position)
drive #n (external|internal|reference|stop reference)
drive #n (master|slaved|user|emulate stepper)
drive #n (enable|disable|upload|download|save|load|clear)
quit
```

Figure 2. The Start Screen

above menu and full prompting on. From this point forward, this manual will refer only to two letter commands in the text and examples.

It is important to know that any time you make an error in typing, use the `<escape>` key to back out and then retype the line. Using the `<backspace>` key appears to work, but actually the line is being built with backspace characters embedded in it, which will create errors.

Also, several of the menu items refer to a `drive #n`. The Design VC1000 is a single axis controller, so the drive will always be drive #1. The multi-axis capability was kept in the firmware, anticipating future development possibilities.

Loading and Saving Programs

The memory management feature requires some instruction to use properly. As mentioned previously, the application program is stored in flash memory, but executed in RAM. A program is loaded into RAM by typing `pr lo <enter>`. Typing `pr sa <enter>` will save it back into the flash memory. If you edit the program, it is *changed only in RAM*. If you cycle power to the drive off and on again, the program changes will be lost. **If you want the changes saved, you must type `pr sa <enter>` to write the program into the flash memory before turning the power off.**

Similarly, if you make changes in the drive or the system set up, the `dr #1 sa <enter>`, and the `sy sa <enter>` commands must be executed to save the changes.

If the drive is in the “auto” mode, it will automatically load the program out of the flash memory and begin executing it. The auto mode is set by typing `sy au on <enter>`, and it is turned off by typing `sy au of <enter>`. Typing `sy au ? <enter>` will cause the drive to display whether the auto mode is turned on or off. If the auto mode is on and the application program is running, pressing the TAB key will stop the running program.

Loading and saving application files to disk will be addressed later in this manual.

Basic Programming Principles

The servo drive is a “state” machine. The program flows from one state to another, as directed by the states as they are executed. Each state has a series of actions, which define the function of the state. Each state has at least one transition statement, which directs the program flow. The states are like a series of subroutines which start with a `LABEL:` and end with a `GOTO` statement.

The action statements are executed only once, so loops internal to the state are not permitted. The transition can do conditional branching, so the program flow can be directed into one of several paths. Execution loops external to a state can be formed by having a state’s transition section direct flow to a second state, which then directs flow back to the prior state. Useless infinite loops can be formed this way, so care must be used to provide a proper exit from the loop.

Also, it is possible for the transition section of a state to loop upon itself. This allows the program to execute the actions of a state, but not proceed until the transition is met. For example, the valve program

initializes the drive and then disables it. The program then loops on the transition of the state “disable” until the enable drive command is true. Again, endless loops can be formed this way, so caution must be exercised.

System Variables, User Variables and Constants

The servo drive programming uses two types of variables; system variables and user variables. All variables are global in scope, that is, any variable can be used anywhere, and if it is changed in one state, it is changed for all states.

System Variables

The system variables are used to send data to the hardware and to monitor what is happening in the drive. For example, the variable `requested pos.(#1)` holds the position the drive is following. Changing this variable causes the drive to move the valve plug to a different position. Similarly, the variable `actual pos.(#1)` holds the value representing the actual, or current, valve position. The value can be monitored by the application programming to determine if the valve plug is out of position, and by how much.

The function and definition of the system variables are predetermined by the drive hardware and firmware. Although they are accessible to the application program, their meaning cannot be changed. Following is a brief list of several of the system variables. (A complete list of the system variables is found in *Appendix D*.)

`requested pos.(#1)` The target position for the valve when the drive is in position mode. The value is the number of encoder counts from the reference (zero) position.

`requested vel.(#1)` The target velocity for the valve plug when the drive is in velocity mode. The value is in encoder counts per time period. Currently the time period is one millisecond.

requested cur(#1) The target current when the drive is in the current mode. The current determines the torque, which determines the force. The value is a number with 0 representing no current and 32678 representing the maximum continous current the drive can produce.

`actual pos.(#1)` The actual plug position, which will be different than he requested position if the drive is moving the plug to a new position.

`actual vel.(#1)` The actual valve plug velocity, which will be different than the requested velocity if the plug is accelerating or decelerating.

`Drive Current Command` The actual current command to the actuator. This is useful during the valve plug homing and calibration procedure. The valve is closed in velocity mode. If the drive current command suddenly increases, it can be assumed the valve plug is on the valve seat.

`prop. gain(#1)` This is the proportional gain, a tuning parameter. See *Appendix E* for details on its use.

`deriv. gain(#1)` This is the derivative gain, a tuning parameter. See *Appendix E* for details on its use.

`int. gain(#1)` This is the integral gain, a tuning parameter. See *Appendix E* for details on its use.

`feed fwd. gain(#1)` This is the feed forward gain, a tuning parameter. See *Appendix E* for details on its use.

`gain scale(#1)` This is gain scale, a tuning parameter. See *Appendix E* for details on its use.

`integral limit(#1)` This is the low pass filter, a tuning parameter. See *Appendix E* for details on its use.

`Analog Position Input` This is the input for the 4 to 20 mA command signal from the control room.

`Auxiliary Analog Input` This is a secondary analog input. Its use is not specifically defined, but it can be used as a position feedback to eliminate the homing after a loss of power incident.

A complete list of the system variables is found in *Appendix D*.

User Variables

User variables are defined by the programmer when writing application, or user programs. When entering the actions for a state, the drive system prompts for the variable to use. The variable can be selected by either entering its index number or by typing its name. Whenever a new variable name is entered, the drive’s user interface prompts:

```
The variable does not exist
Press <ESC> if you do NOT want to add it
to the list
Press <ENTER> to add it to the list
```

Pressing `<ENTER>` adds the variable to the list of variables and assigns an index number. Typing `va ?`

from the start menu displays the variables in the list one screen full at a time. The new variable will have been added to the end of the list.

Constants

Constants used in the program are entered and handled by the user interface as if they were user variables. Whenever a constant is entered, the drive's user interface prompts:

```
The variable does not exist
Press <ESC> if you do NOT want to add it
to the list
Press <ENTER> to add it to the list
```

Pressing <ENTER> adds the constant to the list of variables and assigns an index number. Typing `va ?` from the start menu displays the variables in the list one screen full at a time. The new constant will have been added to the end of the list.

Integer Math

All variables are 32-bit integers. The range of numbers that can be expressed are ±2,147,483,648. All math is fixed decimal; there are no floating point math routines. When a division operation is performed the result is truncated, so all the values to the right of the decimal point are lost. For example, 10/6 = 1, not 1.66667 and not 2. For this reason, some numbers are prescaled by multiplying by 32 or 64. When the integer math is done, a number that is 32 or 64 times too big is obtained. That number is compensated for in the hardware and firmware, with an effective gain in some decimal points worth of precision.

Also, for this same reason, when math operations are performed, all the multiplication operations possible should be done before the division operations. This reduces the truncation error. For example, refer to the current formula below.

$$Current = \frac{(Percent\ Torque \times 32768)}{100}$$

If the desired torque is 60%, and the division is performed before the multiplication, the result is zero (60 / 100 =0; 0 * 32,768 =0) instead of 19,660 (60 * 32,768 =1,966,080; 1,966,080 / 100 = 19,660.8; which truncates to 19,660). This may be an extreme example, but it is a worthy example of what may happen if the order of math operations is not done carefully.

More About States

A state consists of three parts: label, actions, and transitions.

Labels

Every state must have a label. The label may be mixed upper and lower case and it may contain spaces. Examples of some typical state labels are: Init0, Set Position Mode, or Not in Foldback. Generally, the label should express in some way what the state does; it makes troubleshooting easier.

Actions

A state does not require any action in the action section. A state may be used solely to perform logical branching. If it does have actions, only one action can be performed per line of the program. For example, two variables can be multiplied together, or one subtracted from another, but subtracting and multiplying operations are not permitted in the same line.

The maximum number of actions in one state is 25 actions.

As mentioned previously, the action statements are executed only once, and then the program execution moves on to the transition portion of the state.

Transitions

At least one transition statement is required for a state. The transition statements allow conditional, or logical, tests and directs the program flow from one state to another, dependant upon the result of the test. For example, the transition of a state to another may depend on the magnitude of a variable, or whether or not an input line is high or low.

Unlike actions, the transition portion of a state may loop upon itself. For example, if a drive fault condition is detected, the program flow may be directed to a state that disables the drive and then loops upon itself until the fault is cleared.

Also, a transition may direct program flow back to the beginning of the state it is in. For example, a state labeled `Disable Drive` could have a transition statement `GOTO STATE Disable Drive NEXT`. Of course there must also be at least one other transition statement to provide an exit from this loop.

Also, the order of the transition statements is important. The program evaluates the transition statements in the order given. The first statement that evaluates as "True" will determine the next state

executed. For example, if the state Disable Drive has the transitions, GOTO STATE Disable Drive NEXT; and then the statement GOTO STATE Enable Drive NEXT, IF Faults Flag = 0; the program will never leave the state Disable Drive, even if the variable Faults Flag = 0.

The maximum number of transitions in one state is 25 transitions.

Another unique feature of the hardware and firmware is the sense of the digital inputs is inverted. The inputs are optically isolated and when current is flowing through the isolator's LED, the input is considered a logical low. For example, the transition statement `GOTO STATE Enable NEXT, IF INPUT Number2 ON GROUP 0 IS LOW` will not test true until there is current flowing in the opto-isolator's LED for digital input `DI2`.

More Specifics About Working With The Drive System

This portion of the VC1000 programming manual focuses on some of the specifics required to successfully enter, examine or edit an application program.

The section entitled *Selecting, Editing, and Adding a State*, lists the requirements to access the drive. It is assumed the monitor is displaying the "start screen".

CAUTION

Before entering or editing an application program, turn the Auto Run feature off to avoid creating a possible infinite loop.

A word of caution needs to be added here. Before entering or editing an application program, turn the Auto Run feature off. Typing `sy au <enter>` will cause the drive to display if the Auto Run feature is ON or OFF. The auto run is turned OFF by typing `sy au 0`.

The reason for doing this is that it is possible to put the drive into an infinite loop where the only way to exit the loop is to turn off the power, wait a few seconds and then turn the power back on. If the Auto Run feature is ON, and if the program had been saved, the defective program will reload and begin execution again. If the program goes into an infinite loop again, you will not be able to edit the program to fix the program problem. Disable the Auto Run until the program has been fully tested and known to be stable and error free. The Auto Run can be turned ON again by typing `sy au 1`.

Selecting, Editing, and Adding a State

If the statement `pr di st <enter>` is typed into the terminal keyboard, the drive will display a table showing the states that have been programmed into the drive. Every state is identified by an index number which is displayed to the left of the states. When the program is executed, the program begins with state #1 and the flow from there depends on the transition statements.

One of the indices has an asterisk (*) beside it, which marks the "selected" state. If any keyboard instructions are executed to display or edit state content, the selected state will be displayed or edited. For example, if the asterisk is beside state #5, typing `pr di ac <enter>` will cause the actions of state number 5 to be displayed on the terminal monitor. Typing `pr di tr <enter>` will cause the transitions of state number 5 to be displayed.

The state is selected by typing `pr se st #n <enter>`, where `n` represents the index of the desired state. Note that the # must be typed. For example, typing `pr se st #5 <enter>` will select state number 5.

Once selected, the state remains the selected state until another state is specifically selected. Running an application program resets the selected state to index #1.

A state can be deleted by typing `pr de st #n <enter>`, where `n` is the index for the state to be deleted. For example, `pr de st #5 <enter>` will delete state number 5.

A state can be added by typing `pr ad st #n`, where `n` is the index of the state to be added. You will be prompted to type in the name of the state being added. If `n` is less than the total number of states already defined, the new state will be inserted at the point indicated by the new state number. For example, `pr ad st #4 <enter>` will add a state number 4. If there were already 5 states defined, the old state number 4 will become state number 5 and the old state number 5 will become state number 6.

Selecting, Editing, and Adding an Action

If the statement `pr di ac <enter>` is typed into the terminal keyboard, the drive will display a table showing the actions that have been programmed into the drive for the selected state. Every action is

identified by an index number which is displayed to the left of the actions. When the state is executed, the program begins with action #1 and executes the remaining actions in the order displayed.

An action can be deleted by typing `pr de ac #n <enter>`, where `n` is the index for the action to be deleted. For example, `pr de ac #5 <enter>` will delete action number 5 of the selected state.

An action can be added by typing `pr ad ac #n <enter>`, where `n` is the index of the action to be added. If n is less than the total number of actions already defined, the new action will be inserted at the point indicated by the new action index number. For example, `pr ad ac #4 <enter>` will add an action number 4. If there were already 5 actions defined, the old action number 4 will become action number 5 and the old action number 5 will become action number 6.

After typing `pr ad ac #n <enter>`, a list of all possible actions will be listed, one screen at a time, with each possible action identified by a numerical index. You will be prompted to type in the index of the action on that list, which is being added to the selected state. For example, typing `pr ad ac #n <enter>` and when prompted typing `#3 <enter>`, you would then be prompted for the variable name you want the result in and then for the two variables you want to add together. Thus an action statement is built up by the drive operating system.

As you work with the program you will notice that the variables are also identified by a index number. Once the variable has been defined, the index number can be typed in its place to use the variable. The actual variable name will show in the program listing.

For an example of how a typical statement is entered, multiply the variable `temp` by `32768` and store the results in variable `requested pos.(#1)`. Furthermore, the action is being added as action # 8 in state #4. The index of the multiply action is #5. Also assume variable temp is variable index #66, constant 32768 is variable index #49 and requested pos.(#1) is variable index #1. The sequence of commands typed into the terminal keyboard is:

```
pr se st #4 <enter>
```

```
pr ad ac #8 <enter>
```

When prompted for the action, type `#5 <enter>`

When prompted for the result variable, type `#1 <enter>`

When prompted for the first variable or constant, type `#66 <enter>`

When prompted for the second variable or constant, type `#49 <enter>`

At this point the system returns to the start screen. If the command `pr di ac <enter>` is typed in, you will see that action number 4 is displayed as:

```
MULTIPLY: requested pos.(#1) = temp * 32768.
```

Editing an action works the same way. Type `pr ed ac #n <enter>`, where `n` is the index of the action you wish to edit. You will be prompted to enter the definition of the action just as when adding an action.

Remember, when altering the program actions, only the program in active RAM is being altered. If you wish to save the changes, type `pr sa <enter>`, to transfer the program into the flash memory, before shutting down the drive.

Selecting, Editing, and Adding a Transition

Working with transitions is similar to working with actions. If the statement `pr di tr <enter>` is typed into the terminal keyboard, the drive will display a table showing the transitions that have been programmed into the drive for the selected state. Every transition is identified by an index number which is displayed to the left of the transitions. When the state is executed, the program begins with transition #1 and executes the remaining transitions in the order displayed. If the conditions of the test in a line tests true, the transition action is taken. If no transition condition tests true, the last transition in the list will be taken.

A transition can be deleted by typing `pr de tr #n <enter>`, where `n` is the index for the transition to be deleted. For example, `pr de tr #5 <enter>` will delete transition number 5 of the selected state.

An transition can be added by typing `pr ad tr #n <enter>`, where `n` is the index of the transition to be added. If `n` is less than the total number of transitions already defined, the new transition will be inserted at the point indicated by the new transition index number. For example, `pr ad tr #4 <enter>` will add a transition number 4. If there were already 5 transitions defined, the old transition number 4 will become transition number 5 and the old transition number 5 will become transition number 6.

After typing `pr ad tr #n <enter>`, a list of all possible transitions will be listed, one screen at a time, with each possible transition identified by a numerical index. You will be prompted to type in the index of the transition on that list which is being added to the

selected state. For example, typing `pr ad tr #n <enter>` and when prompted typing `#3 <enter>`, you would then be prompted for the state name you want the program to go to. Thus a transition statement is built up by the drive operating system.

For an example of how a typical statement is entered, go to state `Seat Found` if the commanded current exceeds the seat current limit value. The index number for the transition `IS GREATER` is #31. The index number for `Seat Found` is #10. The commanded seat current is the variable `Drive Current Command` and its variable index is #46. The seat current limit is variable `DAC Seat Current` and its variable index is #81. Furthermore, the transition is being added as transition # 8 in state #9. The sequence of commands typed into the terminal keyboard is:

```
pr se st #9 <enter>

pr ad tr #8 <enter>
```

When prompted for the transition, type `#31 <enter>`

When prompted for the variable to test, type `#46 <enter>`

When prompted for the test variable or constant, type `#81 <enter>`

At this point the system returns to the start screen. If the command `pr di tr <enter>` is typed in, you will see that transition number 8 is displayed as:

```
GOTO STATE Seat Found IF Drive Current
Command > DAC Seat Current
```

Editing a transition works the same way. Type `pr ed tr #n <enter>`, where `n` is the index of the transition you wish to edit. You will be prompted to enter the definition of the transition just as when adding a transition.

Remember, when altering the program transitions, only the program in active RAM is being altered. If you wish to save the changes, type `pr sa <enter>`, to transfer the program into the flash memory, before shutting down the drive.

Programming Standards

There are very few programming standards that need to be adopted to improve readability and troubleshooting.

All the user variable parameters should be defined in the first state to be executed (state index #1), and the state should be labeled `Init0`. The parameters to define are the application specific information such as tuning parameters, stroke length, resolver counts per revolution, actuator lead-screw pitch, maximum actuator speed, and so forth. If the number of action statements required exceeds 25 statements, the last statement should be a comment indicating there are more user parameters in the next state. The concept here is to put all the variables and parameters that might be altered or adjusted in the field in one place.

All the user variables (variables with an index number greater than 45) must be defined or initialized at the beginning of the program. Use states labeled `Init1`, `Init2`, etc., to perform the initializations not done in `Init0`. Typically these will be system flags, and converting the user input variables from `Init0` into numbers the servo drive uses. Some of the useful conversion formulas are found in Appendix D.

The first line of `Init0` must be a comment line with the program name. The second line must be a comment with the name or initials of the person who wrote the program and the date it was written. The third line, must be a comment showing the Revision level, the initials of the person who revised it, and the date of the revision.

The fourth line must be a printed text line displaying the Fisher part number for the application program and revision level. It is also recommended a fifth line be added which will display a brief description of the program function.

For example, the first five lines of `Init0` might look like:

```
/* ACME Fuel Valve

/* DJW 12/15/99

/* Rev C, DJW 6/26/00

PRINT: 14B4002X012 Rev C

PRINT: ACME Fuel PN2201-38 Rev A
```

Saving and Retrieving Program Files to Disk

Application programs can be saved to disk or recalled from disk and loaded into the drive. The drive operating system does not have the capability to do this directly. An IBM compatible PC must be connected to the servo drive via a serial cable with a null modem, or a null modem cable. Most likely this can be the same computer used to program the drive

because the interconnection requirements are the same.

Retrieve and Upload a Program Into a Drive

Connect a PC to the drive's serial interface. Using a terminal program, get the drive running so the start screen is displayed. Quit the terminal program by closing the terminal program. **Do Not** type `qu <enter>` as this will stop the drive, and the drive must be running to perform the rest of this procedure.

Execute the program `VC1000a.exe` on the PC. This causes the PC to display a screen that looks like the start screen. At this point the PC is emulating many of the servo drive functions.

Set serial communication to port COM1: by typing `sy co #1 <enter>.`

Retrieve an application program from disk and load it into the PC by executing the commands `pr lo <enter>`. You will then be prompted:

```
Do you want to load the program?

Press <ESC> to cancel or

<Return>to begin loading:
```

Press `<enter>` and you will be prompted:

```
- Recalling program from a file

What file do you wish to read?:
```

Type in the file name, including the path if required. For example, to read a file called FISHER01 from drive A, type `A:FISHER01 <enter>` The program file is recalled from disk and loaded into the PC memory.

Execute the upload command by typing `pr up <enter>` and the program data is transferred to the servo drive.

Similarly, drive setup parameters can be recalled from disk and uploaded to the servo drive. Assuming the program `VC1000a.exe` is still running on the PC and the communications port is still COM1:, execute the command `dr #1 lo <enter>` to load the drive parameters into the PC's RAM. Execute the command `dr #1 up <enter>` to transfer the drive parameters into the servo drive. Notice that this command never asks for a file name or path. The file transferred is named `Drive1.sys` and is located in the same directory as the VC1000.exe program.

At this point, all the data uploaded is in the servo drive's RAM memory, but has not been permanently stored. To save the data, quit the program `VC1000.exe`. Restart the terminal program and the familiar start screen will be displayed. Save the program by typing `pr sa <enter>`. Save the system parameters by typing `sy sa <enter>`. Save the drive parameters by typing `dr #1 sa <enter>`.

You may wish to have both the terminal program and the VC1000.exe programs running on the PC at the same time, but it cannot be done. If both are running, they get confused with each other during the data transfers and either the transfer will not occur, or the data is corrupted during the transfer. Have only one program running at a time, as mentioned above.

Download and Save a Program From a Drive

Connect a PC to the drive's serial interface. Using a terminal program, get the drive running so the start screen is displayed. If the program to be saved onto disk is still in the flash memory, type `pr lo <enter>` to transfer the program into RAM. Type `sy lo <enter>` to transfer the system parameters into RAM, and type `dr #1 lo <enter>` to load the drive parameters into RAM.

Quit the terminal program by closing the terminal program. **Do Not** type `qu <enter>` as this will stop the drive, and the drive must be running to perform the rest of this procedure.

Execute the program `VC1000.exe` on the PC. This causes the PC to display a screen that looks like the start screen. At this point the PC is emulating many of the servo drive functions.

Set serial communication to port COM1: by typing `sy co #1 <enter>.`

Execute the program download command by typing `pr do <enter>` and the program data is transferred from the servo drive to the PC. Type `dr #1 do <enter>` to transfer the drive parameters from the drive to the PC.

The application program can now be saved to a disk by executing the commands `pr sa <enter>`. You will then be prompted:

```
Do you want to save the program?

Press <ESC> to cancel or

<Return>to begin saving:
```

Press `<enter>` and you will be prompted:

- saving program to a file

What file do you wish to use?:

Type in the file name, including the path if required. For example, to save a file called FISHER01 to drive A, type `A:FISHER01 <enter>` The program file is transferred from the PC memory onto the disk.

Similarly, drive setup parameters can be stored on disk. Execute the command `dr #1 do <enter>` to transfer the drive parameters into a disk file. Notice that this command never asks for a file name or path. The file transferred is named `Drive1.sys` and is located in the same directory as the Simacon.exe program.

Again, you may wish to have both the terminal program and the VC1000.exe programs running on the PC at the same time, but it cannot be done. If both are running, they get confused with each other during the data transfers and either the transfer will not occur, or the data is corrupted during the transfer. Have only one program running at a time, as mentioned above.

Example—Entering an Application Program

Following is an example of how to enter a drive program. The example program is found in *Appendix F*. Included is a state diagram to show the program flow and a description of the function of each program state. The program's function is to control a valve where the valve plug is pushed down to close the valve, which means the actuator extends to close the valve.

If starting with a new drive, without any programming loaded in it, hook up the personal computer that will be used to enter the program to the serial port of the computer and the serial port of the drive. Remember that the cable must be a null-modem type that switches pins 2 and 3 at one end of the cable, or a null modem must be used as part of the interconnection. (See the section entitled *The User Interface*). Connect a power cable to the "L1/DC+", "L2/DC–", and "ground" terminals, as appropriate for the power being used. An actuator does not need to be connected to the drive to program it, but a valve and actuator must be connected to the drive to test the programming.

A jumper must be placed between the "I/O Common" and the "24V Common" on the input connector. Another jumper must be connected between the "Isolated +24VDC" and "DI2" of the input connector. It must be possible to temporarily disconnect the second jumper, so some type of switch could be a part of this connection. In normal usage, the connection is closed to enable the drive. If the drive becomes disabled for any reason, the jumper connection must be broken and re-established to re-enable the drive.

Power up the PC and get the terminal program running. (See the section entitled *The User Interface*) Apply power to the servo drive. At this point the start screen should be seen on the PC's display. Turn off the auto run feature if it is on. Type `sy au 0 <enter>` to turn the auto run feature off. (See the section entitled *More Specifics About Working With the Drive System*.)

The program is entered by typing commands into the PC keyboard. The first step is to declare the states for the program. The states must be established first by adding a state and then naming it. Later, the actions and transitions are added to the state. For example, to add the state `Init0`, the command `pr ad st #1<enter>` is typed. The drive prompts:

```
State is ** New State **.
```

Press <ESC> to cancel or Enter the name of the state:

The name `Init0 <enter>` is typed in. The state is now declared and named, and ready for the actions and transitions to be defined. States can be added and edited in almost any order. A new state can be inserted between two existing states. Note that the program execution always begins with state number 1.

To add actions, select the desired state and then type the add actions command and the action index number. The drive will prompt you, depending on the action being added. For example, using the example program, let's say we want to add action #8 of state Init0. It is assumed actions 1 through 7 have already been added. The state is selected by typing `pr se st #1 <enter>`. Add action #8 by typing `pr ad ac #8`. The drive will display a list of actions (See *Appendix B*). The SET action is index #2 so following the servo drive prompting type `#2 <enter>`. The drive will display a list of variables and prompt for the index of the variable to assign the valve to. The variable has not been defined yet so type `Stroke <enter>`. The drive prompts:

```
The variable does not exist.
```

Press <ESC> if you do NOT want to add it to the list
Press <ENTER> to add it to the list:

Type `<enter>` again to establish the new variable name. The drive then prompts for the value to assign to the variable. The value to enter is the constant –1125 because the stroke length is 1.125 inches, and

the negative sign indicates the actuator must retract to open the valve. Type `-1125 <enter>` and the drive will prompt:

```
The variable does not exist.
```

Press <ESC> if you do NOT want to add it to the list
Press <ENTER> to add it to the list:

Type `<enter>` again to establish the new constant name. Remember, the drive treats variables and constants in the same way. (See the section entitled *System Variables, User Variables and Constants*)

To add transitions, select the desired state and then type the add transitions command and the transition index number. The drive will prompt you, depending on the transition being added. For example, using the example program, let's say we want to add transition #1 of state Init0. The state is selected by typing `pr se st #1 <enter>`. Add transition #1 by typing `pr ad tr #1`. The drive will display a list of transitions (See *Appendix C*). The "AS THE NEXT STATE" transition is index #3 so following the servo drive prompting type `#3 <enter>`. The drive will display a list of states and prompt for the index of the state the program flow is to continue to. Note that the state must have been previously declared for it to show up on the list. Continue to state Init1, which is index #2. At the prompt type `#2 <enter>`.

Using these techniques, the program is entered one statement at a time until the entire prgram has been entered. It is wise to save the program from time to time by typing pr sa <enter>. The data has only been entered in RAM and will be lost if the power is removed from the servo drive, unless the save command has been used. Once the program has been debugged and it is stable and functioning as expected, the auto run feature can be turned on by typing the command `sy au 1`. The final program can be saved to a floppy disk by following the method described in the section entitled *Saving and Retrieving Program Files to Disk*.

Appendix A—The Start Screen

```
help         (topic)
program      select|add|edit| (state|action|transition) #index
program      delete|display (state|action|transition) #index
program      upload|download|save|load|new|run
variable     name|#index|?
system       comm|prompt|axis|screen|echo (value)
system       io|mu|line|carriage|target|auto (value)
system       save|load|clear
drive #n (flimit|rlimit) (value)
drive #n (current|velocity|position)
drive #n (external|internal|reference|stop reference)
drive #n (master|slaved|user|emulate stepper)
drive #n (enable|disable|upload|download|save|load|clear)
quit
```

Appendix B—Table of Possible Actions

Following is a listing of all the possible actions that can be taken in a state. Each action includes a description of the action and an example programming statement. When programming the drive, the drive operating system prompts for the variables and constants needed and builds the statement. The example shows how the final statement would look.

Index	Programming Action	Description and Example
#1	COMMENT: string	Puts a comment in the programming text /* This is a comment
#2	SET: var = var | value	Assigns a value to a variable or constant. If the variable or constant is new, it defines the variable of constant SET: requested pos.(#1) = pos counts
#3	ADD: var = var | const + var | const	Adds two variables or constants and assigns the result to a variable. ADD: Foldback Timer = Foldback Timer + 1
#4	SUBTRACT: var = var | const – var | const	Subtracts one variable or constant from another variable or constant and assigns the result to a variable. SUBTRACT: temp = Input Signal Lost - 4000
#5	MULTIPLY: var = var | const * var | const	Multiples two variables or constants together and assigns the result to a constant. MULTIPLY: temp = temp * ADC Range
#6	DIVIDE: var = var | const / var | const	Divides one variable or constant by another variable or constant and assigns the result to a variable. DIVIDE: temp = Filtered Seat Current / 32768
#7	INCREMENT: var = var + 1	Increments a variable by 1 INCREMENT: temp
#8	DECREMENT: var = var - 1	Decreases a variable by 1 DECREMENT: temp
#9	NEGATE: var = -var	Assigns the negative value of a variable to the variable. NEGATE: temp = –temp
#10	AND: var = var | const AND var | const	Performs a logical AND of two variables or constants and assigns the result to a variable AND: temp = Homed AND Number1
#11	OR: var = var | const OR var | const	Performs a logical OR of two variables or constants and assigns the result to a variable OR: temp = Homed OR Number2
#12	NOT: var = NOT var	Performs a logical NOT of a variable and assigns the result to a variable NOT: temp = NOT Homed
#13	WAIT: n (MICROSECONDS)	Pauses program execution for n microseconds. WAIT: 200
#14	PRINT DECIMAL: var | const	Prints the decimal value of a variable or constant PRINT DECIMAL: MIN POS CNTS
#15	PRINT HEX: var | const	Prints the hexadecimal value of a variable or constant PRINT HEX: MIN POS CNTS
#16	PRINT TEXT: string | format	Prints a text string on a PC running a terminal program and which is connected to the drive's serial port. PRINT: Initializing… \n
#17	ENABLE DRIVE: drive #n	Turns on current to the servo actuator. ENABLE DRIVE: drive #1
#18	DISABLE DRIVE: drive #n	Turns off current to the servo actuator. DISABLE DRIVE: drive #1
#19	REFERENCE DRIVE: drive #n	
#20	STOP REFERENCING DRIVE: drive #n	
#21	POSITION MODE: drive #n	Puts the drive into position following mode. The drive will adjust the motor current and speed as needed to follow the value if the variable requested pos.(#1). POSITION MODE: drive #1
#22	Not Implemented. Reserved for future use.	
#23	Not Implemented. Reserved for future use.	
#24	VELOCITY MODE: drive #n	Puts the drive into a constant velocity mode to drive the actuator at a constant speed. Be sure to set the variable requested vel.(#1) to the desired velocity before putting the drive into velocity mode. VELOCITY MODE: drive #1

Index	Programming Action	Description and Example
#25	CURRENT MODE: drive #n	Puts the drive into a constant current mode to drive the actuator at a constant speed. Be sure to set the variable requested cur.(#1) to the desired current before putting the drive into current mode. The current determines the motor torque, which determines to output force, so the current mode is equivalent to a constant force mode. CURRENT MODE: drive #1
#26	Not Implemented. Reserved for future use.	
#27	SET OUTPUT HIGH: output var | const	Sets one digital output line high. SET OUTPUT HIGH: Output 2 of Group 0
#28	SET OUTPUT LOW: output var | const	Sets one digital output line low. SET OUTPUT LOW: Output 2 of Group 0
#29	SET ALL OUTPUTS: output var | const	Individually sets all digital output lines high or low, depending on a bit pattern. SET ALL OUTPUTS: Output Flags of Group 0
#30	READ ALL INPUTS: var from Group 0	Reads all the inputs of a digital input group and assigning the result to a variable. The drive has only one input group, which is Group 0 READ ALL INPUTS: INPUT Stroke from Group 0
#31	SET BIT PATTERN: var | const IN var	Sets a bit pattern, defined by a variable or constant, in a variable SET BIT PATTERN: SET BIT PATTERN OF 250 IN Foldback Current
#32	CLEAR BIT PATTERN: var | const IN var	Clears a bit pattern, defined by a variable or constant, in a variable. CLEAR BIT PATTERN: CLEAR BIT PATTERN OF 250 IN Foldback Current
#33	STOP PROGRAM:	Stops the program execution. Equivalent to pressing the <TAB> key on the terminal keyboard. STOP PROGRAM
#34	Not Implemented. Reserved for future use.	
#35	Not Implemented. Reserved for future use.	
#36	Not Implemented. Reserved for future use.	
#37	Not Implemented. Reserved for future use.	
#38	Not Implemented. Reserved for future use.	
#39	ENABLE FORWARD LIMIT SWITCH: drive #n	Activates the forward limit switch. It can be either active high or active low. ENABLE FORWARD LIMIT SWITCH (ACTIVE HIGH) FOR DRIVE #1
#40	DISABLE FORWARD LIMIT SWITCH: drive #n	Deactivates the forward limit switch. DISABLE FORWARD LIMIT SWITCH FOR DRIVE #1
#41	ENABLE REVERSE LIMIT SWITCH: drive #n	Activates the reverse limit switch. It can be either active high or active low. ENABLE REVERSE LIMIT SWITCH (ACTIVE HIGH) FOR DRIVE #1
#42	DISABLE REVERSE LIMIT SWITCH: drive #n	Deactivates the reverse limit switch. DISABLE REVERSE LIMIT SWITCH FOR DRIVE #1

Appendix C—Table of Transitions

Following is a list of all the possible state transitions. Each transition includes a description of the transition and an example programming statement. When programming the drive, the drive operating system prompts for the states, variables and constants needed and builds the statement. The example shows how the final statement would look.

Index	Programming Event	Description and Example
#1	COMMENT: string	Adds a comment to the transitions /* This is a comment
#2	RESCAN TRANSITIONS	Rescans the transition of a state. There must be another transition statement to provide and exit from the state. RESCAN TRANSITIONS
#3	AS THE NEXT STATE: State #n	Transfers to another state without any logical test. GOTO STATE Init2 NEXT
#4	RESCAN TRANSITIONS IF DRIVE IS IN POSITION: drive #n	Rescans the state transitions if the drive is in position. RESCAN TRANSITIONS IF DRIVE #1 IS IN POSITION
#5	IF DRIVE IS IN POSITION: State #n, drive #n	Transfers to another state if the drive is in position. GOTO STATE HoldPos NEXT, IF DRIVE #1 IS IN POSITION
#6	RESCAN TRANSITIONS IF DRIVE IS NOT IN POSITION: Drive #n	Rescans the state transitions if the drive is not in position. RESCAN TRANSITIONS IF DRIVE #1 IS NOT IN POSITION
#7	IF DRIVE IS NOT IN POSITION: State #n, drive #n	Transfers to another state if the drive is not in position. GOTO STATE Home2 NEXT, IF DRIVE #1 IS NOT IN POSITION
#8	RESCAN TRANSITIONS IF DRIVE IS ENABLED: drive #n	Rescans the transitions if the drive is enabled. RESCAN TRANSITIONS IF DRIVE IS ENABLED
#9	IF DRIVE IS ENABLED: State #n, drive #n	Transfers to another state if the drive is enabled. GOTO STATE Read Position NEXT, IF DRIVE #1 IS ENABLED
#10	RESCAN TRANSITIONS IF DRIVE IS DISABLED: drive #n	Rescans the transitions if the drive is disabled. RESCAN TRANSITIONS IF DRIVE IS DISABLED
#11	IF DRIVE IS DISABLED: State #n, drive #n	Transfers to another state if the drive is disabled. GOTO STATE Enable Drive NEXT, IF DRIVE #1 IS DISABLED
#12	RESCAN TRANSITIONS IF DRIVE HAS FAULTED: drive #n	Rescans transitions if the drive has faulted RESCAN TRANSITIONS IF DRIVE #1 HAS FAULTED
#13	IF DRIVE HAS FAULTED: State #n, drive #n	Transfers to another state if the drive has faulted GOTO STATE Fault Detected NEXT, IF DRIVE #1 HAS FAULTED
#14	RESCAN IF DRIVE HAS NOT FAULTED: Drive #n	Rescans transitions if the drive has not faulted RESCAN TRANSITIONS IF DRIVE #1 HAS NOT FAULTED
#15	IF DRIVE HAS NOT FAULTED: State #n, drive #n	Transfers to another state if the drive has no faults GOTO STATE Read Command NEXT, IF DRIVE #1 HAS NOT FAULTED
#16	RESCAN TRANSITIONS IF HIGH ON INPUT: RESCAN TRANSITIONS IF INPUT var | const ON GROUP 0 IS HIGH	Rescans the transitions if one of the digital inputs is high RESCAN TRANSITIONS IF INPUT Number2 ON GROUP 0 IS HIGH
#17	HIGH ON INPUT: GOTO state #n NEXT, IF INPUT var | const ON GROUP 0 IS HIGH	Transfers to another state if one of the digital inputs is high GOTO STATE Disable NEXT, IF INPUT Number2 ON GROUP 0 IS HIGH
#18	RESCAN TRANSITIONS IF LOW ON INPUT: RESCAN TRANSITIONS IF INPUT var | const ON GROUP 0 IS LOW	Rescans the transitions if one of the digital inputs is low RESCAN TRANSITIONS IF INPUT Number2 ON GROUP 0 IS LOW
#19	LOW ON INPUT: GOTO state #n NEXT, IF INPUT var | const ON GROUP 0 IS LOW	Transfers to another state if one of the digital inputs is low GOTO STATE Enable NEXT, IF INPUT Number2 ON GROUP 0 IS LOW
#20	RESCAN TRANSITIONS IF RISING EDGE ON INPUT: RESCAN TRANSITIONS IF THERE IS A RISING EDGE ON INPUT var | const ON GROUP 0	Rescan transitions of the state if the rising edge of a selected input is detected RESCAN TRANSITIONS IF THERE IS A RISING EDGE ON INPUT number1 ON GROUP 0
#21	RISING EDGE ON INPUT: GOTO STATE State #n NEXT, IF THERE IS A RISING EDGE ON INPUT var | const ON GROUP 0	Transfers to another state if the rising edge of a selected input is detected GOTO STATE Found Seat NEXT, IF THERE IS A RISING EDGE ON INPUT number1 ON GROUP 0
#22	RESCAN TRANSITIONS IF FALLING EDGE ON INPUT: RESCAN TRANSITIONS IF THERE IS A FALLING EDGE ON INPUT var | const ON GROUP 0	Rescan transitions of the state if the falling edge of a selected input is detected RESCAN TRANSITIONS IF THERE IS A FALLING EDGE ON INPUT number1 ON GROUP 0

<table>
<tr><th>Index</th><th>Programming Event</th><th>Description and Example</th></tr>
<tr><td>#23</td><td>FALLING EDGE ON INPUT:
GOTO STATE State #n NEXT, IF THERE IS A FALLING EDGE ON INPUT var | const ON GROUP 0</td><td>Transfers to another state if the falling edge of a selected input is detected
GOTO STATE Found Seat NEXT, IF THERE IS A FALLING EDGE ON INPUT number1 ON GROUP 0</td></tr>
<tr><td>#24</td><td>RESCAN IF INPUT GROUP EQUAL:
RESCAN TRANSITIONS IF GP Inputs OF GROUP 0 = var | const</td><td>Rescans transitions if the input bit pattern of group 0 equals another variable or constant
RESCAN TRANSITIONS IF GP Inputs OF GROUP 0 = Fault Flag</td></tr>
<tr><td>#25</td><td>IF INPUT GROUP EQUAL:
GOTO STATE State #n, IF GP Inputs OF GROUP 0 = var | const</td><td>Transfers to another state if the input bit pattern of group 0 equals another variable or constant
GOTO STATE Disable Drive, IF GP Inputs OF GROUP 0 = Reset Flag</td></tr>
<tr><td>#26</td><td>RESCAN TRANSITIONS IF LESS:
RESCAN TRANSITIONS IF var < var | const</td><td>Rescans transitions if a variable is less than another variable or constant
RESCAN TRANSITIONS IF temp < 0</td></tr>
<tr><td>#27</td><td>IS LESS:
GOTO STATE State #n, IF var < var | const</td><td>Transfers to another state if a variable is less than another variable or constant.
GOTO STATE Negate Current, IF temp < 0</td></tr>
<tr><td>#28</td><td>RESCAN TRANSITIONS IF EQUAL:
RESCAN TRANSITIONS IF var = var | const</td><td>Rescans transitions if a variable is equal to another variable or constant
RESCAN TRANSITIONS IF temp = 0</td></tr>
<tr><td>#29</td><td>IS EQUAL:
GOTO STATE State #n, IF var = var | const</td><td>Transfers to another state if a variable is equal to another variable or constant
GOTO STATE Read Position Reference, IF Homed = Number1</td></tr>
<tr><td>#30</td><td>RESCAN TRANSITIONS IF GREATER:
RESCAN TRANSITIONS IF var > var | const</td><td>Rescans transitions if a variable is greater than another variable or constant
RESCAN TRANSITIONS IF temp > 0</td></tr>
<tr><td>#31</td><td>IS GREATER:
GOTO STATE State #n IF var > var | const</td><td>Transfers to another state if a variable is greater than another variable or constant
GOTO STATE Found Seat, IF temp > DAC Seat Current</td></tr>
<tr><td>#32</td><td>RESCAN TRANSITIONS IF DRIVE IS RUNNING: Drive #n</td><td>Rescans the transitions of a state if the drive is running
RESCAN TRANSITIONS IF DRIVE #1 IS RUNNING</td></tr>
<tr><td>#33</td><td>IF DRIVE IS RUNNING: State #n, drive #n</td><td>Transfers to another state if the drive is running
GOTO STATE Read Pos Command NEXT, IF DRIVE #1 IS RUNNING</td></tr>
<tr><td>#34</td><td>RESCAN TRANSITIONS IF DRIVE IS STOPPED: State #n, drive #n</td><td>Rescans the transitions of a state if the drive is stopped
RESCAN TRANSITIONS IF DRIVE #1 IS STOPPED</td></tr>
<tr><td>#35</td><td>IF DRIVE IS STOPPED: State #n, drive #n</td><td>Transfers to another state if the drive is stopped
GOTO STATE Fault Handler NEXT, IF DRIVE #1 IS STOPPED</td></tr>
<tr><td>#36</td><td>RESCAN TRANSITIONS IF DRIVE IS REFERENCED: Drive #n</td><td>Rescans transitions of a state if the drive is referenced
RESCAN TRANSITIONS IF DRIVE #1 IS REFERENCED</td></tr>
<tr><td>#37</td><td>IF DRIVE IS REFERENCED: State #n, drive #n</td><td>Transfers to another state if the drive is referenced
GOTO Follow On NEXT, IF DRIVE #1 IS REFERENCED</td></tr>
<tr><td>#38</td><td>RESCAN TRANSITIONS IF DRIVE IS NOT REFERENCED: Drive #n</td><td>Rescans transitions of a state if the drive is not referenced
RESCAN TRANSITIONS IF DRIVE #1 IS NOT REFERENCED</td></tr>
<tr><td>#39</td><td>IF DRIVE IS NOT REFERENCED</td><td>Transfers to another state if the drive is referenced
GOTO Follow Off NEXT, IF DRIVE #1 IS NOT REFERENCED</td></tr>
<tr><td>#40</td><td>CARRIAGE RETURN INPUT FROM KEYBOARD:
GOTO STATE State #n NEXT, IF A CARIAGE RETURN IS DETECTED</td><td>Transfers to another state if a carriage return is typed into the terminal keyboard
GOTO STATE Initiate Valve Seat NEXT, IF A CARIAGE RETURN IS DETECTED</td></tr>
<tr><td>#41</td><td>CHARACTER INPUT FROM KEYBOARD:
GOTO STATE State #n, IF char IS INPUT FROM THE KEYBOARD</td><td>Transfers to another state if a specified character is entered from the terminal keyboard
GOTO STATE Initiate Valve Seat, IF s IS INPUT FROM THE KEYBOARD</td></tr>
<tr><td>#42</td><td>VALID NUMBER INPUT FROM KEYBOARD:
GOTO STATE State #n, IF A VALID NUMBER IS INPUT FROM THE KEYBOARD</td><td>Transfers to another state if any valid number is entered from the terminal keyboard
GOTO STATE Stop Program, IF A VALID NUMBER IS INPUT FROM THE KEYBOARD</td></tr>
<tr><td>#43</td><td>SOMETHING INPUT FROM KEYBOARD:
GOTO STATE State #n NEXT IF SOMETHING HAS BEEN ENTERED FROM THE KEYBOARD</td><td>Transfers to another state if anything has been entered from the terminal keyboard
GOTO STATE Stop Program NEXT IF SOMETHING HAS BEEN ENTERED FROM THE KEYBOARD</td></tr>
<tr><td>#44</td><td>RESCAN TRANSITIONS IF FORWARD LIMIT HIT:
Drive #n</td><td>Rescans the state’s transitions if the drive’s forward limit has been reached
RESCAN TRANSITIONS IF Drive #1’s FORWARD LIMIT WAS REACHED</td></tr>
<tr><td>#45</td><td>IF FORWARD LIMIT HIT:
GOTO STATE State #n NEXT, IF Drive #1’s FORWARD LIMIT WAS REACHED</td><td>Transfers to another state if the drive’s forward limit has been reached
GOTO STATE End Travel NEXT, IF Drive #1’s FORWARD LIMIT WAS REACHED</td></tr>
</table>

Appendix D—Table of System Variables

The following is a table of the system variables. Most of them can be both read and written to, but a few of them are read only and cannot be changed by assigning a value in the application programming. The read only variables are #2 (actual pos.(#1)), #23 (actual vel.(#1)), #37 (Analog Position Input), #38 (Auxiliary Analog Input), and #46 (Drive Current Command).

Index	Variable Name	Units	Range of Values	Description
#1	requested pos.(#1)	Resolver counts	–2147483647 to 2147483647	Target position for the motor in Position mode. This value can be changed "on the fly" and the motor adjusts instantly.
#2	actual pos.(#1)	Resolver counts	–2147483647 to 2147483647	Current position of the motor. Note that you cannot change this variable.
#3	prop. gain(#1)	(Counts x Gain)/Scale	0 to 32767	The gain of the error signal applied to the output.
#4	int. gain(#1)	(Counts x Gain)/Scale	0 to 32767	The gain of the integral of the error signal applied to the output.
#5	deriv. gain(#1)	(Counts x Gain)/Scale	0 to 32767	The gain of the differential of the error signal applied to the output.
#6	feed fwd gain(#1)		- - -	Not Available
#7	requested vel.(#1)	Counts/msec	–32767 to 32767	Target velocity for the motor in Velocity mode. Unlike Position mode, this variable is NOT reset when the motor is enabled.
#8	requested cur.(#1)	Max. current/65535	–65535 to 65535	Target current for the motor in Current mode.
#9	acceleration(#1)	Counts/msec	0 to maximum vel. (variable #11)	The maximum acceleration rate allowed. Lowering this value prevents the motor from drawing too much current during acceleration. The drive assumes the motor is capable of accelerating at this rate and generates its motion profiles accordingly. If the motor's actual acceleration lags too far behind this value, it causes following errors in Velocity and Position modes.
#10	deceleration(#1)	Counts/msec	0 to maximum vel. (variable #11)	The complement of Acceleration (Variable #9). A high rate can cause the motor to generate too much current during deceleration.
#11	maximum vel.(#1)	Counts/msec	0 to 32767	Maximum velocity. This variable can be used to prevent the motor from spinning too fast. Again, the drive assumes the motor can attain this velocity and generates following errors if the motor is not keeping up.
#12	maximum cur.(#1)	Percent of Rating 37767 = 100%	0 to 65535	Maximum current delivered by the amplifier in Amps. This variable allows you to limit the maximum output from the drive. When using a new motor for the first time, the maximum current should be set very low until the system is performing well. This prevents damage from overloading the coils and runaway motors.
#13	ace scale(#1)		0 to 32767	The length of the discrete filter unused to generate ideal positions. The larger this number, the slower the motor's acceleration and deceleration.
#14	max. stop vel.(#1)	Counts/msec	0 to 32767	Maximum velocity that the motor can be moving at and still be considered "stopped."
#15	error band(#1)	Counts	0 to 32767	The range in which the motor is still considered in position.
#16	max. error(#1)	Counts	–2147483647 to 2147483647	The maximum difference allowed between the actual and ideal position of the motor. If this limit is exceeded, the motor is disabled. Once the motor is tuned, you should be able to choose a fairly low value for this variable. If this causes errors (the motor is disabled during moves) the acceleration, deceleration, or maximum velocity may be set higher than the motor can achieve.
#17	pos. dead band(#1)	Counts	–2147483647 to 2147483647	The converse of Maximum Error (Variable #16). If the difference between the actual and ideal position is less than half of the position of the dead band, the amplifier output is set to zero. This is useful since some motors cannot be tuned for acceptable performance without oscillating in position.
#18	vel. dead band(#1)		0 to 32767	The converse of Maximum Error (Variable #16). If the difference between the actual and ideal velocity is less than half of the position of the dead band, the amplifier output is set to zero. This is useful since some motors cannot be tuned for acceptable performance without oscillating in position.
#19	in pos. delay(#1)	msec	0 to 32767	How many loop cycles the drive has to be below the maximum stop velocity and within the position band to be considered in position.
#20	skip counter(#1)	- - -	- - -	Not Available
#21	current offset(#1)	D/A counts	0 to 32767	The offset added to any nonzero current command.
#22	integral limit(#1)		0 to 32767	Low pass filter. See *Appendix E.*
#23	actual velocity(#1)	(Counts x Gain)/Scale	0 to 32767	Current velocity of the motor. Like Actual Position, you cannot change this variable.

Index	Variable Name	Units	Range of Values	Description
#24	gain scale(#1)	Value/(2^GainScale)	0 to 32767 (usually 0 to2)	Value used to scale all four PID gain factors (proportional, integral, derivative, and feed forward). This variable is useful if the values needed to tune the PID loop are excessively large or small. The formula is *Actual/Gain = 2^GainScale.*
#25	forward limit(#1)	Counts	–2147483647 to 2147483647	The furthest position the motor can travel in the forward direction. Useful for limited-travel systems, such as linear motors. Note that travel limits are disabled by default.
#26	reverse limit(#1)	Counts	–2147483647 to 2147483647	The furthest position the motor can travel in the reverse direction. Note that travel limits are disabled by default.
#27	follower pos.(#1)	Counts	–2147483647 to 2147483647	Position of the external encoder.
#28	follower mult.(#1)	None	0 to 32767	The ratio of the Follower Multiplier divided by the Follower Divisor times the external position equals the motor reference position.
#29	follower div.(#1)	None	0 to 32767	The ratio of the Follower Multiplier divided by the Follower Divisor times the external position equals the motor reference position.
#30	drive mode(#1)	None		The drive mode word consists of 16 bits, where the lowest five bits contain the mode. Hex 1 Current Mode Hex 2 Velocity Mode Hex 3 Position Mode Other modes are possible, but not implemented on the VC1000. Hex 40 clears the following error, but we recommend that you do not attempt to use this without additional instructions from Fisher Controls. Hex 80 adds friction compensation in the PID loop. Hex 100 forces the drive into Reference mode. Hex 200 enables the forward limit and Hex 400 enables the reverse limit. If the 8000 bit is set to 0, the drive is disabled. Additional bits are reserved.
#31	drive status(#1)	None		The status word returns the drive status to the user. Certain bits that are set or cleared give the user information. When the drive reaches the forward limit, it indicates the Hex 1 bit and when it reaches the reverse limit, it indicates the Hex 2 bit. There are also limit switches. When the system reaches the forward limit switch, it indicates Hex 4 and when it reaches the reverse limit switch, it indicates Hex 8. The forward and reverse limit switches are only active on a DSP card. When there is a drive fault, the Hex 80 bit is set. When there is a drive preference set, Hex 100 is set. Hex 200 is the drive acknowledge bit. When the drive is in position, it will indicate Hex 800 and when the drive is running it indicates Hex 1000. When the drive is enabled it indicates Hex 8000. When the user sends an invalid drive mode in addition to the fault bit, you receive a Hex 40 bit. When there is a drive error overload, you will also receive a Hex 20 bit and when there is a following error, you will receive a Hex 10 bit.
#32	position error(#1)			
#33	ref. position(#1)	Encoder Counts	–2147483647 to 2147483647	The position offset the system uses when a reference is established. The default value is 0.
#34	reset position(#1)	Encoder Counts	–2147483647 to 2147483647	The drive is offset by this value and then the system is reset.
#35	system time	HH:MM:SS		System time in hours, minutes, and seconds.
#36	0	none	0	A constant with the value zero
#37	Analog Position Input		0 to 3896	Reserved for custom use and expansion.
#38	Auxiliary Analog Input		–2047 to 2047	Reserved for custom use and expansion.
#39	Analog Monitor 1		–2047 to 2047	Reserved for custom use and expansion.
#40	Analog Monitor 2		–2047 to 2047	Reserved for custom use and expansion.
#41	Reserved System Variable 5			Reserved for custom use and expansion.
#42	Reserved System Variable 6			Reserved for custom use and expansion.
#43	Reserved System Variable 7			Reserved for custom use and expansion.
#44	Reserved System Variable 8			Reserved for custom use and expansion.
#45	Reserved System Variable 9			Reserved for custom use and expansion.
#46	Drive Current Command			

Appendix E—Useful Formulas for the Design VC1000

Following are useful formulas which can be used to calculate essential system and user variables.

Derivative Gain

The derivative gain is a drive tuning parameter and the VC1000 is shipped with the derivative gain preset. If the desired frequency response is known and the system inertia can be determined, the required derivative gain can be estimated by:

$$Derivative\ Gain\ =\frac{110540\ \times\ Bandwidth}{Torque\ Const}$$

Where :

Bandwidth = Desired maximum frequency response. Usually between 1 and 20 Hz.

Torque Const = The torque constant of the actuator motor. This varies with the actuator construction, so contact the Fisher factory for this value.

The range of values for the derivative gain is 0 to 32767. Increasing this number increases the drive stability. In some cases, using this value will result in the drive making the actuator noisy, increasing wear. If this is the case, the derivative gain will have to be determined by trial and error.

Proportional Gain

The VC1000 is shipped with the proportional gain preset. If the derivative gain is known, the required proportional gain can be estimated by:

$$Proportional\ Gain\ =\frac{\square\ \times\ Bandwidth\ \times\ Derivative\ Gain}{1000}$$

Where :

Bandwidth = Desired maximum frequency response. Usually between 1 and 20 Hz. Usually the same as the frequency response for the derivative gain.

Derivative Gain = The derivative gain calculated above.

The range of values for the proportional gain is 0 to 32767. Increasing this number increases the drive response. In some cases, using this value will result in the drive being too responsive or too sluggish, both of which are undesirable. Being too responsive will make the actuator noisy and increases wear. The actuator may become unstable and even out of control. Being too sluggish may result in the actuator not following the position command well, resulting in poor process control. If either of these is the case, the proportional gain will have to be determined by trial and error.

Integral Gain

The VC1000 is shipped with the integral gain preset. If the proportional gain is known, the required integral gain can be estimated by:

$$Integral\ Gain\ =\frac{\square\ \times\ Bandwidth\ \times\ Proportional\ Gain}{1000}$$

Where :

Bandwidth = Desired maximum frequency response. Usually between 1 and 20 Hz. Usually the same as the frequency response for the derivative gain.

ProportionalGain = The proportional gain calculated above.

The range of values for the integral gain is 0 to 32767. Increasing this number increases the drive response. In some cases, using this value will result in the drive being too responsive or too sluggish, both of which are undesirable. Being too responsive will make the actuator noisy, increases wear and cause the actuator to overshoot the commanded position. The actuator may become unstable and even out of control. Being too sluggish may result in the actuator not following the position command well, resulting in poor process control. If either of these is the case, the derivative gain will have to be determined by trial and error.

Low Pass Filter

The VC1000 has a low pass filter which is useful in electrically noisy environments to quiet the actuator. It is used to filter out high frequency, electrical noise, while allowing the lower frequency position command signals to pass through. The value for the filter can be determined by:

$$Low\ Pass\ Filter\ =\ BW\ \times\ 103$$

Where :

BW = Desired frequency response of the low pass filter, in Hz. Usually at least twice the desired frequency response of the actuator.

The range of values for the low pass filter is 0 to 32767. A value of zero turns the filter off. Increasing this number decreases the drive response. In some cases, using this value will result in the drive being too

sluggish, resulting in the actuator not following the position command well.

Current

The following formula is used to set the current to the servo drive. The calculated current can be assigned to system variables #8 (requested cur.(#1)) and #12 (maximum cur.(#1)) to control the output current. The output units are in counts and can range from –65536 to +65536, which represents ±200%.

$$Current = \frac{Percent\ Torque \times 32768}{100}$$

Where :

PercentTorque = Desired torque as a percent of maximum (ex. Foldback Torque=30)

Velocity

The following formula is used to set the velocity of the actuator. The formula is used to determine the motor speed with the stroking speed determined by the roller screw lead. The calculated result can be assigned to system variables #7 (requested vel.(#1)) and #14 (max. stop vel.(#1)) to control the motor speed. The units are counts per period, with the period being defined by the drive firmware.

$$Velocity = \frac{RPM \times Counts\ Per\ Rev \times 64}{60000}$$

Where:

RPM = Desired motor speed in revolutions per minute

CountsPerRev = Resolver counts per one revolution of the motor. The R/D converter is 12 bits, so this number is 4096.

The period for this drive is 1 millisecond. The number 60000 in the divisor converts the *RPM* to revolutions per msec. Multipliying by the *CountsPerRev* converts this into resolver counts per msec. The number 64 is a scaling factor to improve resolution and reduce truncation errors.

The *CountsPerRev* number could be hard-coded into the software, but it is set as a variable and initialized in the software to improve understandability.

Position Counts

The following formula is used to set the desired valve position. The calculated results can be assigned to system variable #1 (requested pos.(#1)). The resulting number is the number of resolver counts away from the reference position, which could be the valve seat or other reference point. The units are in counts and can either positive or negative. If the actuator extended to find the seat or reference point, all the position counts will be negative numbers. If the actuator retracted to find the seat or reference point, all the position counts will be positive numbers. Remember, the counts represent absolute position and are not relative to the prior position. A number such as 250 or –250 will be near the seat and a number such as 2000 or –2000 will be farther away from the seat.

$$Pos\ Counts = \frac{(Command \square 4000) \times Stroke\ Counts}{16000}$$

Where:

Command = Desired position command in microAmps.

StrokeCounts = The number of resolver counts for the full valve stroke.

This equation assumes the command signal is a 4 to 20mA signal. The number 4000 removes the current offset and the number 16000 represents the maximum possible range of the signal. If the actual signal exceeds 20 mA, the *PosCounts* will be greater than *StrokeCounts* and the valve will over-travel. If this is a problem, some means to prevent this must be written into the user software routine.

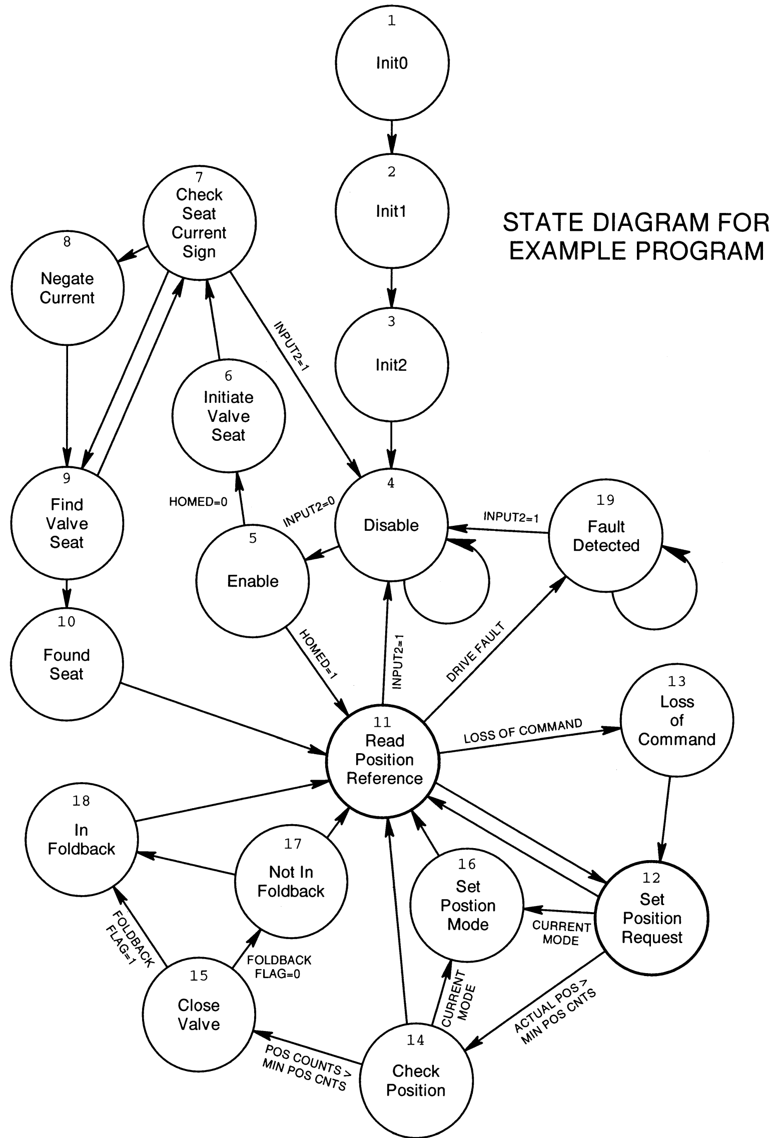

Figure 3. State Diagram for Example Program

Appendix F—Example Program

The following is an example of a drive program. Included is a state diagram (figure 3) to show the program flow, and a description of the function of each program state. The program's function is to control a valve where the valve plug is pushed down to close the valve, which means the actuator extends to close the valve.

The program consists of 19 states. States 1, 2, and 3 are used in initialize variables and constants the program uses. States 6 through 10 are used to calibrate the system as part of a startup sequence. The states are executed only once, every time the drive is powered up.

States 11 and 12 are the principle states, and the program spends most of its time in either of these states. State 11 is where the 4–20 mA analog input command is read and scaled. Checks are made to be sure the valve command is within the correct range and the drive has not faulted. State 12 is where the drive commands the actuator to go to the set point, and checks are made to determine if the valve has been commanded to close.

If the valve has been commanded to close, the drive positions the valve plug until it is near the seat, and then it switches to a constant force mode to actually

close the valve. This is done to be sure the valve closes properly. Due to thermal expansion effects, the valve may have physically changed size since the actuator was calibrated during power up. If the drive was run in a positioning mode all the way to the seat, the valve plug may end up a few thousandths of an inch off the seat, increasing the valve leak rate and decreasing the valve seat life. Or, if the actuator was trying to position the valve plug past the seat line, the valve or actuator could potentially be damaged. States 13 through 18 handle the seat closure requirements.

State 1—Init0

Actions

This state has the program identification information (lines 1 through 5). It also defines and initializes many of the primary user variables such as stroke length (–1125 for 1.125 inches), stroke polarity (the minus sign on the stroke length indicates the actuator extends to close the valve), torque limits, velocity limits and a few other parameters.

Line 7 The resolver to digital converter is a 12–bit device, so there are 4096 resolver “counts” in one revolution of the resolver.

Line 8 The pitch of the actuator lead screw is 0.20 inches, so it takes five revolutions of the motor to move the output shaft one inch.

Lines 9, 10, and 11 The peak, seat, and foldback torque limits are set to 100%, 50%, and 30% respectively. When the valve seats, it does so at 50% of the full drive output current. It will hold that current level until the foldback time runs out, after which the output torque will drop to the foldback current of 30% of the full output current.

Line 12 The foldback delay time is 250 milliseconds.

Lines 13 and 14 The maximum motor speed is 3000 rpm. During the calibration, or “homing” process, the maximum speed is 100 rpm.

Lines 15 through 18 These are valve input commands, expressed in microamperes. The maximum open position is 20000, or 20 mA. If the input goes below 2500 (2.50 mA) the drive assumes the input signal has been lost (due to a bad connection or broken wire) and will drive the actuator to the lost command position, which in this case is 4000, or 4.00 mA. If the input command goes below 4200, or 4.20 mA, the valve will close. With 1.125 travel, the valve is only open by 0.014 inch when the input command is 4.20 mA.

Lines 19 and 20 These are calibration values that are unique to the drive, actuator, and valve combination, and must be determined experimentally. When all the pieces are mounted and mechanical and electrical connections have been made, use an instrument calibrator to apply a 4.000 mA signal to the command input on the servo drive input connector. From the PC type `va #37 <enter>`. The system will respond by displaying a hexadecimal and decimal number. The variable #37 is the analog position input, and what is displayed is the reading of the analog to digital converter connected to that input. The number may look like `0xffff718 (-2280)`. `0xffff718` is the 2’s compliment hex representation of the negative number –2280. Therefore, the ADC count when the valve is closed is –2280. Now the calibrator is used to apply an input command of 20.000 mA. From the PC type `va #37 <enter>`. In our case the result was `0x00000454 (1108)`, so when the valve is fully open, the ADC count is 1108. The numbers –2280 and 1108 must be entered on these two lines, or the valve will not stroke properly.

Transitions

There is only one transition, and that is to state *Init1*.

State 2—Init1

Actions

All of the position moves are based on the number of resolver counts away from the seat. Internally, the servo drive accumulates the resolver counts, adding when moving in one direction, and subtracting when going the other. In the State Init0, we defined the number of resolver counts at 4096 per revolution, and that 5 revolutions were required to move one inch. Therefore, if we want to stroke exactly one inch, we will have to count out 20,480 counts from the resolver and then stop. In state 10, we will establish a zero reference point from which all moves will be made.

Lines 1 through 3 Calculation to determine the number of resolver counts to move from fully closed to fully open. In our case, 23,040 counts.

Line 4 Calculation of the full stroke range of the analog to digital converter on the 4 to 20 mA command input. In our case, 1108 – (–2280) = 3,380 counts.

Lines 5 through 10 Calculation of the operating, seating, and foldback currents. The calculation converts the percentage value into a binary number useable by the servo drive. The order of the operations is important here because of the integer math.

Lines 11 through 17 Calculation of the motor speeds, converting the speed from RPM to resolver counts per millisecond, which is what the drive actually uses.

Lines 18 through 23 Calculations to convert and scale the input position command into resolver counts. The drive always positions the valve on the basis of the number of resolver counts away from a zero reference point. However, the input command is based on a 4 to 20 mA signal. The first step in the conversion is to subtract 4000 from the command, which removes the 4.0 mA offset. The result is divided by 16000 to get the percentage of the command between 4 and 20 mA. The result is multiplied by the number of resolver counts to make a full stroke. The end result is the number of resolver counts from the zero reference point. Note that the actual order of the calculations is different than given here, because of the integer math and the need to preserve accuracy as much as possible.

Transitions

There is only one transition, and that is to state *Init2*.

State 3—Init2

Actions

We continue to establish variables and constants the drive application program will use.

Lines 1 through 3 Initializing some software flags.

Lines 4 and 5 Setting up some parameters that will be used while "homing" the valve.

Lines 6 through 12 Converting and scaling position commands, as was done in the state *Init1*.

Transitions

There is only one transition, and that is to state *Disable.*

State 4—Disable

Actions

This state disables the drive and clears the lost command software flag. The drive will not move the valve plug as long as the drive is disabled.

Transitions

The transitions of this state will be rescanned every millisecond, until the 24 VDC is applied to input number 2 (DI2) on the input connector. When this state is exited, the program will continue its execution at state *Enable*.

State 5—Enable

Actions

This state enables the drive, allowing the drive to position the valve plug.

Transitions

If the valve has been "homed" previously, the program execution will jump to state *Read Position Reference* and begin positioning the valve plug. If the Homing or calibration procedure has not been performed, the drive will begin that procedure next by going to the state *Initiate Valve Seat*.

State 6—Initiate Valve Seat

Actions

This is the start of the homing, or calibration, procedure to locate the seat and determine the "zero" reference point. The basic method is to put the drive into constant velocity mode and drive the valve plug into the seat while monitoring the current to the actuator motor. When the plug hits the seat, the current will rise rapidly. When a certain threshold is crossed, it is assumed the plug is on the seat and the calibration will occur.

This state initializes the variables used in the procedure.

Line 1 Sets the maximum current to the previously determined current for this calibration procedure. (See states *Init0* and *Init1*.)

Line 2 Sets the maximum velocity of the actuator during this procedure. (See states *Init0* and *Init1*.)

In this example program, the velocity is set to 1/3 inches per second.

Line 3 Initializes the variable Filtered seat current, which is used to locate the valve seat

Line 4 Puts the drive into velocity mode.

Transitions

There is only one transition, and that is to state *Check seat current sign.*

State 7—Check seat current sign

Actions

This is the start of the seat detection program loop.

Line 1 Sets the variable `temp` to be the actual current to the actuator motor.

Transitions

The first check is to see if the 24 VDC signal is applied to input number 2 (DI2) on the input connector. If it is not, the drive goes to the state *Disable*, and the drive is disabled without finishing the homing procedure.

When the current is compared, we assume the current is positive. If it is not positive, the program execution is diverted to the state *Negate current*.

If neither of the first two checks changes the program flow, then the program will continue its execution at state *Find Valve Seat*

State 8—Negate current

Actions

If the actual current to the actuator is negative, the value of the variable `temp` is negated to make it a positive number.

Transitions

There is only one transition for this state, and that is to state *Find Valve Seat.*

State 9—Find Valve Seat

Actions

The actions of this state causes the variable `Filtered Seat Current` to rise very rapidly when the valve plug contacts the seat. The object of this to make the process sensitive to finding the seat.

Transitions

If the variable `temp` is less than or equal to the variable `DAC Seat Current`, then the seat has not been found, so the program flow loops back to the state *Check seat current sign*, again. Otherwise, the seat has been located and program execution goes to the state *Found Seat*.

State 10—Found Seat

Actions

The valve plug is on the seat and the servo drive will be reset.

Line 1 The actuator sits motionless for 200 milliseconds. This settling time helps make the calibration more accurate.

Line 2 Sets the system variable `reset position` equal to the actual valve position. This effectively “resets” the drive and now all future moves will be made relative to this “zero” point.

Line 3 Sets the “Homed” flag, indicating the drive is calibrated.

Lines 4 and 5 Resets the velocity and current to the operating velocity and current for normal operation.

Line 6 Puts the servo drive into position following mode. The drive will now follow the command input and position the valve plug relative to the zero position, and proportional to the 4 to 20 mA command signal.

Transitions

There is only one transition, and that is to the state *Read Position Reference*.

State 11—Read Position Reference

Actions

This state reads and scales the 4 to 20 mA command signal. It also puts the actuator current signal on the analog monitor.

Lines 1 through 3 Reads and scales the 4 to 20 mA signal and converts it to the number of resolver counts away from the seated position.

Line 4 Scales and puts the actual current to the actuator motor on the analog output, Analog Monitor 1.

Transitions

There are five transitions.

1. If the drive has detected a fault, the program execution continues with the state *Fault Detected*.
2. If the 24 VDC signal is not applied to input number 2 (DI2) on the input connector, the drive goes to the state *Disable*, and the drive is disabled.
3. If the `Lost Command Flag` has been set, the input command signal has been lost and the program execution will continue with state *Loss of Command.*

4. If the input command signal is less than the loss of signal threshold (2.50mA in this example program) the program execution will continue with the state *Loss of Command*.

5. If none of the above are true, then the signal is within the correct range and there are no faults, so the program execution will continue with the state *Set Position Request* next.

State 12—Set Position Request

Actions

When the drive is in position mode, the drive attempts to position the valve plug to the position set in the variable `requested position`.

Line 1 Sets the commanded position into the variable `requested position`.

Transitions

There are three possible transitions.

1. If the requested position is less than the valve minimum open position, then the program executions continues with state *Check Position* to begin the process of closing the valve. (The valve positions are negative because the actuator is retracting to open the valve. Therefore, if `pos counts` is greater than `MIN POS CNTS`, the commanded position is between the seat and the minimum open position. The valve is to be closed by the actuator.)

2. If the requested position will open the valve and it has been closed (the `Current mode flag` has been set) then the then drive must be returned to the position mode. Program execution continues with the state *Position Mode*.

3. If the last position was with the valve open, and the valve is still to be open (that is to say the valve has not been commanded to close), the program execution continues with state *Read Position Reference* next, to read the position command input again.

Notice that when the drive is positioning the valve and everything is normal (no faults) the program will spend virtually all its time in states 11 and 12.

State 13—Loss of Command

Actions

The input command signal has been lost.

Line 1 Set the variable pos counts to the previously determined failure position. In this example program, the valve is to close. (See Line 15 of the state *Init0*.)

Line 2 Sets the Loss of Command flag.

Transitions

There is only one transition, and that is to the state *Set Position Request*, which will cause the valve to close.

State 14—Check Position

Actions

This state has no define actions. Its sole purpose is to direct program execution.

Transitions

There are three possible transitions. The program flow only came to this state because the valve has been commanded to close.

1. If the actual valve position is closer to the seat than the minimum open position, then the valve is to be closed in the constant current, or constant force, mode. Program execution continues with state *Close Valve*.

2. If for some reason, the valve has been commanded to close and it is still off the seat, but it was previously on the seat (the `Current mode flag` has been set) the drive must be put back into the position mode. Program execution continues with the state *Set Position Mode*.

3. If the actual valve position is further away from the seat than the minimum open position, then the valve is still being controlled in positioning mode. Program execution continues with the state *Read Position Reference*.

State 15—Close Valve

Actions

The only way to get to this state is because the valve has been commanded to close, the valve plug actual position is closer to the seat than the minimum open position, and the drive is still in position mode. This state initializes some variables specific to closing the valve and puts the drive into current mode.

Lines 1 and 2 Sets up the predetermined currents to use.

Line 3 Sets the `Current mode flag`.

Line 4 Puts the drive into current mode to close the valve.

Transitions

If the Foldback Flag has been set, program execution continues with the state *In Foldback*. Otherwise, program execution continues with the state *Not in Foldback*.

State 16—Set Position Mode

Actions

The valve seating action is being terminated. Three flags are cleared and the drive returned to position mode.

Transitions

There is only one transition, and that is to state *Read Position Reference* next.

State 17—Not in Foldback

Actions

The valve is seated using the full Seat Current to overcome packing and seal friction, and be sure the valve is properly seated. After we are sure the valve is on the seat, the packing and seal friction will help keep the plug on the seat and the current in the actuator can be reduced to the Foldback Current.

Line 1 Increments the Foldback timer by one. The drive's loop time is one millisecond, so each count represents one millisecond of the delay time before the actuator current will be reduced to the Foldback Current.

Line 2 Sets the current command to the Seat Current.

Transitions

If the foldback timer has exceeded the Foldback delay time, the program flow continues with the state *In Foldback*. If not, we must continue to check the input command, so program flow continues with state *Read Position Reference* next.

State 18—In Foldback

Actions

The valve plug is on the seat and for a long enough time that the Foldback timer has expried and the current to the actuator is to be reduced.

Line 1 Sets the Foldback Flag.

Line2 Reduces the commanded current to the Foldback Current.

Transitions

There is only one transition, and that is to the state *Read Position Reference*. We must continually monitor the input command.

State 19—Fault Detected

Actions

A fault has been detected, so the drive is disabled.

Transitions

The transitions of this state will loop back to the beginning of this state continuously until the 24 VDC input signal to input 2 on the input connector is removed. When that happens, the program execution will continue with the state *Disable*.

List of States in Example Program

Index	State
001*	Init0
002	Init1
003	Init2
004	Disable
005	Enable
006	Initiate Valve Seat
007	Check seat current sign
008	Negate current
009	Find Valve Seat
010	Found Seat
011	Read Position Reference
012	Set Position Request
013	Stroke
014	Loss of Command
015	Check Position
016	Close Valve
017	Set Position Mode
018	Not In Foldback
019	In Foldback
020	Fault Detected

Listing of Example Program

Index	Actions of State: Init0
01* 02 03 04 05	PRINT: Fuel Valve Control Program \n /* Written DJW 7–28–00 /* Last revised DJW 8–9–00 PRINT: Customer PN XXXXXX–XX Rev NEW \n PRINT: Fisher PN XXXXXXXX012 Rev A \n
06 07 08 09 10	SET: Stroke = –1125 SET: Counts per Rev = 4096 SET: Revs per inch = 5 SET: Peak Torque = 100 SET: Seat Torque = 50
11 12 13 14 15	SET: Foldback Torque = 30 SET: Foldback delay time = 250 SET: Max Vel = 3000 SET: Home Vel = 100 SET: Lost Command Pos = 4000
16 17 18 19 20	SET: Input Signal Lost = 2500 SET: Max Open Pos = 20000 SET: Min Open Pos = 4200 SET: ADC CNT Zero stroke = –2280 SET: ADC CNT Full stroke = 1108
Index	**Transitions of State: Init0**
01*	GOTO STATE Init1 NEXT
Index	**Actions of State: Init1**
01 02 03 04 05	MULTIPLY: Stroke Counts = Stroke * Counts per Rev MULTIPLY: Stroke Counts = Stroke Counts * Revs per inch DIVIDE: Stroke Counts = Stroke Counts / 1000 SUBTRACT: ADC range = ADC CNT Full stroke – ADC CNT Zero stroke MULTIPLY: Operating Current = Peak Torque * 32768
06 07 08 09 10	DIVIDE: Operating Current = Operating Current / 100 MULTIPLY: Seat Current = Seat Torque * 32768 DIVIDE: Seat Current = Seat Current / 100 MULTIPLY: Foldback Current = Foldback Torque * 32768 DIVIDE: Foldback Current = Foldback Current / 100
11 12 13 14 15	MULTIPLY: temp = Max Vel * Counts per Rev MULTIPLY: temp = temp * 64 DIVIDE: Max Vel Counts = temp / 60000 SET: maximum vel.(#1) = Max Vel Counts MULTIPLY: temp = Home Vel * Counts per Rev
16 17 18 19 20	MULTIPLY: temp = temp * 64 DIVIDE: Home Vel Cnts = temp / 60000 SUBTRACT: temp = Min Open Pos – 4000 MULTIPLY: temp = temp * Stroke Counts DIVIDE: MIN POS CNTS = temp / 16000
21 22 23	SUBTRACT: temp = Max Open Pos – 4000 MULTIPLY: temp = temp * Stroke Counts DIVIDE: MAX POS CNTS = temp / 16000
Index	**Transitions of State: Init1**
01	GOTO STATE Init2 NEXT

Listing of Example Program (continued)

Index	Actions of State: Init2
01	SET: Homed = 0
02	SET: Number1 = 1
03	SET: Number2 = 2
04	SUBTRACT: DAC Seat Current = Seat Current – 100
05	SET: Seat Current Filter = 800
06	SUBTRACT: temp = Input Signal Lost – 4000
07	MULTIPLY: temp = temp * ADC range
08	DIVIDE: temp = temp / 16000
09	ADD: Signal Lost Count = temp + ADC CNT Zero stroke
10	SUBTRACT: temp = Lost Command Pos – 4000
11	MULTIPLY: temp = temp * Stroke Counts
12	DIVIDE: Lost Command Pos Counts = temp / 16000
Index	**Transitions of State: Init2**
01	GOTO STATE Disable NEXT
Index	**Actions of State: Disable**
01	DISABLE DRIVE : drive #1
02	SET: Lost Command Flag = 0
Index	**Transitions of State: Disable**
01	GOTO STATE Enable NEXT, IF INPUT Number2 ON GROUP 0 IS LOW
02	RESCAN TRANSITIONS
Index	**Actions of State: Enable**
01	ENABLE DRIVE : drive #1
Index	**Transitions of State: Enable**
01	GOTO STATE Read Position Reference, IF Homed = Number1
02	GOTO STATE Initiate Valve Seat NEXT
Index	**Actions of State: Initiate Valve Seat**
01	SET: maximum cur.(#1) = Seat Current
02	SET: requested vel.(#1) = Home Vel Cnts
03	SET: Filtered Seat Current = 0
04	VELOCITY MODE : drive #1
Index	**Transitions of State: Initiate Valve Seat**
01	GOTO STATE Check seat current sign NEXT
Index	**Actions of State: Check Seat Current Sign**
01	SET: temp = Drive Current Command
Index	**Transitions of State: Check Seat Current Sign**
01	GOTO STATE Disable NEXT, IF INPUT Number2 ON GROUP 0 IS HIGH
02	GOTO STATE Negate current, IF temp < 0
03	GOTO STATE Find Valve Seat NEXT
Index	**Actions of State: Negate Current**
01	NEGATE: temp = – temp
Index	**Transitions of State: Negate Current**
01	GOTO STATE Find Valve Seat NEXT
Index	**Actions of State: Find Valve Seat**
01	DIVIDE: temp2 = Filtered Seat Current / 32768
02	SUBTRACT: temp = temp – temp2
03	MULTIPLY: temp = Seat Current Filter * temp
04	ADD: Filtered Seat Current = Filtered Seat Current + temp
05	DIVIDE: temp = Filtered Seat Current / 32768
Index	**Transitions of State: Find Valve Seat**
01	GOTO STATE Found Seat, IF temp > DAC Seat Current
02	GOTO STATE Check seat current sign NEXT

Listing of Example Program (continued)

Index	Actions of State: Found Seat
01	WAIT: 200
02	SET: reset position(#1) = actual pos.(#1)
03	SET: Homed = Number1
04	SET: maximum vel.(#1) = Max Vel Counts
05	SET: maximum cur.(#1) = Operating Current
06	POSITION MODE : drive #1
Index	**Transitions of State: Found Seat**
01	GOTO STATE Read Position Reference NEXT
Index	**Actions of State: Read Position Reference**
01	SUBTRACT: temp = Analog Position Input – ADC CNT Zero stroke
02	MULTIPLY: temp = temp * Stroke Counts
03	DIVIDE: pos counts = temp / ADC range
04	DIVIDE: Analog Monitor 1 = Drive Current Command / 32
Index	**Transitions of State: Read Position Reference**
01	GOTO STATE Fault Detected NEXT, IF DRIVE #1 HAS FAULTED
02	GOTO STATE Disable NEXT, IF INPUT Number2 ON GROUP 0 IS HIGH
03	GOTO STATE Loss of Command, IF Lost Command Flag = Number1
04	GOTO STATE Loss of Command, IF Analog Position Input < Signal Lost Count
05	GOTO STATE Set Position Request NEXT
Index	**Actions of State: Set Position Request**
01	SET: requested pos.(#1) = pos counts
Index	**Transitions of State: Set Position Request**
01	GOTO STATE Check Position, IF pos counts > MIN POS CNTS
02	GOTO STATE Set Position Mode, IF Current mode flag = Number1
03	GOTO STATE Read Position Reference NEXT
Index	**Actions of State: Loss of Command**
01	SET: pos counts = Lost Command Pos Counts
02	SET: Lost Command Flag = Number1
Index	**Transitions of State: Loss of Command**
01	GOTO STATE Set Position Request NEXT
Index	**Actions of State: Check Position**
	There are no Actions defined for state Check Position
Index	**Transitions of State: Check Position**
01	GOTO STATE Close Valve, IF actual pos.(#1) > MIN POS CNTS
02	GOTO STATE Set Position Mode, IF Current mode flag = Number1
03	GOTO STATE Read Position Reference NEXT
Index	**Actions of State: Close Valve**
01	SET: temp = Seat Current
02	SET: temp2 = Foldback Current
03	SET: Current mode flag = Number1
04	CURRENT MODE : drive #1
Index	**Transitions of State: Close Valve**
01	GOTO STATE In Foldback, IF Foldback Flag = Number1
02	GOTO STATE Not In Foldback NEXT
Index	**Actions of State: Set Position Mode**
01	SET: Foldback Flag = 0
02	SET: Foldback Timer = 0
03	SET: Current mode flag = 0
04	POSITION MODE : drive #1
Index	**Transitions of State: Set Position Mode**
01	GOTO STATE Read Position Reference NEXT

Listing of Example Program (continued)

Index	Actions of State: Not in Foldback
01 02	ADD: Foldback Timer = Foldback Timer + Number1 SET: requested cur.(#1) = temp
Index	**Transitions of State: Not in Foldback**
01 02	GOTO STATE In Foldback, IF Foldback Timer > Foldback delay time GOTO STATE Read Position Reference NEXT
Index	**Actions of State: In Foldback**
01 02	SET: Foldback Flag = Number1 SET: requested cur.(#1) = temp2
Index	**Transitions of State: In Foldback**
01	GOTO STATE Read Position Reference NEXT
Index	**Actions of State: Fault Detected**
01	DISABLE DRIVE : drive #1
Index	**Transitions of State: Fault Detected**
01 02	GOTO STATE Disable NEXT, IF INPUT 2 ON GROUP 0 IS HIGH GOTO STATE Fault Detected NEXT

VC1000 Programming Manual

Fisher and Fisher-Rosemount are marks owned by Fisher Controls International, Inc. or Fisher-Rosemount Systems, Inc.
All other marks are the property of their respective owners.

©Fisher Controls International, Inc. 2000; All Rights Reserved

The contents of this publication are presented for informational purposes only, and while every effort has been made to ensure their accuracy, they are not to be construed as warranties or guarantees, express or implied, regarding the products or services described herein or their use or applicability. We reserve the right to modify or improve the designs or specifications of such products at any time without notice.

For information, contact Fisher Controls:
Marshalltown, Iowa 50158 USA
Cernay 68700 France
Sao Paulo 05424 Brazil
Singapore 128461

FISHER-ROSEMOUNT™

Printed in U.S.A.