Compumotor 4000-M

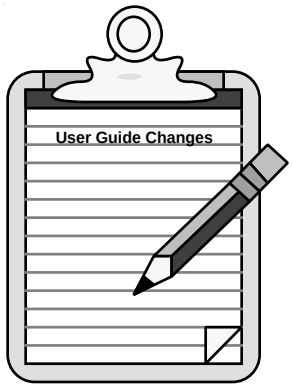
Motion Controller

Limited Availability
Used and in Excellent Condition

Open Web Page

https://www.artisantg.com/62774-2

USER GUIDE CHANGE SUMMARY



The following is a summary of the primary technical changes to this user guide since the last version was released. This reference guide p/n 88-013526-01 C, supersedes 88-013526-01 B.

The entire user guide has been changed according to the new Compumotor user guide styles, format, and illustration standards. Also, the chapters have been renumbered and reorganized. Technical changes to each chapter are summarized below.

Chapter ① IEEE-488

Minor technical changes were made to this chapter

Chapter ② Contouring

Minor changes were made to the **VEL PATH** statement

Chapter ③ Multi-Tasking

There were no changes to this chapter

Chapter ④ Following

The following changes were made to this chapter

- ❏ Added section on Cam Profiling
- ❏ Added section on Motor and Encoder Comparisons
- ❏ Technical changes made to Velocity Smoothing
- ❏ The following statements were added to this chapter

```
FOL CAM
FOL LEAD
FOL DIRSET
FOL ENCCHK
```

Table of Contents

O V E R V I E W

How To Use This User Guide

This user guide is designed to help you install, develop, and maintain your system. Each chapter begins with a list of specific objectives that should be met after you have read the chapter. This section should help you find and use the information in this user guide.

Assumptions

This user guide assumes that the user has a fundamental understanding of computers, basic electrical concepts, basic motion control concepts, and basic serial communication (RS-232C) concepts.

User Guide Contents

This user guide contains the following information.

Chapter ① IEEE-488	This chapter provides a description of the IEEE_488 Option, installation procedures, and command statements.
Chapter ② Contouring	This chapter provides a description of the Contouring Option, installation procedures, and command statements.
Chapter ③ Multi-Tasking	This chapter provides a description of the Multi-Tasking Option, installation procedures, and command statements.
Chapter ④ Following	This chapter provides a description of the Following Option, installation procedures, and command statements.

Installation Process Overview

To ensure trouble-free operation, you should follow the installation procedures outlined in this user guide. Pay special attention to the environment in which the Model 4000 will operate, the layout and mounting, and the wiring and grounding practices used.

Developing Your Application

Before you develop and implement your application, there are several issues that you should consider.

① Clarify the requirements of your application. Clearly define what you expect the system to do.

② Assess your resources and limitations. This will help you find the most efficient and effective means of developing and implementing your application.

③ Follow the guidelines and instructions outlined in this user guide. Proper installation and implementation can only be ensured if all procedures are completed in the proper sequence.

Installation Preparation

Before you attempt to install this product, you should complete the following steps. Successful completion of these steps should prevent subsequent performance problems and allow you to resolve any potential system difficulties before they affect your system's operation.

① Become familiar with the user guide's contents so that you can find information that you need quickly.

② Develop a basic understanding of all system components, their functions, and interrelationships (refer to the *Model 4000 User Guide*).

③ Begin the installation process. Do not deviate from the sequence or installation methods provided.

④ Before you begin to customize your system, check the system functions and features to ensure that you have completed the installation process correctly.

Related Publication

For more information on motion control concepts and Compumotor's complete product line and product capabilities, refer to the current *Parker Compumotor Motion Control Catalog.*

C H A P T E R ①

IEEE-488

Product Description

The Model 4000 IEEE-488 option allows the Model 4000 to act as a peripheral instrument on the IEEE-488 bus of a host computer. The Model 4000 serves as a talker-listener, it can not act as an IEEE-488 bus controller. The Model 4000 uses the *TMS9914* GPIB controller chip, which provides handshake signals according to the IEEE-488 standard. The Model 4000 offers a subset of full IEEE-488 functionality, as summarized below:

- ❏ Talker-Listener
- ❏ EOI on End of Transmission
- ❏ Serial Poll Support
- ❏ Device Clear, Interface Clear
- ❏ Service Request (SRQ) generation
- ❏ Standard IEEE-488 addressing

Installation

Use the following steps to properly install the IEEE-488 Option card into your Model 4000 if you have purchased the option separately. If you purchased your Model 4000 with the IEEE-488 option factory installed, proceed to *Set-Up Procedures*.

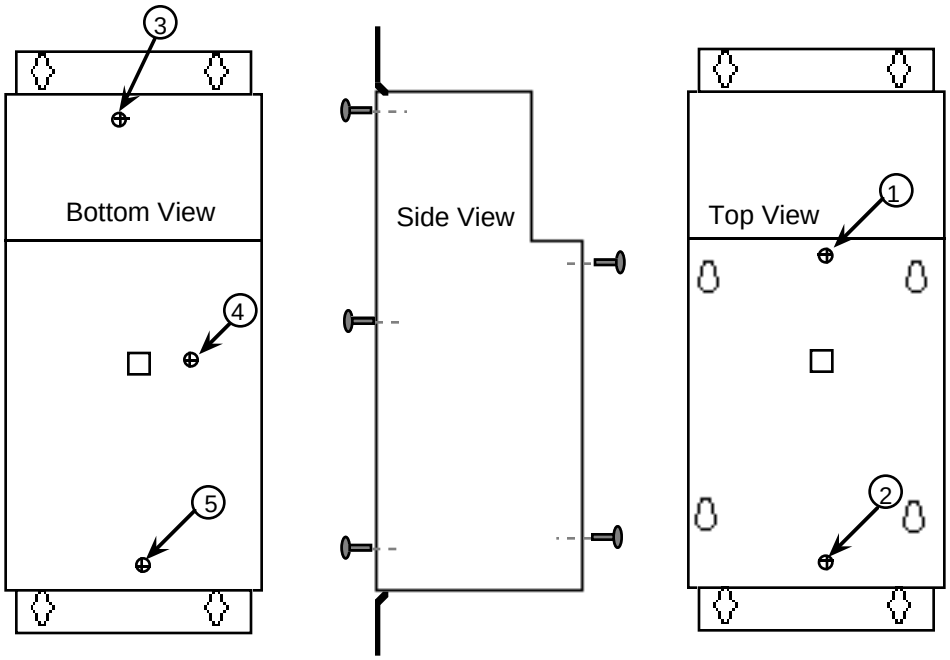
☛ These devices are sensitive to static discharge.

Wear a grounding strap when performing this installation. If a grounding strap is not available you may discharge any buildup of static by touching a grounded piece of metal before opening the Model 4000.

Step ①

Remove AC Power.

To open the Model 4000 enclosure you must disconnect the phone cord and remove screws 1 through 5. Slide internal assembly off, by pushing on the fan side and remove completely.



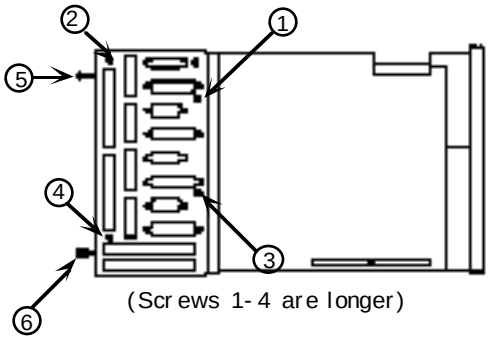
Step ②

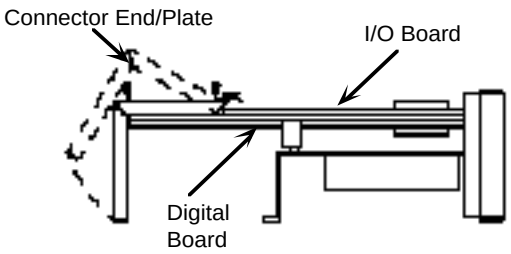
To remove the connector end plate, remove the four axes and two AUX phoenix connectors (make a note of the orientation of these connectors, you will need to re-install them in the same order, do not remove the jumper wire). Remove connector/end plate, by removing screws 1 through 6.

Step ③

Lift end plate off as shown and replace with option card end plate.

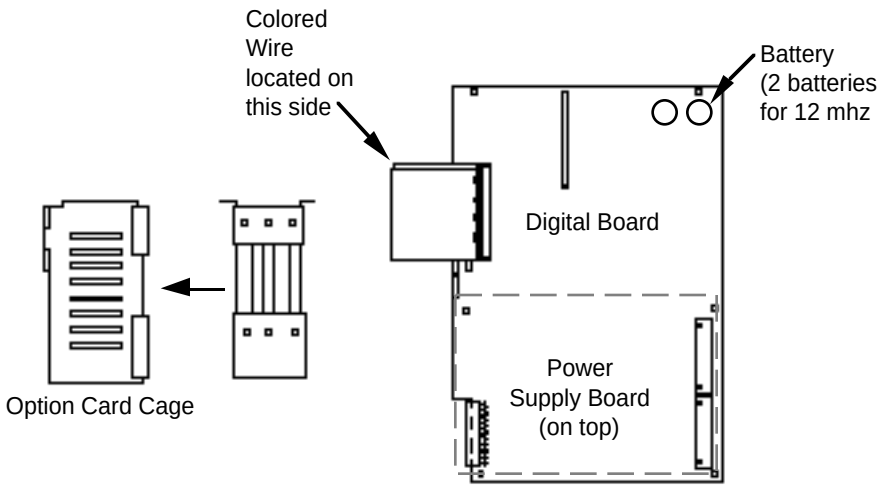
☛J connectors used with IEEE-488 older units are labeled J3 and J2, newer units are labeled J5 and J4



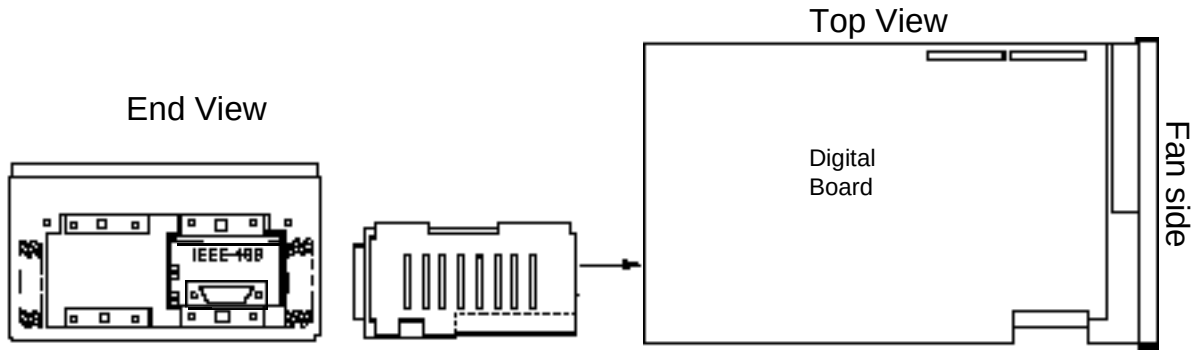


Step ④

Remove the J5 (phone cord) connector (on Digital Board), plug option cable into connector J4 (50 Pin Connector).



Step ⑤ Slide option card cage into position and plug option cable into option card J1. Attach option card cage to end plate using HEX nuts supplied.



Step ⑥ Plug option cable into option card.

Step ⑦ Attach cover plate. Install jack screws into IEEE-488 connector and then install screws into cover plate. Install J5 (phone cord) in its connector.

Step ⑧ Attach the AUX and axis connectors. Slide enclosure cover over the fan end and install the five screws. Plug in the Control Panel connector (phone cord) and AC cord. Apply power.

When starting up your Model 4000 for the first time after installing your IEEE-488 Card, an error message may appear on the front panel. You will need to press the following keys in order to reset the Model 40000.

```
ACCESS
4000
ENTER
ETC  (F6)
RESET  (F2)
ENTER
```

Set-up Procedures

The following paragraphs contain a general discussion of set-up procedures, followed by sample Model 4000 statements and getting started procedures.

The following group of statements are typical of the IEEE-488 definition statements, include them in the 4000's power-up program. Please refer to the statement descriptions for power-up default values. In this example, the

GPIB address is set to 4, and the host will expect to receive error messages from the Model 4000, but not prompts. The host expects that an SRQ will be generated when a message is ready, but not for other bits in the serial poll register.

```
DEFINE GPIB ADDR = 4                    'Set GPIB address to 4.
DEFINE GPIB ERR_MSG ON                  'Enable Model 4000 error messages.
DEFINE GPIB PROMPTS OFF                 'Disable prompts from Model 4000.
DEFINE GPIB SRQ IF SPAS Y N N N N N     'Generate SRQ when message ready.
```

All these statements are needed to set up the Model 4000 IEEE-488 Interface. The Model 4000 automatically detects the presence of the IEEE-488 card, and enables the interface. The serial ports (Port 1 and Port 2) may be used along with the IEEE-488 Interface (Port 3). They are not mutually exclusive.

Step ①

Be sure the host can address the Model 4000. The Model 4000 must be powered up and execute the statements listed above. If the host software addresses the Model 4000 properly, and the IEEE-488 cable is securely fastened, the host will be able to read the serial poll register on the Model 4000. If no other IEEE-488 statements are executed, the value of the serial poll register will be 128 in decimal (or 80 in hex). If the host times out while attempting to read the serial poll register, the addressing could be wrong or the cable not connected.

Step ②

Be sure the host can communicate with the Model 4000. Most host computers will have statements or functions that perform basic IEEE-488 functions. These include:

① Issue device clear
② Conduct serial poll
③ Enable interrupt from SRQ
④ Read string from addressed device (e.g., the Model 4000)
⑤ Write string to addressed device (e.g., the Model 4000)

The sample section of Model 4000 program below could be found inside a typical Model 4000 application. Suppose the programmer wants Bit 0 in the serial poll register to represent a *process ready* state for their application. The **IN** statement serves to generate a message in the form of a prompt, and causes the Model 4000 program to wait for a response from the host. The main loop of the host program will act on this bit 0 when it is detected. Using the statements in Step 1, the Model 4000 has been set up to generate an SRQ when it has a message. The host program should be structured as follows:

① Issue Device Clear
② Enable interrupt from SRQ
③ Write the string **START** with carriage return, line feed
④ Do whatever else it does until interrupted by SRQ.

The Model 4000 statements **START**ed by the host include:

```
OUT SPOL XXXX1                  'Set bit 0 in serial poll register.
IN Q3 = PORT3 ^ READY ^         'This will generate an SRQ and the
                                'Model 4000 will wait for host data.
```

The interrupt service routine of the host should include the following functions:

① Conduct serial poll to identify requesting device
② If it is device 4 (in this example) is bit 0 high?
③ If yes, Write a data string (e.g. "25000")

These examples illustrate the basic use of the IEEE-488 interface.

Use of the Serial Poll Register and SRQ

The Model 4000 takes advantage of the serial poll register as a means of providing device handshake signals and status bits for the host computer.

The lower five bits are strictly under user control, and may be programmed to cause a service request on a low to high transition. The upper three bits have fixed definitions, and may not be altered under direct program control. The Model 4000 uses Bit #7 and Bit #5 as receive and transmit ready bits respectively. Bit #6 indicates that the Model 4000 has generated a service request (**SRQ**). The lower five bits do not carry any predefined meaning and must be set and cleared with the use of the **OUT SPOL** statement. The **OUT SPOL** statement may be placed in the body of the destination of **ON** and **IF** statements, and allow the Model 4000 to indicate program status to a host. The **DISPLAY SPOL** statement allows the current contents of the serial poll register to be displayed on the Model 4000's display.

You can use the **OUT SPOL** statement to set and clear the serial poll register bits. The **DEFINE GPIB SRQ** statement allows a low-to-high transition to generate a service request. When the Model 4000 generates a service request, bit #6 in the serial poll register becomes high. This allows a host computer to receive a service request from any Model 4000 on the bus, and identify the requesting Model 4000(s) by conducting a serial poll. Bit #6 goes low as soon as the Model 4000 has received one serial poll, it is assumed that the host services the request after the serial poll register is read.

Communication with the 4000

The Model 4000 uses bits #7 and #5 of the serial poll register as receive and transmit ready bits respectively. Bit #7, the most significant bit, is the Model 4000's receive ready bit. This bit will be set whenever the Model 4000 is ready to receive at least 85 characters. The Model 4000 has a 256 character receive buffer which must have room for 85 new characters before the receive ready bit goes high. Host computer programs should verify that this bit is high before sending a character string to the Model 4000.

Bit #5 is the transmit ready bit. This bit is set by the Model 4000 whenever it has characters to send to the host computer. The transmit ready bit is cleared when the host reads the last byte. Host computer programs may be structured to accept characters in one of two ways. The simplest method requires that the host program periodically monitor the serial poll register. When the transmit ready bit is high, the host should read the Model 4000's character string. The other method requires that the **DEFINE GPIB SRQ** statement specify that **SRQ** should be generated whenever transmit ready is high and the host is ready to receive another character. The host would then service this request just as it would any other service request. If the host services the request by reading characters from the Model 4000, but does not read all the characters the Model 4000 has to send, then the Model 4000 will issue another **SRQ**. The Model 4000 could issue more than one **SRQ** per string sent, if the host reads the string slowly.

Just as with RS-232C communications, the Model 4000 analyzes the data in its receive buffer as soon as it receives the carriage return character. It will not automatically assume the end of a line if a character is received from the host with the EOI line set. When the Model 4000 sends characters, there may not be a unique character at the end of a line, but the EOI line will always be set with the last character of each individual string the Model 4000 sends. For this reason, the host computer should be set up to terminate reads on EOI. The host may read one character at a time, checking the transmit ready bit of the serial poll before each read. It is very important that the host computer attempt to communicate with the Model 4000 when the bits in the serial poll register indicate it is ready. If the Model 4000 is made a *talker* when it is not ready to transmit, or if it is made a *listener* when it is not ready to receive, it will lock out the IEEE-488 bus until it is ready. Statements that send and receive characters are the **`OUT PORT3`** and **`IN PORT3`** statements. It is recommended that more than one character be sent to a host which is using a *National Instruments* GPIB card. When an **`OUT PORT`** is used with a **`;`** operator, the normal carriage return and prompt are suppressed. This has been known to cause difficulty when the output text was a single character to a *National Instruments GPIB card*.

Device Clear, Interface Clear

The Model 4000 will respond to a device clear from the host controller by stopping any motion and will empty it's IEEE-488 input and output buffers. Device Clear stops all Model 4000 processes (motion will stop, program printing will stop, etc.) programs are not lost, and program statements will continue executing from the point where the Device Clear was encountered. Device Clear clears the serial poll register with the exception of bit 7 (the receive ready bit becomes true). The Model 4000 determines device clear is active when the DCAS bit in the *TMS9914* is set. The Model 4000's response to Interface Clear is to return the IEEE-488 interface to an idle state. Interface Clear has no effect on Model 4000 internal operation. Other IEEE-488 commands that have no affect on the Model 4000 include:

- ❏ **`GET`** (Group execute trigger)
- ❏ **`GTL`** (Go to local)
- ❏ **`LLO`** (Local lockout)
- ❏ **`PPC`** (Parallel poll)
- ❏ **`REN`** (Remote enable)

IEEE-488 Interface Pin-Out

This 25-pin, double row, leaf connector complies with the IEEE-488 specifications and connects to an IEEE-488 bus device. The following table lists the pin assignment for this connector.

Pin #	Signal	Pin #	Signal
1	DI01	14	DI05
2	DI02	15	DI06
3	DI03	16	DI07
4	DI04	17	DI08
5	EO1	18	REN
6	DAV	19	SIGNAL GROUND (DC COMMON)
7	NRFD	20	SIGNAL GROUND (DC COMMON)
8	NDAC	21	SIGNAL GROUND (DC COMMON)
9	IFC	22	SIGNAL GROUND (DC COMMON)
10	SRQ	23	SIGNAL GROUND (DC COMMON)
11	ATN	24	SIGNAL GROUND (DC COMMON)
13	SHIELD (CASE GROUND)		

The following Model 4000 statements are designed to be used with the IEEE-488 option.

DEFINE GPIB SRQ IF SPAS

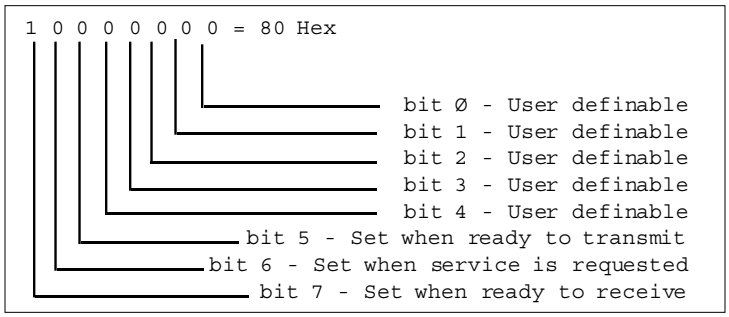
Name	**DEFINE GPIB SRQ IF SPAS**
Descriptor	Define a SRQ Generation
Type	Set-Up
Default	DEFINE GPIB SRQ IF SPAS * * * * * *
Syntax	DEFINE GPIB SRQ IF SPAS Y N N N N N
Options	TAB Y N NULL
	F1 F2 F3 F4 F5 F6

Description

The **DEFINE GPIB SRQ IF SPAS** statement specifies which bits in the serial poll register will activate the service request signal (**SRQ**). The six entries correspond to the six least significant bits of the serial poll response register. The left-most entry is the most significant of the 6 bits: bit #5. The right-most entry is the least significant of the 6 bits: bit #Ø. When any of these six bits become active (1) and this statement has a Y for that entry, the **SRQ** bit (bit #6) is set at the same time that specified bit is set.

Bits #7, #6, and #5 of the serial poll register have special meanings. Bit #7 is active when the Model 4000 is ready to receive characters (a maximum of 85, after which the bit must be checked again). Bit #6 is the **SRQ** bit. It represents the current state of the service request signal. Bit #6 is only activated when the **GPIB SRQ IF SPAS** statement has a Y in one or more of the bit positions (Ø - 5) and the corresponding bit becomes active. Bit #5 is active when the Model 4000 has a string message to transmit.

The data byte below is a common value for the serial poll register.



See Also: **OUT SPOL**

DEFINE GPIB ADDR

Name	**DEFINE GPIB ADDR**
Descriptor	Define GPIB Address
Type	Set-Up
Default	DEFINE GPIB ADDR = n
Syntax	DEFINE GPIB ADDR = n
Options	TAB

F1	F2	F3	F4	F5	F6

Description

The **DEFINE GPIB ADDR** statement sets the address of the Model 4000 for GPIB communication. The address may be 0 - 31 inclusive. The default address is 1. You can control up to 32 Model 4000's with a single IEEE-488 interface.

DEFINE GPIB ERR_MSG

Name	**DEFINE GPIB ERR_MSG**
Descriptor	Define GPIB Error Message
Type	Set-Up
Default	DEFINE GPIB ERR_MSG ON
Syntax	DEFINE GPIB ERR_MSG ON
Options	TAB ON OFF

F1	F2	F3	F4	F5	F6

Description

This statement turns on or off the sending of error messages when invalid data is entered over the IEEE-488 interface. The default is to send an error message string: **Invalid data entered in line.** This is followed by the data that was invalid. The default is **ERR_MSG ON**.

DEFINE GPIB PROMPTS

Name	**DEFINE GPIB PROMPTS**
Descriptor	Define Prompts On/Off
Type	Set-Up
Default	DEFINE GPIB PROMPTS
Syntax	DEFINE GPIB PROMPTS ON
Options	TAB ON OFF

F1	F2	F3	F4	F5	F6

Description

This statement turns ON or OFF the sending of the prompt string (>) after receiving a string over the IEEE-488 interface. The default is ON (send the prompt string). When the **GPIB PROMPTS** are on, you will receive a > as a prompt for each remote statement send down. Prompts that you will receive for different modes of operation are listed below:

Loading a program	**LOAD>**
Inserting statements into an existing program	**INSERT>**
In immediate mode	**IMMED>**

OUT SPOL

Name	**OUT SPOL**
Descriptor	Set Serial Poll Register Bits
Type	Status
Default	OUT SPOL XXXXX
Syntax	OUT SPOL XØ11X
Options	TAB X Q

F1	F2	F3	F4	F5	F6

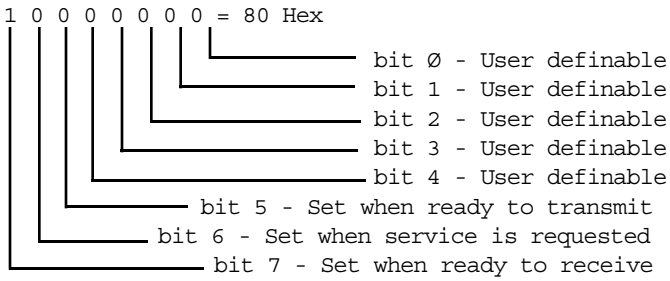
Description

The **OUT SPOL** statement sets, clears, or leaves unchanged the least significant 5 bits of the **GPIB** serial poll register. An **X** means leave the bit unchanged, a 1 means set the bit, and a 0 means clear the bit. A variable (**Q1-Q99**) may be used instead of the bit pattern.

If a variable is entered, the contents of the variable are used to create a binary number of 5 bits of magnitude (less than 32, as in the example below). The fractional portion and the sign of the variable is ignored (digits to the right of the decimal). Also, the number must not exceed 31 or an execution error will be generated.

If the **DEFINE GPIB SPAS** statement enables an **SRQ** for any of the 5 least significant bits, the **SRQ** bit is also set when the corresponding serial poll register is set. An **OUT PORT3** statement will generate an **SRQ** if the **DEFINE GPIB SPAS** statement's bit 5 is set.

The data byte below is a common value for the serial poll register.



Statement	Description
DEFINE GPIB SPAS Y N Y N N N	'Set SRQ when bit 3 or transmit ready.
MATH Q1 = 8	
OUT SPOL Q1	'Same as OUT SPOL 01000. SRQ is generated. 'Serial poll register = 11101000 = ØE8H.

C H A P T E R ②

Contouring

Product Description

The Model 4000 allows the user to define and execute up to 100 two dimensional motion paths. A path refers to the path traveled by the load in an XY plane, and must be defined before any motion takes place along that path. The Model 4000 assumes that the X and Y axes have the same linear resolution (i.e., it takes the same number of steps from the Model 4000 to make each axis go a specified distance in the XY plane). If you have either different drive resolutions or mechanical translations, contact the Custom Products Group at Compumotor (800-358-9068). They can provide information on custom software that is available which allows compensation for mechanical differences, or elliptical contouring if your application requires. A third axis labeled the C axis may be included to keep an angular position which changes linearly with the path direction. The path direction is the vector addition of the travel of axes X and Y. A fourth axis labeled the P axis may be included to keep a position which is proportional to the distance traveled along the path described by X and Y. The X, Y, C and P axes can be specified as any of the Model 4000's four axes.

A path consists of one or more line or arc segments whose endpoints are specified in terms of X and Y positions. The endpoint position specifications may be made using either absolute or incremental programming. The segments may be lines or arcs, both of which are described in greater detail in the following sections. Each path segment is determined by the endpoint coordinates, and in the case of arcs, by the direction and radius or center. It is possible to accelerate, decelerate or travel at constant velocity (feedrate) during any type of segment, even between segments. For each segment, the user may also specify an output pattern which can be applied to the POB outputs at the beginning of that segment.

All paths are continuous paths (i.e., the motion will not stop between path segments, but must stop at the end of a path). It is not possible to define a path which stops motion and then continues that path. To achieve this result, two individual paths must be defined and executed. A path may, however, be stopped and resumed by using the **STOP** and **RESUME** functions, either from the front panel or from the remote RS-232 interfaces. In this case, motion will be decelerated and resumed along the path without loss of position. If any of the participating axes are stopped due to any other reason, all participating axes will stop abruptly, and motion may not be resumed.

These reason may include encountering an end of travel limit, issuing a **KILL** command, or stall detection.

Installation Instructions

Use the steps below to properly install the -C option into your Model 4000 if you have purchased the option separately. If you purchased the contouring option factory installed, proceed to *Path Definition*. If you purchased the -CFM option separately, please refer to the installation instructions in Chapter 3, *Following*.

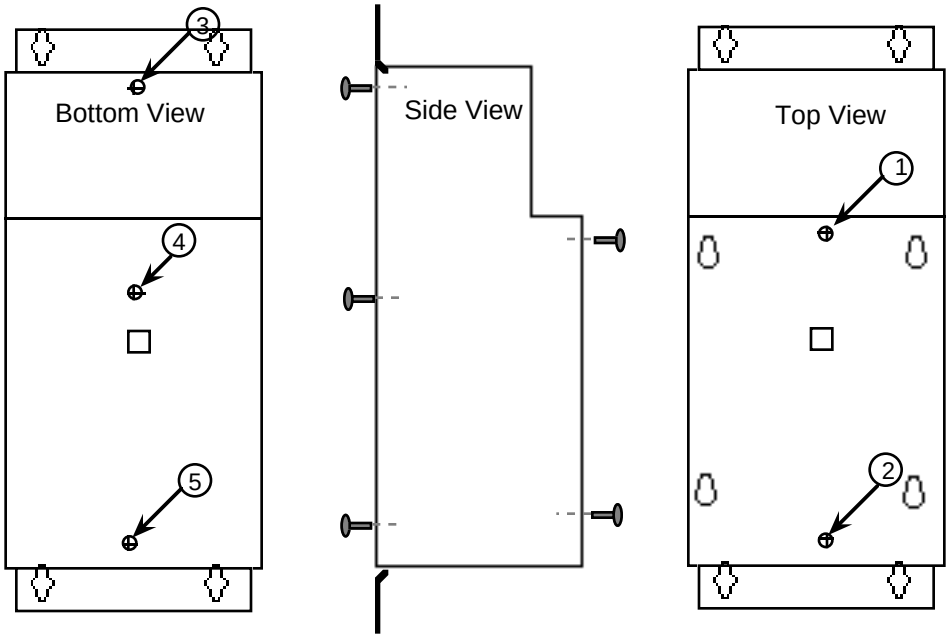
☛ These devices are sensitive to static discharge.

A grounding strap should be worn when performing this installation. If you do not have a grounding strap available you may discharge any buildup of static by touching a grounded piece of metal before opening the Model 4000.

Step ①

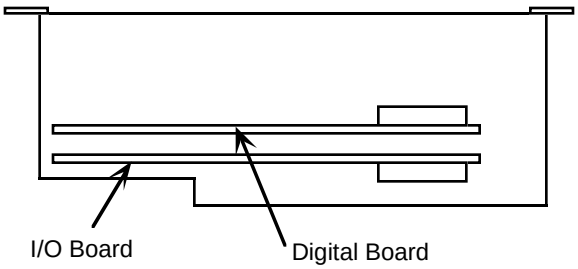
Remove AC Power.

To open the Model 4000 enclosure you must disconnect the phone cord and remove screws 1 through 5. Slide internal assembly off, by pushing on the fan side and remove completely.



Step ②

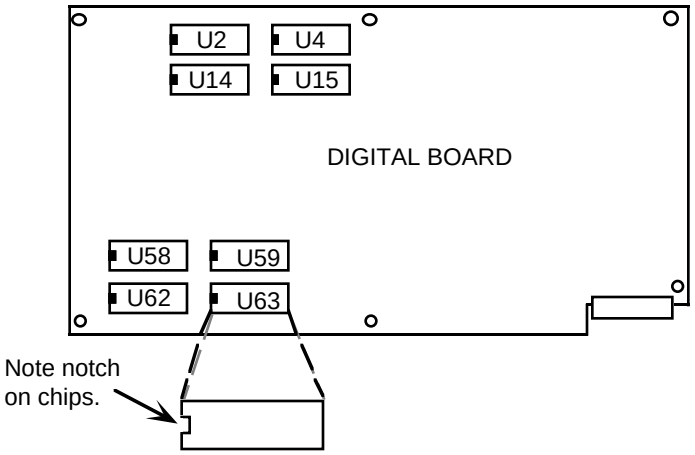
Refer to the following tables to determine the IC's you will need to remove. Carefully turn unit over and remove the appropriate IC's with a small screwdriver between the IC and the socket.



Use the following table if the sticker on the outside sheet metal of your unit has a serial number less than 91-1114XXXX.

IC #	Non-Contouring	Contouring	Memory Expansion
U2	92-010871-01	92-010871-01	
U4	92-010871-11	92-010871-11	
U14		32-011057-01	
U15		32-011057-11	
U58			32-011057-01
U59			32-011057-11
U62	92-010872-01	92-011619-01	
U63	92-010872-11	92-011619-11	

☛ The IC's for slots U14, U15, U58, and U59 are unmarked and interchangeable

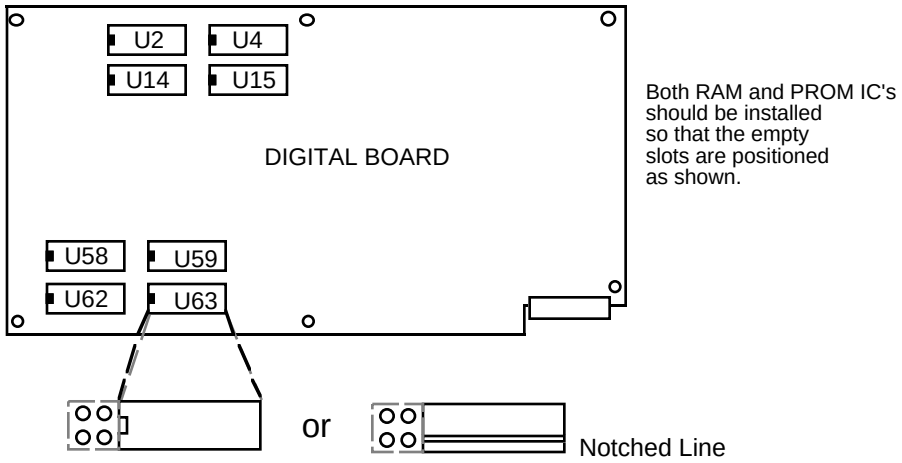


Use the following table if the sticker on the outside sheet metal of your unit has a serial number greater than or equal to 91-1114XXXX.

IC #	Non-Contouring	Contouring	Memory Expansion
U2	92-012117-01	92-012117-01	
U4	92-012117-11	92-012117-11	
U14		32-012051-01	
U15		32-012051-11	
U58			32-012051-01
U59			32-012051-11
U62	92-012118-01	92-012119-01	
U63	92-012118-11	92-012119-11	

Install the IC's from the upgrade kit. Verify that the IC's have been installed with the notched position as shown and that no pins are bent.

☛ The IC's for slots U14, U15, U58, and U59 are unmarked and interchangeable



Both RAM and PROM IC's should be installed so that the empty slots are positioned as shown.

Step ③ Reassemble the Model 4000. Slide enclosure cover over the fan end and install the five screws. Plug in the Control Panel connector (phone cord) and AC cord. Apply power.

Step ④ When starting up your Model 4000 for the first time after installing your new PROMS, an error message may appear on the front panel. You will need to press the following keys in order to reset the Model 4000. This will clear the error message.

```
ACCESS
4000
ENTER
ETC  (F6)
RESET  (F2)
ENTER
```

Path Definition

Path definition and execution may be done by using the X command language or Model 4000 statements in a program. The Model 4000 compiles a completed path definition and stores the compiled path data in a form which can be used for subsequent execution. To compile means to translate a path definition from a form that is meaningful to the programmer into one that is optimum for the Model 4000 internal control.

Up to 100 individual paths may be defined and compiled, as long as each path has at least one segment, and the sum of all the segments of all the paths does not exceed 500. All 100 path definitions may be compiled and ready to execute at any time. Paths defined using Model 4000 statements are specified with a path name. Once a path definition is compiled, it may be executed repeatedly without being re-compiled. Path compilation delay will be the sum all of the individual segment compilation delays included in the path. The following table gives the maximum compilation time for each type of segment. Path execution delay will be 3-5 ms/per path.

Segment type	Max. time (msec.)
line	25
circle	25
radius specified arc	50
center specified arc	80

If the Model 4000 application requires multiple paths whose combined segment count exceeds 500, then an existing path compilation must be deleted to make room for a new path compilation. Redefining an existing path name will automatically delete the existing path compilation with that name. The ability to delete and redefine paths allows a Model 4000 program to contain multiple paths whose combined segment count exceeds 500, even though only 500 segments may exist in the pre-compiled form at one time. Path definitions are common to all programs in the Model 4000's memory. The 100 path limit means 100 paths total in all programs, not 100 paths per program. The **PATH UNCOMP** statement would occur within any program which needed to make room for a new compiled path.

In the following example, storage space is made available for the definition of path **WIDGET3** by first deleting the compiled version of paths **WIDGET1** and **WIDGET2**. The **PATH DEF** statement begins the compilation of the path **WIDGET3**.

Example

Statement	Description
PATH UNCOMP WIDGET1	'Remove compilation of WIDGET1
PATH UNCOMP WIDGET2	'Remove compilation of WIDGET2
PATH DEF WIDGET3 4 2 1 3	'Begin definition of WIDGET3

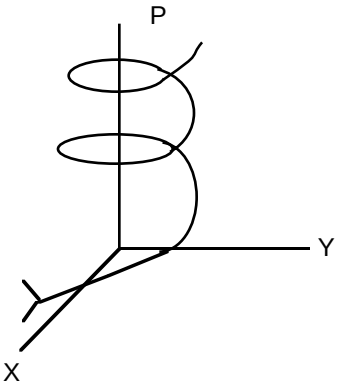
Participating Axes

Some Model 4000 contouring applications may require the execution of more than one path in order to complete a part or finish an operation. The application may require that different paths take place in different planes of a three dimensional work area. In addition, some of the paths may require a third axis to move either tangent, or proportional to the path. For these reasons, the Model 4000 offers great flexibility in the specification of participating axes.

A path definition must start with the Model 4000 statement or X command which specifies the participating axes. The Model 4000 statement used to specify the axes also contains a name for the path. The X, Y, C and P axes can be specified as any of the four axes, and this specification must be made before any of the path travel specifications are made. The X and Y axes must be specified, the third axis labeled C and the fourth axis labeled P are optional.

The C axis will maintain an angular position which changes linearly with the direction of travel in the X-Y plane. This allows the C axis to control an object, which must stay tangent (or normal) to the direction of travel such as a cutting tool. The C axis must also be specified by its signed resolution. The magnitude of the resolution is the number of C axis motor steps in 360 degrees of an arc drawn by the X and Y axes. The sign of the resolution specifies the direction of rotation of the C axis.

The P axis will keep a position which is proportional to the distance traveled along the X-Y path as the path is executed. This allows the P axis to act as the Z axis in helical interpolation, or to control the motion of any object which moves with distance and velocity proportional to the path. The P axis must also be specified by the signed ratio of P axis travel to path travel. The magnitude of this ratio may range from .to 1000. The sign of this ratio specifies the direction of rotation of the P axis.



A sewing machine application may require all four axes (X,Y,C, and P). The X and Y axes would direct the sewing head along the required path. The C axis would keep the sewing head pointed into the direction of travel. The P axis would control the speed of the needle, so that an even stitch is made, regardless of path speed.

The following example begins the definition of a path named **DRAW1**. The X and Y axes are specified to be axes 4 and 2. The path includes the C axis to be axis 1, with a resolution of 100,000 steps. It also includes the P axis to be axis 3, with a ratio of P axis travel to path travel specified as 2.5:1.

Example

Statement	Description
`PATH DEF DRAW1 4 2 1 3`	`'Begin definition of DRAW1.`
`PATH C_RES 100000`	`'Define C axis resolution.`
`PATH P_RATIO 2.5`	`'Define P axis ratio.`

Path Acceleration, Deceleration, and Velocity

A path may be composed of many segments, each with their own motion parameters. The path velocity, acceleration, and deceleration specifications currently in effect at the time a segment is defined will apply to that segment. This allows construction of a path which moves at one velocity for a section of the path, then moves at a different velocity for another section. In most cases, it will be desirable to maintain a constant velocity throughout the path, but is easy to define a path in which each segment has its own velocity. For example, this may be useful when a tool needs to slow down to round a corner, or to allow the rate of glue application to be controlled by the path speed. Acceleration and deceleration may also be specified separately. The example below illustrates the specification of velocity, acceleration, and deceleration in that order.

Example

Statement	Description
`VEL PATH 10000`	`'Path velocity 10,000 steps/sec.`
`ACCEL PATH 400000`	`'Path accel 400,000 steps/sec`^{2}.
`DECEL PATH 700000`	`'Path decel 700,000 steps/sec`^{2}.

Encoder Mode

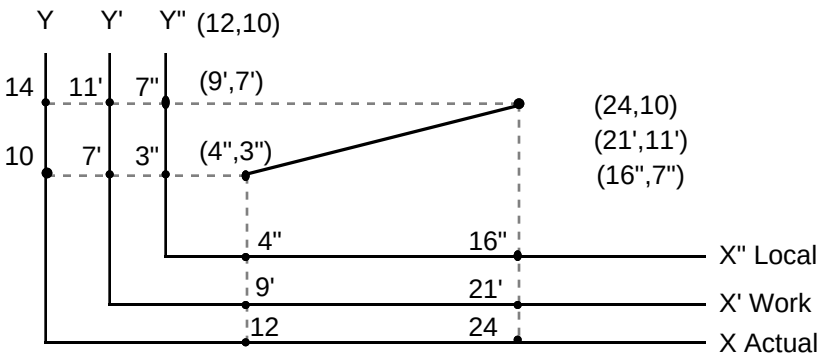
The Model 4000 does not recognize encoders for positioning during contouring. If the present mode of operation is **EABS** or **EINC**, the Model 4000 will not position to encoder feedback steps. The user may need to adjust their **UNIT PATH POS** statement to reflect the correct scale factor for motor steps. However, during contouring the encoder can still be used for stall detection.

Segment Endpoint Coordinates

The endpoint position specifications of lines and arcs may be either absolute or incremental. The Model 4000 stores the endpoint data for all of its compiled segments internally as incremental, relative to the start of the segment. But in order to ease the programming task, absolute coordinates and multiple coordinate systems may be used. When incremental coordinates are used to specify an endpoint, the X and Y endpoint values represent the distances from the X and Y start point of the segment being specified. Center specifications of an arc are always incremental (i.e., relative to the start of that arc segment). When absolute coordinates are used to specify an endpoint, the X and Y endpoint values represent that segment's position in the specified coordinate system. Incremental and absolute programming are specified with the **PATH XY MINC** and **PATH XY MABS** statements. Incremental programming is the default at the beginning of a path definition.

Coordinate systems allow the assignment of an arbitrary X-Y position as a reference position for subsequent absolute endpoint specifications. The Model 4000 allows the use of two coordinate systems for use with absolute coordinate programming. These are called the *Work* coordinate system and the *Local* coordinate system. These are specified with the **PATH XY WORK** and **PATH XY LOCAL** statements. Neither coordinate system needs to represent the set of absolute positions on the path when the path actually executes. Those positions could be any value at the time path execution begins.

The figure below illustrates a line with its coordinates labeled in three coordinate systems. The first set represents the actual machine position at the time the line is being drawn. The next set is the Work coordinate system, used for a reference during definition of the path. The last is the Local coordinate system.



The Work and Local coordinate systems are provided to allow absolute endpoint definition of a segment without needing to know the actual position of that segment during execution. If no **PATH XY** statements precede the first segment statement when a path definition begins, the Model 4000 will place the start of the first segment at location (0,0) in the Work coordinate system. By using the **PATH XY WORK Xpos Ypos** statement, the programmer defines subsequent absolute endpoints to refer to the Work coordinate system, and also locates that coordinate system such that the starting position of the next segment is at (Xpos, Ypos) of the Work coordinate system.

The Local coordinate system is provided so that if a section of a path is to appear in multiple locations along the path, the segments which compose that section can be put in a subroutine and programmed in absolute coordinates. By using the **PATH XY LOCAL Xpos Ypos** statement, the programmer defines subsequent absolute endpoints to refer to the Local coordinate system, and also locates that coordinate system such that the starting position of the next segment is at (Xpos, Ypos) of the Local coordinate system.

A single path definition may include both absolute and incremental programming, and be required to switch between Work and Local coordinates several times. At any point along a path definition, coordinates may be switched from absolute to incremental, or from incremental to absolute. When switching to absolute, all subsequent endpoint specifications are assumed to be absolute with respect to the coordinate system in effect at that time. This remains true until the reference system is switched to incremental, or to a new absolute reference. When switching from Work coordinates to Local coordinates, the Local X and Y start positions of the following segment must be specified with the **PATH XY LOCAL Xpos Ypos** statement. When starting a path definition with Work coordinates, or when switching to Work coordinates, the starting position of the next segment may either be specified or assumed. If Work X and Y start positions are specified with the **PATH XY WORK Xpos Ypos** statement, the Model 4000 returns to the Work coordinate system. The Work coordinate system is also shifted so that the starting position of the next segment is Xpos, Ypos. If Work X and Y start positions are not specified, by using the **PATH XY WORK * *** statement, the Model 4000 returns to the Work coordinate system, but does not shift the Work coordinate system.

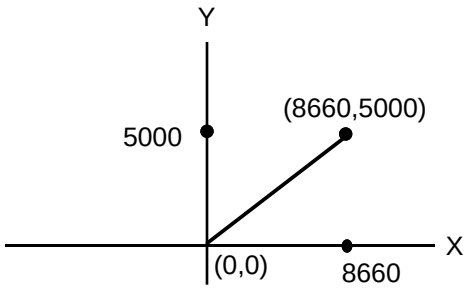
Ease of programming results from the ability to switch between absolute and incremental, and to re-define the coordinate systems between sections of a path. This allows individual sections of path definition to have Local coordinate systems, yet still be integrated into the complete path.

Lines

Lines are the simpler of the two types of segments which compose a path. The placement, length, and orientation of the line is completely specified by the endpoint of the line segment and the endpoint of the previous segment. As described above, endpoints can be specified with absolute or incremental coordinates. Each line may take up to 25 ms to compile. The example below is specified with incremental coordinates and results in a line segment 10,000 steps in length, at 30 degrees in the X-Y plane.

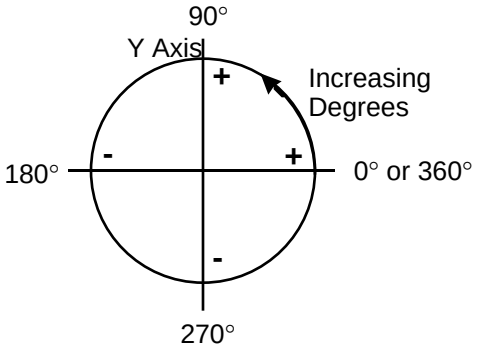
Example

Statement	Description
`PATH LINE 8660 5000`	`'LINE segment to (8660,5000).`



Arcs

Arcs are more complex to specify than lines, because there are four possible ways to get from the start point to the end point. The radius of an arc may either be specified directly or implied by the center specification. In the Model 4000, all path descriptions refer to the X-Y plane. The general convention describing the X-Y plane, as viewed from a drawing, is as follows. The X axis is shown as the left-right axis, with left being negative and right being positive. The Y axis is the up-down axis with down being negative and up being positive. Angles start at zero and increase in the CCW direction of rotation. A line segment, or the radius of an arc is at zero degrees if the incremental endpoint has a positive X component and zero Y component. The angle is 90 degrees if the endpoint has a positive Y component and zero X component.

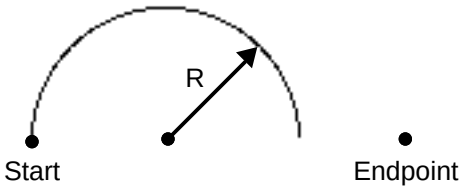


Radius Tolerance Specifications

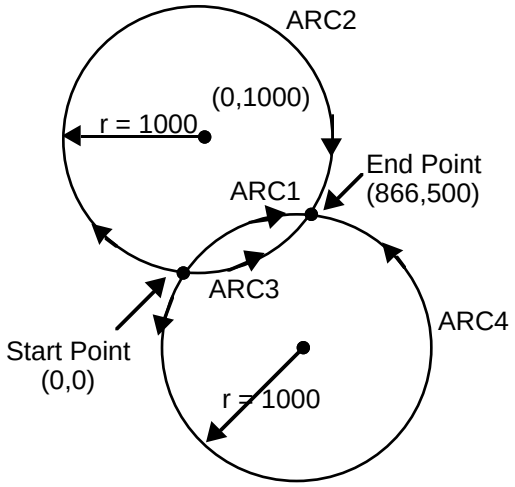
All arcs have an associated radius. In the Model 4000, the radius may either be specified explicitly, or implied by an center specification. In both cases, it is possible that the radius may not be consistent with the specified endpoint of the arc. This could be a result of improper specification, user calculation error, or of round-off error in the internal arithmetic of the Model 4000. For this reason, the Model 4000 allows the specification of a radius tolerance. The radius tolerance is specified in the same units as the radius and X and Y data. The radius tolerance has a factory default of +/- one step, which is just enough to overcome round-off errors. The radius tolerance may be specified at any point along the path definition, and may be changed between one arc and the next. Each arc definition will be compared to the most recently specified radius tolerance. The radius tolerance should be about the same as the dimension tolerances of the finished product. The following paragraphs explain how the radius tolerance is used for the two types of arc specifications, and give syntax examples for the radius tolerance specification.

Radius Specified Arcs

Specification of an arc using the radius method requires knowledge of the start point, the end point, and the sign and magnitude of the radius. The Model 4000 knows the start point to be either the start of the path, or the end of the previous segment. The end point and radius are provided by the user's program. It is possible to specify an impossible arc by specifying an end point which is more than twice the radius away from the start point. In this case, the Model 4000 will automatically extend the radius to reach the endpoint, provided that the automatic radius change does not exceed the user specified radius tolerance. If the required radius extension exceeds the radius tolerance, the Model 4000 will respond with an execute error, and no arc will be generated.



The following figure shows the four possible ways to move from the start point to the end point using an arc of radius 1000. Arc 1 and 2 both travel in the CW direction, arc 3 and 4 both travel in the CCW direction. Arc 1 and 3 are both less than 180 degrees. An arc of 180 degrees or less is specified with a positive radius. Arc 2 and 4 are both greater than 180 degrees. An arc of more than 180 degrees is specified with a negative radius. A radius specified arc takes up to 50 milliseconds to compile. The example below shows the radius tolerance specification and the specifications of arcs 1, 2, 3, and 4 respectively. In the Model 4000 statements, the order of the data is X, Y, R from left to right.



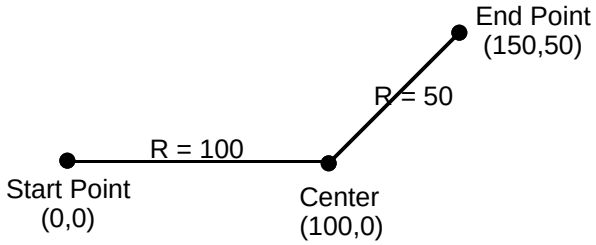
Example

Statement	Description
`PATH RAD_TOL 5`	`'Five steps of radius tolerance.`
`PATH RCW 866 500 1000`	`'Arc 1, CW < 180 degrees.`
`PATH RCW 866 500 -1000`	`'Arc 2, CW > 180 degrees.`
`PATH RCCW 866 500 1000`	`'Arc 3, CCW < 180 degrees.`
`PATH RCCW 866 500 -1000`	`'Arc 4, CCW > 180 degrees.`

Center Specified Arcs

Specification of an arc using the center method requires knowledge of the start point, the end point, and the center point of the arc. The X coordinate of the center is referred to with the letter **I**, and the Y coordinate of the center is referred to with the letter **J**. When an arc is specified with the center, another potential problem arises. It is possible to specify the center of an arc such that the radius implied by the start point does not equal the radius implied by the end point. In this case, the Model 4000 will re-locate the center so that the resulting arc has a uniform radius and the starting and ending angles come as close as possible to those implied by the user's center specification. This automatic center relocation will take place only if the start point and end point radius difference does not exceed the user specified radius tolerance. If the radius tolerance is exceeded, an execute error will result, and the arc will not be included in the compiled path. While automatic center relocation will ensure a continuous path, it may result in an abrupt change in path direction. This happens because a new location for the center results in a new tangent direction for an arc about that center. A center specified arc may take up to 80 milliseconds to compile.

The example below shows the specifications of arcs 1, 2, 3, and 4. In the Model 4000 statements, the order of the data is X, Y, I, J from left to right.

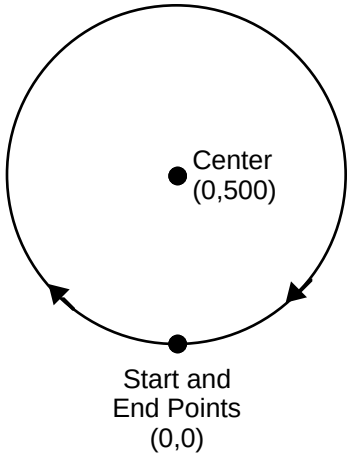


Example

Statement	Description
`PATH OCW 866 500 866 -500`	`'Arc 1, CW < 180 degrees.`
`PATH OCW 866 500 0 1000`	`'Arc 2, CW > 180 degrees.`
`PATH OCCW 866 500 0 1000`	`'Arc 3, CCW < 180 degrees.`
`PATH OCCW 866 500 866 -500`	`'Arc 4, CCW > 180 degrees.`

Circles

A circle is a special case of an arc whose endpoint is the same as the starting point. Because these two points are the same, it is impossible to determine the location of the circle's center from a radius specification. For this reason, an arc which is a complete circle must be specified using the arc center specification method. An arc with identical starting and ending points specified with the radius method will be ignored. A circle may take up to 25 milliseconds to compile.

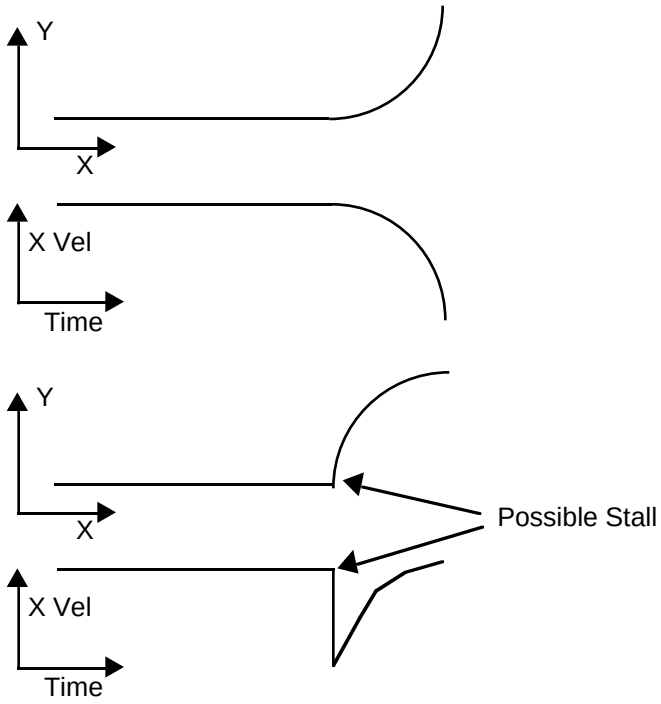


Example

Statement	Description
`PATH OCW 0 0 0 500`	`'Circle with center at (0,500).`

Segment Boundary

So far, all the examples given have shown isolated line or arc segments. Most paths will consist of many segments put together. The point at which the segments are connected is called a segment boundary in this text. The Model 4000 automatically ensures that the path is continuous, in that segments are placed end to end. The path velocity may either be constant or change from segment to segment, according to user specification. Velocity changes use the specified acceleration and deceleration and may take place even across segment boundaries. The programmer should ensure that direction of travel is also continuous across segment boundaries. If the direction change is abrupt the X and Y axes will suffer abrupt acceleration or deceleration. The Model 4000 ensures that there will be no abrupt direction change within a segment, but the programmer is responsible for ensuring that the direction is continuous across segment boundaries. At low speeds, some motor and mechanical configurations will tolerate such abrupt changes, and the Model 4000 will accept such a program, but it is generally good practice to design paths with smooth direction changes. This may be done by designing a path using arcs to round corners.



Using the C Axis

The C axis is an axis whose position changes in a manner linearly related to the direction of travel in X and Y (i.e., the path direction). The C axis would be used in applications that require a work piece or tool to remain tangent or perpendicular to the path direction. Examples would be a knife always pointing into the cut, or a welding head staying normal to the weld.

The magnitude of the C axis resolution refers to the number of steps of C axis position change for 360 degrees of direction change in the X-Y plane. This number may be the same as, or different from the C axis motor resolution, allowing any gearing that is convenient for the mechanics. If the C axis load is to be driven directly, the C axis resolution should be the same as the C axis motor resolution. This will cause the C axis motor and load to rotate once when a circle is drawn by the X and Y axes. If the C axis load is to be geared, for example 5:1, the C axis resolution specifications should be five times the C axis motor resolution. This will cause the actual motor to rotate five times and the load to rotate once when a circle is drawn by the X and Y axes.

The number may be positive or negative, allowing greater flexibility in C axis motor mounting orientation. If the sign is positive, the C axis will rotate in the positive direction when CCW arcs are drawn. If the sign is negative, the C axis will rotate in the negative direction when counter-clockwise arcs are drawn.

The C axis is assumed to be in the proper position when path execution begins. It will change position only as the direction of travel changes. The program must position the C axis before the path is executed. This can be done with the **`MOVE HOME`** statement or a **`MOVE`** to a position.

Because the C axis position changes linearly with the direction of X-Y travel, it is important to avoid path definitions which result in an abrupt direction change between segments. The segment boundary considerations for the C axis are similar to those for the X and Y axes, except that abrupt direction changes will result in abrupt C axis position changes. The X and Y axis would only suffer large accelerations, which may or may not cause a stall. The C axis will suffer impossibly high velocity commands, causing stall and position loss.

Using the P Axis

The P axis is an axis whose position and velocity are proportional to the position and velocity traveled by the load along the path generated by X and Y. It can be used as the Z axis in helical interpolation, or to control other motion which must be proportional to the X-Y path motion. The proportionality of the P axis is specified as a ratio, with a range of plus or minus .to 1000. The sign of the ratio determines which direction the motor will turn. The magnitude specifies the ratio of P axis travel to path travel, regardless of path direction or segment type. This ratio is essentially a position ratio, but because the ratio is maintained at every instant, it also becomes a velocity ratio.

The P axis only responds to distance traveled along the path, and is not affected by direction changes in the path. The only caution which must be observed comes when a high ratio is specified. In this case, path velocity and acceleration are amplified, which may result in stalls or impossible velocities.

Outputs Along the Path

For each segment, the user may also specify an output pattern which is to be applied to the POB outputs at the beginning of that segment and remain throughout that segment. These segment defined POB output patterns are stored as part of the compiled path definition. These outputs will change state at some time in the range of 1.5 ms before the beginning of the segment to 0.5 ms after the beginning of the segment. The POB outputs may not be controlled more precisely than this, because the Model 4000 updates its record of path position every 2 milliseconds. These are the most precise outputs that the Model 4000 can provide. The OUT24 outputs and the POB outputs may also be controlled via the standard **OUT** statements during path execution, but with less accuracy in timing, because the desired output patterns are not part of the compiled path definition.

The path segment defined POB outputs are provided so that plotting applications may raise and lower the pen, laser cutters may turn the laser on and off, glue applicators may be turned on and off, all at prescribed positions along the path. The output specification is stated before the segment definition which holds that output state. In the example below, POB1 is on and the other have no change for the duration of the arc that follows.

Example

Statement	Description
`PATH POB 1XXX`	`'POB pattern for next segment.`
`PATH RCCW 500 500 500`	`'CCW quarter circle arc.`

Paths Built Using Model 4000 Statements

When using the Model 4000 statements to define a given path, the statements which specify all of the path definitions must be contained in a named block defining that path. Each path definition block has a unique name (not a label) which is used to distinguish one path from another. Because the path definition is stored as part of the user's program, many different paths may be stored, each defined with a unique name. A path definition block begins with a **PATH DEF** statement which contains its name, and ends with a **PATH END** statement. Each segment in a path may take up to 80 milliseconds to compile. As a result, multi-segmented paths may take quite long to compile. For this reason, the Model 4000 offers a statement to compile a named path definition block, and a separate statement to execute a named path. Once a named path is compiled, it may be executed repeatedly without delay.

A group of statements composing a path definition block may be executed by three different methods. These are the **PATH COMPILE** statement, the

PATH EXECUTE statement, and simply encountering a block as the next group of statements. For programming clarity, it is recommended that the **PATH COMPILE** statement, and the **PATH EXECUTE** statements are used. These statements essentially perform subroutine calls to the path definition block, requesting the block to be compiled or executed respectively. When these statements are used, the path definition blocks should be placed in a section of the program which is not part of the normal program flow. The path definition blocks do not need **LABEL** statements or **RETURN** statements when used in this manner.

It is also possible to simply encounter a path definition block via normal program flow. In this case, the path name specified by the **PATH DEF** statement is compared to an internal list of previously compiled path names. If the path had never been compiled, it will first be compiled, then executed. The statements following the path definition will then be executed. If the path had been previously compiled, it will be executed without being re-compiled. While this minimizes the delays associated with re-compiling a path each time it is to be executed, it brings up two other possible problems. One is the fact that the first execution includes a delay and that subsequent executions do not. This constitutes non-repeatable execution delay. The second involves the use of Q variables for path motion or position parameters. If a path parameter is specified with a Q variable, that variable is only read during the compile pass through the path definition block. If the variable changes, the Model 4000 will not know to automatically re-compile the path. The second execution of the path will be done using the old value of the Q variable specified path parameter, not the current value. For these reasons, it is advisable to use the **PATH COMPILE** and **PATH EXECUTE** statements rather than simply encounter a path definition block.

Non-path statements, (i.e. statements other than path definition statements) may be included within a path definition block. Because a path definition block may be accessed in two ways, (i.e., **PATH COMPILE** and **PATH EXECUTE**) the Model 4000 allows groups of non-path statements within a block to be conditionally included. Statements to be included (acted on) only during compile time may be preceded by the **PATH ONLY IF COMPILING** statement and followed by the **PATH ONLY ENDIF** statement. Statements to be included (acted on) only during execute time may be preceded by the **PATH ONLY IF EXECUTING** statement and followed by the **PATH ONLY ENDIF** statement. **PATH** statements may not be bracketed by **PATH ONLY IF** and **PATH ONLY ENDIF** pairs. Attempting to do so will result in an execute error. This feature is illustrated in the examples at the end the contouring description in this chapter.

Compiling the Path

A **PATH COMPILE** statement will cause the Model 4000 to find the named path definition block and compile the path described by those statements, even if that path name had been previously compiled. The use of Q variables as parameters in path definition statements allow the same basic path to be re-defined with slightly different sizes and shapes. They may also be used to conditionally include or omit sections of the path. The **PATH COMPILE** performs the equivalent of a **GOSUB** to the named path definition block. The Model 4000 executes the following statements in the path compiling mode until a **PATH END** statement is encountered. It then does the equivalent of a **RETURN**, allowing the statements following the **PATH COMPILE** statement to execute.

During the path compiling pass through the named definition block, all **PATH** statements are acted on in the order in which they are encountered. All non-path statements will be executed normally except those which are bracketed with **PATH ONLY IF EXECUTING** and **PATH ONLY ENDIF** statements. Those bracketed statements will be ignored. The Model 4000

was designed to allow run- time determination of path parameters. There may be cases when the Model 4000 should prompt the operator or host computer for the value to be used for path velocity or segment endpoints. Alternatively, these values may be read with the **IN DATA** or **IN IN24** statements, allowing multiple calls of a single subroutine to define similar path sections with different data values. Statements which retrieve this data would be bracketed by the **PATH ONLY IF COMPILING** and **PATH ONLY ENDIF** statements. This way, they would only be executed when they are needed for path definition, but ignored during path execution.

The endpoints for path segments may not be taught using **ENABLE TCH_JOG** or **ENABLE TCH_JOY**. Those statements affect only **MOVE** and **MOVI** statements. It is possible to teach the Model 4000 path parameters indirectly by using Q variables for those parameters and taking advantage of the **PATH ONLY IF COMPILING** statement. For example, if Q1 and Q2 were used to specify the X and Y endpoints of a line, the operator could jog the Model 4000 into position and read the positions into those variables. The example below illustrates this, assuming axis 3 and 2 to be X and Y respectively.

Example

Statement	Description
`PATH ONLY IF COMPILING`	`'Start of conditional block.`
`ENABLE JOG NO YES YES NO`	`'Enable jog on axes 2,3.`
`IN Q1 = POSITION OF AXIS3 MINC`	`'Q1 gets axis 3 motor position.`
`IN Q2 = POSITION OF AXIS2 MINC`	`'Q2 gets axis 2 motor position.`
`PATH ONLY ENDIF`	`'End of conditional block.`
`PATH LINE Q1 Q2`	`'LINE segment to (Q1,Q2).`

Executing the Path

A **PATH EXECUTE** statement will cause the Model 4000 to find the named path definition block and execute the path described by those statements, if that path name has already been compiled. If the path name specified by the **PATH EXECUTE** has not been compiled, the **PATH EXECUTE** statement will first compile the new path, then execute it. In this case, the first time a path is executed, it is delayed by the compilation time, but subsequent executions will have no delay. If it is important that there be no delay in even the first execution of a path, the **PATH COMPILE** statement should be executed first.

The use of Q variables as parameters in the path definition statement is a method of allowing segment parameters to take new values each time the path is compiled. When the path is executing, the values of the Q variables do not affect the path parameters. If a change in a Q variable value is intended to affect the path parameters, that path must be re-compiled. The **PATH EXECUTE** statement performs the equivalent of a **GOSUB** to the named path definition block. The Model 4000 executes the following statements in the **PATH EXECUTING** mode until a **PATH END** statement is encountered. It then does the equivalent of a **RETURN**, allowing the statements following the **PATH EXECUTE** statement to execute.

During the **PATH EXECUTING** pass through the named definition block, all PATH segment statements are executed in the order in which they were compiled. **PATH** segment statements are **LINE**, **RCW**, **RCCW**, **OCW**, and **OCCW**. All other **PATH** statement are ignored during the **PATH EXECUTING** pass, because the information contained in them has already been incorporated into the compiled path. When the Model 4000 encounters a **PATH** segment statement, it waits for that segment to finish its motion before going on to the next statement. This allows any non-path statement to be synchronized with the end of a segment in a path.

All non-path statements will be executed normally except those which are bracketed with **PATH ONLY IF COMPILING** and **PATH ONLY ENDIF** statements. Those bracketed statements will be ignored. The Model 4000 was designed to allow other actions during path execution. There will probably be cases in which outputs need to be set or other axes need to start motion along certain sections of the execution of the path, but not during the compilation of the path. Statements which command these actions would be bracketed by the **PATH ONLY IF EXECUTING** and **PATH ONLY ENDIF** statements. This way, they would only be executed during path execution, but ignored when the path is being compiled.

Synchronizing Non-Path Statements

As explained earlier, the **PATH ONLY IF** statements were intended to increase the flexibility of the use of non-path statements within a path. During compilation, non-path statements may be used to prompt for operator or computer input, or retrieve path parameters from **DATA** statements. These statements would be placed before the **PATH** statements affected by the retrieved data. During path execution, non-path statements may be used to set outputs, send messages, read inputs used for path related data collection, or initiate moves on other axes which must be coordinated with the path motion. These statements would be placed after the path segment statement whose endpoint specifies the desired path position for statement's action. This means that if non-path actions are required at certain path positions, the path segments must be designed so that the endpoints of segments occur at those positions.

The Model 4000 waits for the travel specified by a path segment to be finished before it executes the statement following that path segment. Under the best circumstances, the non-path statements will be executed within 2 milliseconds of the end of the previous path segment. Just as with the **PATH POB** statement, the precision is limited by the fact that the path position record is updated every 2 milliseconds. The precision of the synchronization of path and non-path statements is also dependent on the amount of background processing occurring during path execution. Background processing refers to display updates, polling for **ON** conditions, and servicing the communication interfaces. Use of the display takes the most time, and the precision degrades proportionally with the number of items being continuously displayed and the number of **ON** conditions enabled. It is possible for a non-path statement execution to be delayed by 10 to 20 milliseconds under these conditions.

Possible Programming Errors

The flexibility created by allowing separate Compiling and Executing passes through the path definition block brings with it the possibility for programming errors. The Model 4000 expects that the path segment statements encountered and waited on during the **PATH EXECUTE** pass will be exactly the same as those compiled during the **PATH COMPILE** pass. It is possible, unfortunately, to create a situation in which the segment statements during these two passes do not match. This could occur if conditional branching is used, (i.e. **IF ... GOTO**) or if enabled **ON** conditions cause either Compiling or Executing to miss part of the path. If conditional branching is used, great care must be taken to ensure that the conditions which would determine a branch are the same during **PATH COMPILE** as they are during **PATH EXECUTE**. **ON** condition statements also cause unexpected branches. Although it may be very useful to allow these interruptions, care should be taken that they do not result in an unexpected change in the way segments are encountered.

There is a significant exception to this general warning. It may be desirable to allow **ON FAULT** or **ON HLIMIT** to result in a **MOVE KILL** statement and

not return to the path. In this case, the motion of all participating axes will be stopped, and path execution will be terminated.

If part of the path compilation pass was missed but none of the path execution pass is missed, the resulting path will not execute as intended. Non-path statements will execute after waiting on path segments actually compiled, so non-path actions will be out of synchronization with path sections. At the end of the path, the Model 4000 will be waiting for one or more segments than were actually compiled, so the program will become stuck waiting for those segments. If part of the path execution pass is missed but none of the path compilation pass is missed, a similar error in synchronization will result. The Model 4000 will believe it has finished the path execution before the last segment actually finishes moving, and will proceed with the statements following the path.

Another possibility for problems can occur if a non-path statement which is executed during **PATH EXECUTING** requires the Model 4000 to **WAIT**. Any **WAIT** statement or any **MOVE** (not **MOVI**) statement could result in this. If the resulting **WAIT** requires more time than the currently executing path segment, statements intended to execute immediately following that path segment will execute late. This is not necessarily an error, but if the synchronization of path and non-path statements is important, this situation should be avoided. In the example below, the programmer intends for the **MOVE** on axis 3 and the **LINE** to begin together, and for the **OUT24** statement to be executed immediately after the **LINE** is finished. This will happen as intended as long as the **LINE** takes longer to execute than the axis 3 **MOVE**. If the axis 3 **MOVE** takes longer than the **LINE**, the **OUT24** statement will not execute until the **MOVE** is finished. If **MOVI** is used instead of **MOVE**, the program will always execute as intended.

Example

Statement	Description
`PATH RCW  100 100 100`	`'CW 3/4 circle arc.`
`MOVE * * Q1 *`	`'Start and wait for axis 3 move.`
`PATH LINE 0 200`	`'LINE segment to (0,200).`
`OUT OUT24 PATT1`	`'Set OUT24 to PATT1.`

Path Statement Summary

The following list of statements is intended to provide a summary of all the **PATH** statements which may be contained in a Model 4000 program. The statements are listed in an order which represents the way they would be found in a Model 4000 program. Each statement provides one example of syntax, and is accompanied by a very brief comment. Each statement is described in greater detail later in this chapter.

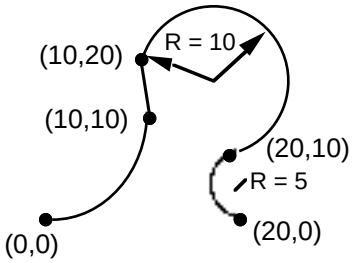
Example

Statement	Description
`UNIT PATH POS 25000`	`'X, Y, and R steps/unit.`
`UNIT PATH VEL 25000`	`'Path steps/second per vel unit.`
`UNIT PATH ACCEL 25000`	`'Path steps/sec2 per accel unit.`
`VEL PATH 5`	`'Path velocity in PATH VEL units.`
`ACCEL PATH 100`	`'Path accel in PATH ACCEL units.`
`DECEL PATH 200`	`'Path decel in PATH DECEL units.`
`PATH UNCOMP CONTOUR2`	`'Remove compiled CONTOUR2.`
`PATH COMPILE CONTOUR1`	`'Compile CONTOUR1 (not execute).`
`PATH EXECUTE CONTOUR1`	`'Execute compiled path CONTOUR1.`
`PATH DEF CONTOUR1  2 1 4 3`	`'Start CONTOUR1 definition.`
`PATH C_RES 125000`	`'Specify C axis resolution.`
`PATH P_RATIO 2.5`	`'Specify P axis ratio.`

```
PATH RAD_TOL .001                    'Radius tolerance in PATH POS units.
PATH XY MINC                         'Program in incremental coordinates.
PATH XY MABS                         'Program in absolute coordinates.
PATH XY WORK  10  50                 'Specify WORK coordinate system.
PATH XY LOCAL 5   3                  'Specify LOCAL coordinate system.
PATH ONLY IF COMPILING               'Start of conditional block.
PATH ONLY ENDIF                      'End of conditional block.
PATH ONLY IF EXECUTING               'Start of conditional block.
PATH POB 0011                        'POB bit pattern for next segment.
PATH LINE 4 7                        'Line segment to (4,7).
PATH RCCW  9 12 5                    'CCW arc to (9,12) radius 5.
PATH RCW 4 7 5                       'CW arc to (4,7) radius 5.
PATH OCCW 9 12 4 12                  'CCW arc to (9,12) center (4,12).
PATH OCW 4 7 4 12                    'CW arc to (4,7) center (4,12).
PATH END                             'End of CONTOUR1 definition.
```

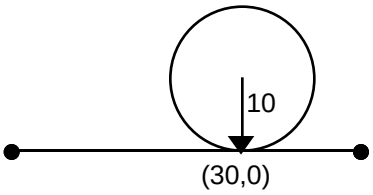
Programming Examples

The following figures show two simple paths which illustrate most of the Model 4000 segment types. For both figures, axis 3 is X, and axis 2 is Y. The C and P axes are not included. The first figure specifies the endpoints with absolute coordinates. The default Work coordinate system with start point of (0,0) is used, so no **PATH XY WORK** statement is needed. The second figure specifies the endpoints with incremental coordinates. The state of the POB outputs needs to be different for Handles than for Knobs. No other Model 4000 actions take place during these paths.



Example

Statement	Description
`PATH DEF HANDLE 3 2 NONE NONE`	'Begin HANDLE path definition.
`PATH XY MABS`	'Use absolute coordinates.
`PATH POB 1100`	'POB pattern for next segments.
`PATH OCCW 10 10 0 10`	'CCW quarter circle.
`PATH LINE 10 20`	'Vertical LINE segment.
`PATH RCW 20 10 -10`	'CW 3/4 circle.
`PATH RCCW 20 0 5`	'CCW half circle.
`PATH END`	'End of HANDLE path definition.



Example

Statement	Description
`PATH DEF KNOB 3 2 NONE NONE`	`'Begin KNOB path definition.`
`PATH XY MINC`	`'Use incremental coordinates.`
`PATH POB 0011`	`'POB pattern for next segments.`
`PATH LINE 30 0`	`'Long LINE into circular knob.`
`PATH OCCW 0 0 0 10`	`'CCW circle for the knob.`
`PATH LINE 10 0`	`'Short LINE out of knob.`
`PATH END`	`'End of KNOB path definition.`

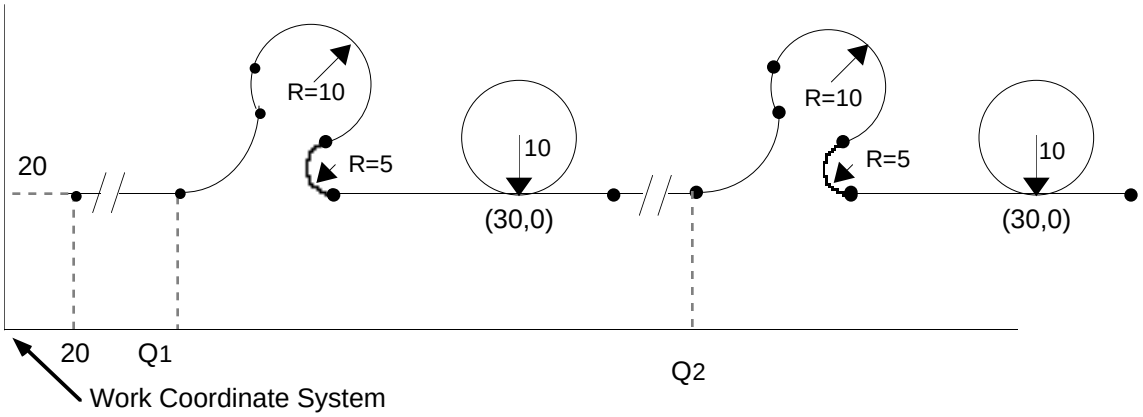
In the above example, the two paths are separate, each with their own name. We can create a new path by combining the segment statements of the two original paths. In the program example below, we've re-organized the program layout of Handle and Knob with subroutines. This allows us to use the segment statements for a third path named Parts. The program example below consists of three path definition blocks and two subroutines. Notice that the paths for Handle and Knob are unchanged. Notice also that the starting and ending directions of travel for both paths are the same. This allows them to be connected and does not result in an abrupt change in direction.

Example

Statement	Description
`PATH DEF HANDLE 3 2 NONE NONE`	`'Begin HANDLE path definition.`
`GOSUB HANDLE`	`'Statements describing HANDLE.`
`PATH END`	`'End of HANDLE path definition.`
`PATH DEF KNOB 3 2 NONE NONE`	`'Begin KNOB path definition.`
`GOSUB KNOB`	`'Statements describing KNOB.`
`PATH END`	`'End of KNOB path definition.`
`PATH DEF PARTS 3 2 NONE NONE`	`'Begin PARTS path definition.`
`LABEL PARTS`	`'LABEL for GOTO use.`
`PATH XY MABS`	`'Use absolute coordinates.`
`PATH XY WORK 20 20`	`'Establish WORK coordinates.`
`PATH LINE Q1 20`	`'LINE to (Q1,20).`
`GOSUB HANDLE`	`'Statements describing HANDLE.`
`GOSUB KNOB`	`'Statements describing KNOB.`
`PATH XY MABS`	`'Use absolute coordinates.`
`PATH XY WORK * *`	`'Return to WORK coordinates.`
`PATH LINE Q2 20`	`'LINE to (Q2,20).`
`GOSUB HANDLE`	`'Statements describing HANDLE.`
`GOSUB KNOB`	`'Statements describing KNOB.`
`PATH END`	`'End of PARTS path definition.`
`LABEL HANDLE`	`'LABEL for GOSUB use.`
`PATH XY MABS`	`'Use absolute coordinates.`
`PATH XY LOCAL 0 0`	`'Specify LOCAL coordinate system.`
`PATH POB 1100`	`'POB pattern for next segments.`
`PATH OCCW 10 10 0 10`	`'CCW quarter circle.`
`PATH LINE 10 20`	`'Vertical LINE segment.`
`PATH RCW 20 10 -10`	`'CW 3/4 circle.`
`PATH RCCW 20 0 5`	`'CCW half circle.`
`RETURN`	`'End of subroutine.`
`LABEL KNOB`	`'LABEL for GOSUB use.`
`PATH XY MINC`	`'Use incremental coordinates.`
`PATH POB 0011`	`'POB pattern for next segments.`
`PATH LINE 30 0`	`'Long LINE into circular knob.`
`PATH OCCW 0 0 0 10`	`'CCW circle for the knob.`
`PATH LINE 10 0`	`'Short LINE out of knob.`
`RETURN`	`'End of subroutine.`

Our third path consists of two pairs of the first two. Each pair is placed at variable locations within the Work coordinate system and the two pairs are connected with a Line segment. The line leading into the first pair starts at (20,20) in the Work coordinate system. The first pair starts at (Q1,20) and the second pair starts at (Q2,20) in the Work coordinate system. Handle is

defined using the Local coordinate system. Even though Handle is defined in absolute coordinates and appears in two different places along the path in Parts, the statements describing it appear only once, in a subroutine using local coordinates.



The path Parts is very simple now, but suppose that the path velocity needs to be different for Handles and Knobs, and that the operator wants to specify the velocity for each individual Handle and Knob. The program should prompt the operator for **PATH VELOCITY** during compile time, but these prompts should not appear while the path is executing. Also, axis 1 needs to make a short move when a Knob is started, but this move should not take place during path compilation. To be complete, some setup statements and the **PATH COMPILE** and **PATH EXECUTE** statements are shown first. Notice that even though all axis are specified as motor incremental, this does not affect the absolute coordinate specification for Handles.

Example

Statement	Description
`MODE M_INC M_INC M_INC M_INC`	`'All axes are motor incremental.`
`UNIT PATH POS 10000`	`'Path position steps/unit.`
`UNIT PATH VEL 10000`	`'Path steps/sec per unit.`
`UNIT PATH ACCEL 10000`	`'Path steps/sec2 per unit.`
`ACCEL PATH 50`	`'Path accel in path accel units.`
`OUT LCD3,01 ^COMPILING PARTS^`	`'Send message to line 3.`
`PATH COMPILE PARTS`	`'Compile the path PARTS.`
`OUT LCD3,01 ^MAKING PARTS ^`	`'Send message to line 3.`
`PATH EXECUTE PARTS`	`'Execute path PARTS.`
.	
.	
.	
`PATH DEF PARTS 3 2 NONE NONE`	`'Begin PARTS path definition.`
`LABEL PARTS`	`'LABEL for later GOTO use.`
`PATH XY MABS`	`'Use absolute coordinates.`
`PATH XY WORK 20 20`	`'Establish WORK coordinates.`
`PATH LINE Q1 20`	`'LINE to (Q1,20).`
`GOSUB ONEPART`	`'Statements for handle and knob.`
`PATH XY MABS`	`'Use absolute coordinates.`
`PATH XY WORK * *`	`'Return to WORK coordinates.`
`PATH LINE Q2 20`	`'LINE to (Q2,20).`
`GOSUB ONEPART`	`'Statements for handle and knob.`
`PATH END`	`'End of PARTS path definition.`
.	
.	
.	

```
LABEL ONEPART                          'LABEL for use with GOSUB.
PATH ONLY IF COMPILING                 'Start of conditional block.
IN Q1 = LCD2,01 ^HANDLE VEL?^          'Prompt for handle velocity.
VEL PATH Q1                            'Set path vel to user's value.
PATH ONLY ENDIF                        'End of conditional block.
GOSUB HANDLE                           'Statements describing HANDLE.
PATH ONLY IF COMPILING                 'Start of conditional block.
IN Q1 = LCD2,01 ^KNOB  VEL? ^          'Prompt for knob velocity.
VEL PATH Q1                            'Set path vel to user's value.
PATH ONLY ENDIF                        'End of conditional block.
PATH ONLY IF EXECUTING                 'Start of conditional block.
MOVI 25000 * * *                       'Start short move on axis 1.
PATH ONLY ENDIF                        'End of conditional block.
GOSUB KNOB                             'Statements describing KNOB.
RETURN                                 'End of subroutine.
```

To amplify on this example, suppose some of the Handles need a stripe of red paint on the outside surface, and that the paint is applied with a nozzle which is controlled by the POB output state defined at the beginning of the handle path. The nozzle needs to be located outside the path and always point in. This requires that axis 4 (the C axis) rotate the paint nozzle. The C axis rotates 360 degrees in 10000 steps. Our new path definition Redparts takes advantage of everything developed for Parts. Even though the path definition block Redparts shares most of its programs statements with the path definition Parts, and is identical in X and Y, the two have separate **PATH DEF** statements, and are separate and unique paths. As before, the path is compiled before it is executed. In this example, however, the C axis is also positioned before the path is executed.

Example

Statement	Description
`OUT LCD3,01 ^COMP. REDPARTS^`	'Send message to line 3.
`PATH COMPILE REDPARTS`	'Compile the path REDPARTS.
`OUT LCD3,01 ^MAKE REDPARTS ^`	'Send message to line 3.
`MOVE * * * HOMECW`	'Position C axis before path starts.
`PATH EXECUTE REDPARTS`	'Execute path REDPARTS.
.	
.	
.	
`PATH DEF REDPARTS 3 2 4 NONE`	'Define REDPARTS with C axis.
`PATH C_RES 10000`	'C axis resolution.
`GOTO PARTS`	'LABEL PARTS has remainder of path.

The following statements are designed to be used with the Model 4000 Contouring option.

ACCEL PATH

Name	**ACCEL PATH**
Descriptor	Set Linear Interpolation Path Acceleration
Type	Motion
Default	ACCEL PATH *
Syntax	ACCEL PATH 25000
Options	TAB Q NULL

F1	F2	F3	F4	F5	F6

Description

The **ACCEL PATH** statement specifies the acceleration used for a linear interpolation move and the contouring option. The path acceleration is the vector sum of the accelerations of the axis participating in the interpolation. The **ACCEL** parameters for preset moves are not used for the axis in an interpolated move. The **ACCEL PATH** statement uses the **UNIT PATH ACCEL** value when scaling the path acceleration.

See Also: **DECEL PATH**, **VEL PATH**, **UNIT PATH ACCEL**

Example

In the calculations below the path acceleration is 250,000. Using the distances given in the statement example, the individual accelerations for axes 1, 3, and 4 calculated by the Model 4000 are:

$$\text{Axis 1 ACCEL} = \left[\frac{25000}{\sqrt{((25000^2)+(200000^2)+(100000^2))}} \right] \text{X } (250000) = 27777.78$$

$$\text{Axis 3 ACCEL} = \left[\frac{200000}{\sqrt{((25000^2)+(200000^2)+(100000^2))}} \right] \text{X } (250000) = 222222.22$$

$$\text{Axis 4 ACCEL} = \left[\frac{100000}{\sqrt{((25000^2)+(200000^2)+(100000^2))}} \right] \text{X } (250000) = 111111.11$$

Axes #1, #3, and #4 will start moving, reach their peak velocity, start decelerating, and stop at the same time. Axis #2 will not participate in the linear interpolated move, it does a preset move of 50,000 steps.

DECEL PATH

Name	**DECEL PATH**
Descriptor	Deceleration for Interpolation Moves
Type	Motion
Default	DECEL PATH *
Syntax	DECEL PATH Q10
Options	TAB Q NULL

F1	F2	F3	F4	F5	F6

Description

The **DECEL PATH** statement specifies the **PATH** deceleration used in an interpolation move and the contouring option. For each linearly interpolated move, each axis' deceleration is computed based upon each axis distance component of that move. The **DECEL PATH** statement uses the **UNIT PATH ACCEL** value when scaling the path deceleration.

See Also: **ACCEL PATH**, **VEL PATH**, **UNIT PATH ACCEL**

Example

In the calculations below, the path velocity is 250,000. Using the distances given, the individual deceleration's calculated by the Model 4000 are:

$$\text{Axis 1 DECEL} = \left[\frac{25000}{\sqrt{((25000^2)+(200000^2)+(100000^2))}}\right] \quad X\ (250000) = 27777.78$$

$$\text{Axis 3 DECEL} = \left[\frac{200000}{\sqrt{((25000^2)+(200000^2)+(100000^2))}}\right] \quad X\ (250000) = 222222.22$$

$$\text{Axis 4 DECEL} = \left[\frac{100000}{\sqrt{((25000^2)+(200000^2)+(100000^2))}}\right] \quad X\ (250000) = 111111.11$$

Axes 1, 3, and 4 will start moving, reach their peak velocity, start decelerating, and stop at the same time. Axis #2 will not participate in the linear interpolated move, it does a preset move of 50,000.

See Also: **LINT, MOVE, MOVI**

PATH

Name	**PATH**
Descriptor	Define Path Taken
Type	Set-Up
Default	PATH
Syntax	PATH

Options

TAB	DEF	EXECUTE	END	COMPILE	ETC
TAB	RCW	RCCW	OCW	OCCW	ETC
TAB	LINE	TYPE	ONLY	POB	ETC
TAB	C_RES	P_RATIO	UNCOMP	RAD_TOL	ETC
F1	**F2**	**F3**	**F4**	**F5**	**F6**

Description

The **PATH** statements define and initiate circular interpolated motion. These are also known as contouring statements. They are functional with the Model's 4000 contouring option. The following is a summary of the contouring statements.

PATH DEF	Begin a path definition and specify the X,Y, and optional C or P axes.
PATH XY	Set line and arc statements' end point coordinate system to absolute or incremental.
PATH LINE	Define end point of a line segment.
PATH RCW	Define an arc segment via end point and radius, load travels CW.
PATH RCCW	Define an arc segment via end point and radius, load travels CCW.
PATH OCW	Define an arc segment via end point and center point, load travels CW.
PATH OCCW	Define an arc segment via end point and center point, load travels CCW.
PATH POB	Define a bit pattern to be sent to the four axis outputs at the beginning of the next arc or line segment.
PATH ONLY IF EXECUTING	Define place in path definition after which statements are only executed if **PATH EXECUTE** is in progress.
PATH ONLY IF COMPILING	Define place in path definition after which statements are only executed if **PATH COMPILE** is in progress.
PATH ONLY ENDIF	All statements are executed while both **COMPILE** and **PATH EXECUTE** are in progress. End of conditional statement execution.
PATH EXECUTE	Begin motion of path name specified. (If path has not previously been compiled then it will first be compiled, then executed.)
PATH COMPILE	Compile path name specified. This involves pre-calculating the line and arc statements which causes a delay of roughly 80 ms per line or arc.
PATH UNCOMP	Uncompile the path name specified. This allows other path(s) to be compiled. Used if the 500 arc or line limitation is reached.
PATH C_RES	Define the motor steps required to move the load on the C axis 1 revolution. This informs the Model 4000 of the gearing on the C axis.
PATH P_RATIO	Define the ratio of P axis velocity to the path velocity. This is also the ratio of the P axis distance traveled to the path distance traveled.
PATH RAD_TOL	Define the maximum error in radius calculations allowed. This error is measured when an arc statement is **COMPILED**.

PATH DEF

Name	**PATH DEF**
Descriptor	Define Path Taken
Type	Set-Up
Default	PATH DEF CONTOURØ x y c p
Syntax	PATH DEF BIG_CIRC 1 2 3 4
Options	TAB ALPHA FND_PATH
	F1 **F2** **F3** **F4** **F5** **F6**

Description

The **PATH DEF** statement begins the definition of a path for the Model 4000's contouring option. A path is defined as any combination of line, arc (and other statements) that result in continuous motion along a path. If motion must stop, another path must be defined and executed for motion to continue. Up to 100 paths may exist in their compiled form at one time. These 100 paths may contain up to 500 line and arc statements.

CONTOURØ is a default path name that can be modified using the **ALPHA** option or the numeric keys. Each path name must be unique, less than 9 characters long, and have a letter for the first character. The four numbers following the path name specify the X, Y, C, and P axes respectively. The X and Y axes must be specified, but the C and P axes are optional.

If numbers are not specified for C or P, it signifies that the C or P axis should not be included in that path definition. The axis specification for the entire path is done with this statement, and may not be changed within a path definition.

Following the **PATH DEF** statement there must be at least one **PATH LINE** or **PATH ARC** statement. The path definition is completed with a **PATH END** statement.

Example

Statement	Description
PATH DEF CIRCLE89 1 2	'Defines path CIRCLE89 and sets axis '1 & 2 to be the X and Y axis, 'respectively.

PATH EXECUTE

Name	**PATH EXECUTE**
Descriptor	Execute Path Taken
Type	Set-Up
Default	PATH EXECUTE CONTOURØ
Syntax	PATH EXECUTE CONTOUR8
Options	TAB ALPHA Q FND_PATH
	F1 **F2** **F3** **F4** **F5** **F6**

Description

The **PATH EXECUTE** statement is used in the Model 4000's contouring option to start execution of a previously compiled (defined) path. The **PATH EXECUTE** statement is like a **GOSUB** statement in that a branch is taken to the **PATH DEF** label and statements within the path are executed. When the **PATH END** statement is encountered, a return is done so that execution then continues with the statement following **PATH EXECUTE**.

If the path name specified has not been compiled, the path will first be compiled and then execution will start. By compiling the program before the **PATH EXECUTE** statement, the time delay associated with compiling the program may be isolated from path execution.

If any of the axes included in the specified path are not ready, an execution error will result and the program will stop. An axis is not ready if it is shutdown, moving in mode continuous, or in joystick mode. When path execution begins, all included axes become busy until path execution is finished.

Example

Statement	Description
`PATH EXECUTE CIRCLE89`	`'Create a circle with path named 'CIRCLE89. This will execute the 'path as if the path were a 'subroutine. Execution will then 'continue with line 2.`
`PATH EXECUTE CIRCLE89`	`'Create a second circle.`
`DONE`	`'Without a DONE statement we would 'hit the PATH DEF statement which 'would cause us to execute the path 'a third time.`
`PATH DEF CIRCLE89 1 2`	`'Defines path CIRCLE89 and sets axis '1 and 2 to be the X and Y axes.`
`PATH XY MABS`	`'Use absolute coordinates.`
`PATH OCCW 0 0 0 89`	`'Draw a circle of radius 89 in the 'CCW direction.`
`PATH END`	`'A path must finish with a PATH END.`

PATH END

Name	**PATH END**
Descriptor	End Path Taken
Type	Set-Up
Default	PATH END
Syntax	PATH END
Options	TAB
	F1 **F2** **F3** **F4** **F5** **F6**

Description

The **PATH END** statement completes the definition of a path for the Model 4000's contouring option. A path is defined as any combination of line, arc (and other statements) that result in continuous motion along a path. If motion must stop, then to continue motion another path must be defined and executed. Each path must start with a **PATH DEF** statement, have at least one line or arc statement, and finish with a **PATH END** statement.

The path definition is started with the **PATH DEF** statement, and remains under definition as segments are added. The **PATH END** statement tells the Model 4000 that the previous segment was the last segment, and that motion is to stop at the end of that segment. Only one path may be under definition at one time. If one path is currently under definition, the **PATH END** statement must be issued before another path may be defined.

Example

Statement	Description
`PATH DEF CIRCLE89 1 2`	`'Defines path CIRCLE89 and sets axis '1 and 2 to be the X and Y axes.`
`PATH XY MABS`	`'Use absolute coordinates.`
`PATH OCCW 0 0 0 89`	`'Draw a circle of radius 89 in the 'CCW direction.`
`PATH END`	`'A path must finish with a PATH END.`

PATH COMPILE

Name	**PATH COMPILE**
Descriptor	Compile Path Taken
Type	Set-Up
Default	PATH COMPILE CONTOURØ
Syntax	PATH COMPILE CONTOUR3
Options	TAB ALPHA FND_PATH

F1	F2	F3	F4	F5	F6

Description

The **PATH COMPILE** statement is used with the Model 4000's contouring option to do all the necessary calculations for a path before the path motion begins. The path name specified by the statement is a unique path. The name may have up to 8 characters and the first character must be a letter. Up to 100 paths can be entered and all may be compiled at the same time. The sum of all the line and arc statements contained in all the compiled paths must not be greater than 500. The number of line and arc statements that can exist in the 100 paths is limited only by program memory, and this number can be greater than 500. However, only 500 line and arc statements can be compiled at one time. The **UNCOMP** statement may be used to uncompile one path so that another path can be compiled.

The **PATH COMPILE** function has a delay of 25 to 80 milliseconds for each **PATH** line or arc statement within the path. This delay could be unwanted, so care should be taken to place **PATH COMPILE** statements in the program at places that are not time critical. These places in the program could be in the beginning of a program or in a **POWER_UP** program. A path does not have to be re-compiled unless it has been changed in some way. This allows the **PATH** to be executed within the program many times without the compile delay. A path needs to be re-compiled only if one of three situations exist.

① If a variable is used in a **PATH** statement and that variable has changed.

② If the current path must be uncompiled and a new path compiled because of the 500 line/arc limitation.

③ If branching statements such as **IF** or **ON** would cause the path to execute a different number of **PATH** statements during a subsequent path execution.

The path needs to be re-compiled so that the **PATH COMPILE** encounters the same number of **PATH** statements as the **PATH EXECUTE**. When a different number of **PATH** line or arc statements are encountered during **PATH EXECUTE** than were encountered during **PATH COMPILE**, errors will occur.

Example

Statement	Description
PATH COMPILE CIRCLE89	'Do the pre-calculations now so that the delay 'will not occur at the PATH EXECUTE.
PATH EXECUTE CIRCLE89	'Execute a circle with path named CIRCLE89. This 'will execute the path as if the path were a 'subroutine. Execution will then continue with 'line 2. Another PATH COMPILE is not needed.
PATH EXECUTE CIRCLE89	'Execute a second circle.
DONE	'If a DONE statement is not entered it would 'cause a third path to be 'executed.
PATH DEF CIRCLE89 1 2	'Defines path CIRCLE89 and sets axis 1 & 2 to be 'the X and Y axis, respectively.
PATH XY MABS	'Use absolute coordinates.
PATH OCCW 0 0 0 89	'Draw a circle of radius 89 in the 'CCW direction.
PATH END	'A path must finish with a PATH END.

PATH RCW

Name	**PATH RCW**
Descriptor	Define Endpoint and Radius of CW Arc
Type	Set-Up
Default	PATH RCW x y r
Syntax	PATH RCW Ø 5 2.5
Options	TAB Q NULL
	F1 F2 F3 F4 F5 F6

Description

The **PATH RCW** statement is used in the Model 4000's contouring option to specify a CW arc segment using endpoint X-Y coordinates and radius . The first two numbers following **RCW** are the endpoint X-Y positions or distances and the last number is the radius of the arc. The values are each scaled by the **UNIT PATH POS** value during **PATH COMPILE**. Executing the **PATH RCW** statement does not initiate motion. The placement, length, radius of curvature, and orientation of the arc are completely specified by the endpoint and radius specifications of the RCW statement and the endpoint of the previous segment. The direction of rotation in the X-Y plane will be CW. Segment endpoint position specifications may be either absolute with respect to the current coordinate system, or incremental, relative to the start of each individual segment.

Radius specifications are signed values. A positive radius specifies an arc which is less than 180°. A negative radius specifies an arc which is 180° or more.

If the end point and start point of an **RCW** are the same, a circle will not be created. The statement will be ignored. A more complete description of contouring, can be found at the beginning of this chapter.

Example

Statement	Description
`PATH DEF CONTOUR6 1 2`	'Defines path CONTOUR6 and sets axis '1 and 2 to be the X and Y axes.
`PATH XY MABS`	'Use absolute coordinates.
`PATH RCW 0 5000 2500`	'Defines a half circle arc segment 'to travel CW to endpoint 0,5000 in 'the X-Y plane.
`PATH END`	'A path definition is completed with 'a PATH END.

PATH RCCW

Name	**PATH RCCW**
Descriptor	Define Endpoint and Radius of CCW Arc
Type	Set-Up
Default	PATH RCCW x y r
Syntax	PATH RCCW 5 Ø 2.5
Options	TAB Q NULL
	F1 F2 F3 F4 F5 F6

Description

The **PATH RCCW** statement is used in the Model 4000's contouring option to specify a CCW arc segment using endpoint X-Y coordinates, and radius. The first two numbers following **RCCW** are the endpoint X-Y positions or distances and the last number is the radius of the arc. The values are each scaled by the **UNIT PATH POS** value during **PATH COMPILE**. Executing the **PATH RCCW** statement does not initiate motion. The placement, length,

radius of curvature, and orientation of the arc are completely specified by the endpoint and radius specifications of the **RCCW** statement and the endpoint of the previous segment. The direction of rotation in the X-Y plane will be CCW. Segment endpoint position specifications may be either absolute with respect to the current coordinate system, or incremental, relative to the start of each individual segment.

Radius specifications are signed values. A positive radius specifies an arc that is 180° or less. A negative radius specifies and arc that is 180° or more.

If the end point and start point of an **RCCW** are the same, a circle will not be created. The statement will be ignored.

Example

Statement	Description
`PATH DEF CONTOUR6 1 2`	`'Defines path CONTOUR6 and sets axis '1 and 2 to be the X and Y axes.`
`PATH XY MABS`	`'Use absolute coordinates.`
`PATH RCCW 0 5000 2500`	`'Defines a half circle arc segment 'to travel CCW to endpoint 0,5000 in 'the X-Y plane.`
`PATH END`	`'A path definition is completed with 'a PATH END.`

PATH OCW

Name	**PATH OCW**
Descriptor	Define Endpoint and Centerpoint of CW Arc
Type	Set-Up
Default	PATH OCW x y i j
Syntax	PATH OCW Ø 50 0 25
Options	TAB Q NULL
	F1 F2 F3 F4 F5 F6

Description

The **PATH OCW** statement specifies a CW arc segment using endpoint and centerpoint X-Y coordinates. The first two numbers following **OCW** are the endpoint X-Y positions or distances. The last two numbers are the center point X-Y distances relative to the start of the arc. The values are each scaled by the **UNIT PATH POS** value during **PATH COMPILE**. Executing the **PATH OCW** statement does not initiate motion. The placement, length, radius of curvature, and orientation of the arc are completely specified by the endpoint and center point specifications of the **OCW** statement and the endpoint of the previous segment. The direction of rotation in the X-Y plane will be CW. Segment endpoint position specifications may be either absolute with respect to the current coordinate system, or incremental, relative to the start of each individual segment. Center point position specifications are always incremental, relative to the start of the arc segment, even if a **PATH XY MABS** statement has been executed.

If the end point and start point of an **OCW** are the same, a circle will be created.

Example

Statement	Description
`PATH DEF CONTOUR6 1 2`	`'Defines path CONTOUR6 and sets axis '1 and 2 to be the X and Y axes.`
`PATH XY MABS`	`'Use absolute coordinates.`
`PATH OCW 0 0 0 8000`	`'Defines a arc segment to travel in 'a circle of radius 8000.`
`PATH END`	`'A path definition is completed with 'a PATH END.`

PATH OCCW

Name	**PATH OCCW**
Descriptor	Define Endpoint and Centerpoint of CCW Arc
Type	Set-Up
Default	PATH OCCW x y j k
Syntax	PATH OCCW
Options	TAB Q NULL

F1	F2	F3	F4	F5	F6

Description

The **PATH OCCW** statement is used in the Model 4000's contouring option to specify a CCW arc segment using endpoint and centerpoint X-Y coordinates. The first two numbers following **OCCW** are the endpoint X-Y positions or distances. The last two numbers are the center point X-Y distances relative to the start of the arc. The values are scaled by the **UNIT PATH POS** value during **PATH COMPILE**. Executing the **PATH OCW** statement does not initiate motion. The placement, length, radius of curvature, and orientation of the arc are completely specified by the endpoint and center point specifications of the **OCCW** statement and the endpoint of the previous segment. The direction of rotation in the X-Y plane will be CCW. Segment endpoint position specifications may be either absolute with respect to the current coordinate system, or incremental, relative to the start of each individual segment. Center point position specifications are always incremental, relative to the start of the arc segment, even if a **PATH XY MABS** statement has been executed.

If the end point and start point of an **OCW** are the same, a circle will be created.

Example

Statement	Description
PATH DEF CONTOUR6 1 2	'Defines path CONTOUR6 and sets axis '1 and 2 to be the X and Y axes.
PATH XY MABS	'Use absolute coordinates.
PATH OCCW 0 0 0 8000	'Defines an arc segment to travel in 'a circle of radius 8000.
PATH END	'A path definition is completed with 'a PATH END.

PATH LINE

Name	**PATH LINE**
Descriptor	Define Endpoint of a Line
Type	Set-Up
Default	PATH LINE x y
Syntax	PATH LINE 5 6
Options	TAB Q NULL

F1	F2	F3	F4	F5	F6

Description

The **PATH LINE** statement is used in the Model 4000's contouring option to specify a line segment. The first number specifies the position or the distance to travel on the X axis, and the second number specifies the position or the distance to travel on the Y axis. The values entered are scaled by the **UNIT POS** value during **PATH COMPILE**. The **PATH LINE** statement does not initiate motion of the line. The placement, length, and orientation of the line are completely specified by the endpoint of the line segment and the endpoint of the previous segment. Segment endpoint position specifications may be either absolute with respect to current coordinate system, or incremental, relative to the start of each individual segment.

Example

```
Statement                          Description
PATH DEF CONTOUR6 1 2              'Defines path CONTOUR6 and sets axis
                                   '1 and 2 to be the X and Y axes.
PATH XY MABS                       'Use absolute coordinates.
PATH LINE 25000 35888              'Defines a line segment to travel
                                   '25000 in the X and 35888 in the Y.
PATH END                           'A path definition is completed with
                                   'a PATH END.
```

PATH XY

Name	**PATH XY**
Descriptor	Set Path Mode for Endpoint Specification
Type	Set-Up
Default	PATH XY *
Syntax	PATH XY WORK Ø Ø
Options	TAB MINC MABS WORK LOCAL
	F1 F2 F3 F4 F5 F6

Description

The **PATH XY** statement is used in the Model 4000's contouring option to indicate that subsequent segment endpoints are specified in either incremental or absolute coordinates. **PATH XY MABS** is used to indicate absolute position endpoints will be used in subsequent path line and arc statements. **PATH XY MINC** is used to indicate incremental endpoints will be used in subsequent path line and arc statements. Using incremental endpoints means that each path line or arc is specified by the X and Y *distances from the start of the segment to the end of the segment.* Using absolute coordinates means that each path line or arc is specified by the X and Y *position of its endpoint in the current X-Y coordinate system.* Incremental programming can be inter-mixed with absolute programming without loss of the absolute coordinate system(s) defined.

There are 3 levels of coordinate systems available on the Model 4000. The over-all coordinate system is the machine coordinate system. This is the coordinate system to which the actual motor step positions refer during program execution. The **MODE M_ABS** statement enables absolute position programming (for **MOVE** and **MOVI**) statements within the machine coordinate system. The **PDEF** statement defines the machine coordinate systems origin.

When using absolute endpoint programming in path definitions, the machine coordinate system is not used, and may not be referenced or changed by **PATH** statements. When a path definition is started, incremental programming is the default, and a temporary coordinate system called the *Work* coordinate system is created for the duration of that path definition. To enter that work coordinate system in absolute mode, a **PATH XY MABS** statement must first be executed. A **PATH XY WORK Xpos Ypos** statement may then be executed to set the start position for the first line or arc statement. The default is **PATH XY WORK ØØ**.

In addition to the work coordinate system a local coordinate system may be defined. To enter a local coordinate system, execute the **PATH XY LOCAL Xpos Ypos** statement.

The local coordinate system allows a path section to use local absolute coordinates and be defined in a subroutine. This section may then be repeated multiple times within a path definition via a subroutine call. Once the local section is defined, it is possible to return to the work coordinate system for subsequent segment definitions. If the work coordinate system is being re-entered and you do not want to change the origin of the coordinate system, then use the **NULL** option for the Xpos and Ypos values. This would look like **PATH XY WORK * ***.

The work and local coordinate systems are only used to define the endpoints of arc and line statements. The center point of an arc is always an incremental value from the start of the arc. This is true both for X,Y,R specifications and for X,Y,J,K specifications of the center point.

At any point along a path definition, coordinates may be switched from incremental to absolute, or from absolute to incremental. To change to absolute coordinates, the **PATH XY MABS** statement must be executed. Then the **PATH XY WORK Xpos Ypos** or **PATH XY LOCAL Xpos Ypos** statements may be executed to locate and enter those coordinate systems.

The **PATH XY** statement does not affect the **DISPLAY** statements position report for either incremental or absolute position nor does it affect the **MODE** or **PDEF** statements parameters. Also, the **PDEF** and **MODE** statements do not affect path endpoint or center point specifications. The **MODE** statement is ignored for all axis involved in a path. All path axis are temporarily placed into motor step mode while involved in a path.

Example #1

Statement	Description
PATH XY MABS	'Enter absolute programming mode.
PATH XY WORK 0 0	'Sets a reference value of 0,0 for 'the X,Y endpoints for subsequent 'path line and arc statements. The 'center point is always an 'incremental value from the start of 'the current segment.

Example #2

This example shows how you can enter absolute mode, change to incremental mode and change back to absolute mode, several different ways.

Statement	Description
PATH DEF GOOD1 2 1 NONE NONE	'Start path definition. The default is 'incremental mode.
PATH RCW 5 0 2.5	'Endpoint is in incremental 'coordinates.
PATH XY MABS	'The default is WORK 0 0.
PATH RCW 0 0 2.5	'Endpoint is in work coordinates.
PATH XY MINC	'Now we go into incremental mode
PATH RCW 5 0 2.5	'Endpoint is in incremental 'coordinates.
PATH XY MABS	'Return to absolute endpoint specifications 'in the WORK coordinate system.
PATH XY LOCAL 0 0	'Start a new coordinate system without 'changing the previous work 'system.
PATH LINE 5 0	'Endpoint is in local coordinate 'system.
PATH OCW 15 0 5 0	'Endpoint is in local coordinate 'system.
PATH XY MINC	'Return to incremental coordinates.
PATH LINE -15 0	'Endpoint is in incremental system.
PATH OCW 10 0 5 0	'Endpoint is in incremental system.
PATH XY MABS	'Now return to the local coordinate 'system.
PATH XY WORK 0 0	'Return to work coordinates, and make 'current position the origin.
PATH OCCW 10 0 5 0	'Endpoint is in work coordinates.
PATH LINE -15 0	'Endpoint is in work coordinates.
PATH END	'Complete path definition.

Example #3

This example shows how to integrate the machine coordinate statements with the path coordinate systems.

Statement	Description
`MODE MABS MABS * *`	`'Set axis 1 and 2 to absolute 'positioning.`
`PDEF 0 0 0 0`	`'Define the current motor positions 'to be the origin in the machine 'coordinate system.`
`MOVE 12 15 * *`	`'Start the path at absolute position 'X=12, Y=15.`
`PATH EXECUTE GOOD1`	`'Do the path motion.`
`MOVE 12 15 * *`	`'Return to where the path started.`

PATH ONLY

Name	**PATH ONLY**
Descriptor	Begin or End Selective Execution of Statements
Type	Set-Up
Default	PATH ONLY IF *
Syntax	PATH ONLY IF EXECUTING
Options	TAB COMPILING EXECUTING ENDIF
	F1 F2 F3 F4 F5 F6

Description

The **PATH ONLY** statement is used with the Model 4000's contouring option to allow statements within a path to be executed either during a **PATH COMPILE** or during a **PATH EXECUTE**. This prevents the statement from being executed both during **PATH COMPILE** and **PATH EXECUTE**. For instance, an **IN** statement could be used to read in data from the LCD for use as an endpoint during a **PATH COMPILE**, but the IN statement delay would not be desirable during execution of the path. On the other hand, you may want to set an output using the **OUT OUT24** statement at the start of the **PATH**, but that may not desirable when the path is being compiled. The **ONLY IF EXECUTING** option causes the statements between that statement and the next **PATH ONLY ENDIF** statement to be executed only when the path is executing. The **ONLY IF COMPILING** option causes the statements between that statement and the next **PATH ONLY ENDIF** statement to be executed only when the path is compiling.

Example

Statement	Description
`PATH COMPILE BALL1`	`'Gosub to the PATH DEF BALL1 code 'to begin compile.`
`PATH EXECUTE BALL1`	`'Gosub to the PATH DEF BALL1 code 'to begin execute.`
`DONE`	
`PATH DEF BALL1 1 2`	`'Begin path definition.`
`PATH XY MABS`	`'Use absolute programming.`
`PATH ONLY IF COMPILING`	`'Execute following statements only 'during PATH COMPILE.`
`IN Q98 = LCD1,1 ^Enter circle radius^`	`'Wait for data to be entered.`
`PATH ONLY ENDIF`	`'Return to both compile & execute 'mode for lines or 'arcs.`
`PATH ONLY IF EXECUTING`	`'Execute following statements only 'during PATH EXECUTE`
`OUT OUT24 PATT19`	`'Send output pattern 19 to the 24 'outputs.`
`PATH ONLY ENDIF`	`'Return to both compile & execute 'mode for lines or arcs.`
`PATH OCW 0 0 0 Q98`	`'Define a circle of radius Q98, 'endpoint at starting point.`
`PATH END`	`'Complete PATH definition.`

PATH POB

Name	**PATH POB**
Descriptor	Set or Clear Outputs at Start of Path Line or Arc
Type	Set-Up
Default	PATH POB XXXX
Syntax	PATH POB Ø1Ø1
Options	TAB Q X

F1	F2	F3	F4	F5	F6

Description

The **PATH POB** statement is used in the Model 4000's contouring option to specify the output pattern which is to be applied to the 4 programmable outputs at the beginning of the next path line or arc statement. The outputs will remain in that state until changed by another **PATH POB** statement or an **OUT POB** statement. The **XXXX** specify axis outputs 1 thru 4, respectively. An **X** means ***don't change the state of the output.*** A 1 means set the output to 5V (current not flowing). A 0 means set the output to 0 volts (current flowing). Or, a variable can be substituted for the XXXX in which case all for outputs are set or cleared depending on the value of the variable. Add the following values to set the desired outputs to 5V:

8 ⇒output 1 (pin #5 of axis I/O connector 1) is set to 5V.
4 ⇒output 2 (pin #5 of axis I/O connector 2) is set to 5V.
2 ⇒output 3 (pin #5 of axis I/O connector 3) is set to 5V.
+1 ⇒output 4 (pin #5 of axis I/O connector 4) is set to 5V.
15

To set all 4 outputs with a variable at the time the **PATH RCW** executes, use the following:

```
MATH Q99 = 15
OUT POB Q99
PATH RCW 5 5 2.5
```

To change the state of the outputs at the end of a path, or outside of a path definition, the **OUT POB** statement must be used. The advantage of using the **PATH POB** statement over the **OUT POB** statement is that the **PATH POB** statement sets the outputs within a shorter and more repeatable time. The window is 1.5 ms before to 0.5 ms after the start of the next path line or arc statement. The **OUT POB** statement sets the outputs typically from 2 to 20 ms after executing the **OUT POB** statement, and is not as repeatable.

Example

Statement	Description
`MATH Q97 = 6`	`'Set variable Q97 to 6.`
`PATH POB Q97`	`'Sets the outputs at the start of 'the next path segment to the 'contents of variable Q97. Axis 'Outputs 1 and 4 are set to ØV '(current flowing) and outputs 2 and '3 are set to 5V (current not 'flowing).`
`PATH XX01`	`'Leave axes outputs 1 and 2 'unchanged, set axis 3 to ØV, and 'set axis 4V to 5V.`

PATH C_RES

Name	**PATH C_RES**
Descriptor	Set Number of Motor Steps per C Axis Rev
Type	Set-Up
Default	PATH C_RES *
Syntax	PATH C_RES 125000
Options	TAB
	F1 **F2** **F3** **F4** **F5** **F6**

Description

The **PATH C_RES** statement is used to specify the C axis resolution in the Model 4000's contouring option. The C axis will keep an angular position which changes linearly with the direction of travel implied by X and Y. This allows the C axis to control an object which must stay at a constant angle to the direction of travel.

The C axis resolution is the number of motor steps in 360 degrees of arc. C axis resolution does not necessarily equal the number of steps per revolution of the motor, but if the motor directly drove the rotating piece, then these numbers would be the same. If a 2 to 1 gear ratio were connected to a motor that has 25,000 steps per revolution, the **C_RES** would be 50,000.

The **PATH C_RES** statement should be given only once during a path definition, since the last **PATH C_RES** statement issued within a path definition will specify the C axis resolution for the entire path.

A negative value for the C axis resolution causes rotation in the negative direction as angle in the X-Y plane gets larger. Thus, if the motor is installed upside down or in such a way that the C axis does not track the path as desired, then changing the sign of the C resolution should correct the problem.

Example

Statement	Description

`PATH C_RES 75000`	`'Sets the resolution for the C axis`
	`'to be 75000 steps/rev. This would`
	`'be correct if a 3 to 1 gear ratio`
	`'is used with a 25,000 step per`
	`'revolution motor.`

PATH P_RATIO

Name	**PATH P_RATIO**
Descriptor	Set Ratio of (P Axis Velocity) (Path Velocity)
Type	Set-Up
Default	PATH P_RATIO *
Syntax	PATH P_RATIO 3.125Ø
Options	TAB Q NULL
	F1 **F2** **F3** **F4** **F5** **F6**

Description

The **PATH P_RATIO** statement is used to specify the P axis to path travel ratio in the Model 4000's contouring option. The P axis will travel a distance which is proportional to the distance traveled along the X-Y path. Also, the P axis will travel at a velocity which is proportional to the velocity traveled by the load along the X-Y path. The P axis motion occurs at the same time that the load is moved along the X-Y path. This proportionality constant is the **PATH P_RATIO**.

This allows the P axis to act as the Z axis in helical interpolation or to control the motion of any object which moves with distance and velocity proportional to the path. The **PATH P_RATIO** statement should be given only once during a path definition, since the last **PATH P_RATIO** statement issued within a path definition will specify the P axis travel to path travel ratio for the entire path. A negative value for the P axis ratio causes rotation

in the negative direction on the P axis as the path is traveled in the X-Y plane.

The maximum **P_RATIO** is 1,000. The default **P_RATIO** is 1.

Example

Statement	Description
`PATH P_RATIO 2.5`	`'Set the distance and velocity to be 'traveled by the P axis as 2.5 'times that traveled by the load 'along the path.`

PATH UNCOMP

Name	**PATH UNCOMP**
Descriptor	Uncompile a Path to Make Room for Another
Type	Set-Up
Default	PATH UNCOMP CONTOURØ
Syntax	PATH UNCOMP BIG_SHOE
Options	TAB ALPHA Q FND_PATH
	F1 F2 F3 F4 F5 F6

Description

The **PATH UNCOMP** statement is used with the Model 4000's contouring option to uncompile a path name specified to make room for another path.

The path name specified by the statement is a unique path. The name may have up to 8 characters and the first character must be a letter. Up to 100 paths may be in the compiled state at the same time as long as the total sum of line and arc statements within those 100 paths is less than 500. The number of line and arc statements that can exist uncompiled is limited only by program memory and this number can be greater than 500. The **UNCOMP** statement can be used to uncompile one path so that another path can be compiled.

Example

Statement	Description
`PATH COMPILE CONTOUR5`	`'Fill the 500 available path segments.`
`PATH EXECUTE CONTOUR5`	`'Execute 500 circles.`
`PATH UNCOMP CONTOUR5`	`'Make space available for another path to be 'compiled.`
`PATH COMPILE CONTOUR6`	`'Fill the 500 segments with a different paths line 'and arc statements.`
`PATH EXECUTE CONTOUR6`	`'Execute 500 different circles.`
`DONE`	`'If a DONE statement is not entered, it would 'cause a third path to be executed.`
`PATH DEF CONTOUR5 1 2`	`'Defines path CONTOUR5 and sets axis 1 and 2 to be 'the X and Y axes.`
`PATH XY MABS`	`'Use absolute coordinates.`
`LOOP 500`	`'Do 500 circles without stopping.`
`PATH OCCW 0 0 0 89`	`'Draw a circle of radius 89 in the CCW direction.`
`ENDLOOP`	
`PATH END`	`'A path must finish with a PATH END.`
`PATH DEF CONTOUR6 1 2`	`'Defines path CONTOUR6 and sets axis 1 and 2 to be 'the X and Y axes.`
`PATH XY MABS`	`'Use absolute coordinates.`
`LOOP 500`	`'Do 500 circles without stopping.`
`PATH OCW 0 0 0 54`	`'Draw a circle of radius 54 in the CW direction.`
`ENDLOOP`	
`PATH END`	`'A path must finish with a PATH END.`

PATH RAD_TOL

Name	**PATH RAD_TOL**
Descriptor	Set Allowable Error for Radius Value
Type	Set-Up
Default	PATH RAD_TOL*
Syntax	PATH RAD_T0L
Options	TAB Q NULL

F1	F2	F3	F4	F5	F6

Description

The **PATH RAD_TOL** statement is used to specify the allowable radius error that is encountered in the Model 4000's contouring option. The radius tolerance specified in the statement is scaled by the **UNIT PATH POS** value.

The radius error is encountered in one of two ways. The first way is through use of the **RCW** or **RCCW** statements. This error is the difference between the radius value specified in the **RCW** or **RCCW** statement and the minimum radius implied by the starting point and endpoint. If the radius provided in the statement is smaller than the minimum radius implied by the distance from starting to endpoints and the error is within the radius tolerance then just enough is added to the radius to make a half circle.

A second way to encounter a radius tolerance error is with the **OCCW** and the **OCW** statements. This error is the difference between the radius implied by the start point and center point and the radius implied by the endpoint and center point. If the difference in the two radius values is within the radius tolerance specified, then the center point is moved such that an arc can be traveled thru the start point and endpoint. The **PATH RAD_TOL** statement can be executed many times within a path definition allowing some arcs to be exactly known and others to be approximated.

The maximum **RAD_TOL** is 99,999,999 after scaling with **UNIT PATH POS**. The default **RAD_TOL** is 1.

Example

Statement	Description
`PATH DEF CONTOUR6 1 2`	`'Defines path CONTOUR6 and sets axis '1 and 2 to be the X and Y axis, 'respectively.`
`PATH XY MABS`	`'Use absolute coordinates.`
`PATH RAD_TOL 5`	`'Set the radius tolerance to 5 motor 'steps.`
`PATH RCCW 0 5000 2495`	`'Defines a half circle arc segment 'to travel counter CCW to endpoint '0,5000 in the X-Y plane. Note that 'the radius specified is short by 5 'steps, since this is within the 'tolerance give, the radius is ''automatically lengthened to 2500 'by the 4000.`
`PATH END`	`'A path definition is completed with 'a PATH END.`

UNIT PATH POS

Name	**UNIT PATH POS**
Descriptor	Set Scale Factor for Path Position Parameters
Type	Set-Up
Default	UNIT PATH POS *
Syntax	UNIT PATH POS
Options	TAB Q NULL

F1	F2	F3	F4	F5	F6

Description

The **UNIT PATH POS** statement scales the endpoint, center point and radius parameters of the **PATH** line and arc statements. Fractional values are allowed, but truncation errors may occur. Variables (**Q1 - Q99** or pointer variables **QQ1 - QQ99**) could also be used as the scale factor. The default value is 1. After scaling the **PATH** parameter with the **UNIT PATH POS** parameter, the result must not be larger than 99,999,999 or an execution error will result.

See Also: **PATH**

Example

Statement	Description
`PATH DEF LINECIRC 1 2`	'Start definition of a path with 'axes 1 and 2 being X and Y.
`UNIT PATH POS 125000`	'Set path position scale factor to '125000. This would be appropriate 'to allow programming PATH lines and 'arcs in inches/sec assuming a 25000 'step/rev motor drive and a 5 turn 'per inch table for both path axes 'are installed.
`UNIT PATH VEL 125000`	'Set path velocity scale factor to '125000. Subsequent VEL PATH 'parameters will be in inches per 'second (ips).
`VEL PATH 4`	'Define the path velocity to be 4 * '(the scale factor) steps per 'second. Resulting in 4 ips.
`UNIT PATH ACCEL 125000`	'Set path acceleration scale factor 'to 125000. Subsequent ACCEL PATH 'parameters will be in ips.
`ACCEL PATH 99`	'Define the path acceleration to be '99 * (the scale factor) 'steps/second squared. Resulting in '99 ips^2.
`PATH LINE 3 3`	'During PATH EXECUTE, move the load 'on the table 3 inches at 4 ips.
`PATH XY MINC`	'Define endpoint to be incremental 'from start of arc.
`PATH OCCW 0 0 0 25`	'During PATH EXECUTE, move the load 'in a circle of radius 25 inches.
`PATH END`	'Finish path definition.

Example

This example shows how to use variables to change the over-all dimensions of a contoured figure.

Statement	Description
`MATH Q80 = 1`	'Set a scale factor to scale the 'position scale factor.
`MATH Q81 = 125000`	'Set the position scale factor.
`MATH Q82 = Q81 * Q80`	'Set the variable used in the UNIT 'PATH POS statement.
`UNIT PATH POS Q82`	
`LABEL TRYAGAIN`	'Display F-KEY functions on LCD line '4.
`OUT LCD4,01`	

```
0 ^UP DOWN START^
ON F-KEY1 GOTO SCALE_UP                    'Increase Q80 scale factor so that
                                           'all PATH position parameters are
                                           'scaled up.
ON F-KEY2 GOTO SCALE_DN                    'Decrease Q80 scale factor so that
                                           'all PATH position parameters are
                                           'scaled down.
ON F-KEY3 GOTO START
GOTO TRYAGAIN
LABEL START                                'This section does the motion.
ON F-KEY1 DISABLE                          'Disable the F-KEY ON statements so
                                           'that the motion is 12ON F-KEY2
                                           'DISABLE is not interrupted.
ON F-KEY3 DISABLE
PATH COMPILE CIRCLE                        'Re-compile to account for any
                                           'changes made to scale factors.
PATH DEF CIRCLE 1 2                        'Start a path definition.
PATH OCCW 0 0 0 9                          'Define a circle of radius 9.
PATH END
GOTO TRYAGAIN                              'Go back and allow dimensions to be
                                           'adjusted.
LABEL SCALE_UP
MATH Q80 = Q80 + .01                       'Increase dimensions by 1%.
OUT LCD1,01 ^Overall scaling factor is: ^ Q80   'Display scaling factor on LCD.
GOTO TRYAGAIN                              'Go back and allow dimensions to be
                                           'adjusted.
LABEL SCALE_DN
MATH Q80 = Q80 - .01                       'Decrease dimensions by 1%.
OUT LCD1,01 ^Overall scaling factor is: ^ Q80   'Display scaling factor on LCD.
GOTO TRYAGAIN                              'Go back and allow dimensions to be
                                           'adjusted.
```

UNIT PATH VEL

Name	**UNIT PATH VEL**
Descriptor	Set Scale Factor for Path Velocity Parameters
Type	Set-Up
Default	UNIT PATH VEL *
Syntax	UNIT PATH VEL
Options	TAB Q NULL

F1	F2	F3	F4	F5	F6

Description

The **UNIT PATH VEL** statement scales the **VEL PATH** statements parameters. Fractional values are allowed, but truncation errors may occur. Variables (**Q1 - Q99** or pointer variables **QQ1 - QQ99**) could also be used as the scale factor. The default value is 1. After scaling the **VEL PATH** parameter with the **UNIT PATH VEL** parameter, the result must not be larger than 99,999,999 or an execution error will result.

See Also: **PATH**

UNIT PATH ACCEL

Name	**UNIT PATH ACCEL**
Descriptor	Set Scale Factor for Path Acceleration
Type	Set-Up
Default	UNIT PATH ACCEL *
Syntax	UNIT PATH ACCEL
Options	TAB Q NULL

F1	F2	F3	F4	F5	F6

Description

The **UNIT PATH ACCEL** statement scales the **ACCEL PATH** statements parameters. Fractional values are allowed, but truncation errors may occur. Variables (**Q1 - Q99** or pointer variables **QQ1 - QQ99**) could also be used as the scale factor. The default value is 1. After scaling the **ACCEL PATH** parameter with the **UNIT PATH ACCEL** parameter, the result must not be larger than 99,999,999 or an execution error will result.

See Also: **PATH**

Example

This following example shows how to use variables to change the overall dimensions of a contoured figure.

Statement	Description
`MATH Q80 = 1`	`'Set a scale factor to scale the 'position scale factor.`
`MATH Q81 = 125000`	`'Set the position scale factor.`
`MATH Q82 = Q81 * Q80`	`'Set the variable used in the UNIT 'POS statement.`
`UNIT PATH POS Q82`	
`LABEL TRYAGAIN`	`'Display F-KEY functions on LCD line '4.`
`OUT LCD4,01`	
`0 ^ UP DOWN START^`	
`ON F-KEY1 GOTO SCALE_UP`	`'Increase Q80 scale factor so that 'all PATH position parameters are 'scaled up.`
`ON F-KEY2 GOTO SCALE_DN`	`'Decrease Q80 scale factor so that 'all PATH position parameters are 'scaled down.`
`ON F-KEY3 GOTO START`	
`GOTO TRYAGAIN`	
`LABEL START`	`'This section does the motion.`
`ON F-KEY1 DISABLE`	`'Disable the F-KEY ON statements so 'that the motion is 12ON F-KEY2 'DISABLE is not interrupted.`
`ON F-KEY3 DISABLE`	
`PATH COMPILE CIRCLE`	`'Re-compile to account for any 'changes made to scale factors.`
`PATH DEF CIRCLE 1 2`	`'Start a path definition.`
`PATH OCCW 0 0 0 9`	`'Define a circle of radius 9.`
`PATH END`	
`GOTO TRYAGAIN`	`'Go back and allow dimensions to be 'adjusted.`
`LABEL SCALE_UP`	
`MATH Q80 = Q80 + .01`	`'Increase dimensions by 1%.`
`OUT LCD1,01 ^Overall scaling factor is: ^ Q80`	`'Display scaling factor on LCD.Go 'back and allow dimensions to be 'adjusted.`
`GOTO TRYAGAIN`	
`LABEL SCALE_DN`	
`MATH Q80 = Q80 - .01`	`'Decrease dimensions by 1%.`
`OUT LCD1,01 ^Overall scaling factor is: ^ Q80`	`'Display scaling factor on LCD.`
`GOTO TRYAGAIN`	`'Go back and allow dimensions to be 'adjusted.`

VEL PATH

Name	**VEL PATH**
Descriptor	Set Linear Interpolation Path Velocity
Type	Motion
Default	VEL PATH *
Syntax	VEL PATH 25000
Options	TAB Q NULL
	F1 F2 F3 F4 F5 F6

Description

The **VEL PATH** statement specifies the velocity used for a linear interpolated move and for Contour or Path definitions. The path velocity is the vector sum of the velocity of the axis participating in the interpolation.

Example

In the calculations below the path velocity is 250,000. Using the distances and **VEL PATH** given, the Model 4000 calculates the peak velocity for each axis. The peak velocity is reached, if the move is long enough for each axis. The Model 4000 makes the calculations shown below when the **MOVE** statement is executed. The **VEL PATH** statement is scaled by the **UNIT PATH VEL** parameter.

$$\text{Axis 1 VEL} = \left[\frac{25000}{\sqrt{((25000^2)+(200000^2)+(100000^2))}}\right] \text{X } (250000) = 27777.78$$

$$\text{Axis 3 VEL} = \left[\frac{200000}{\sqrt{((25000^2)+(200000^2)+(100000^2))}}\right] \text{X } (250000) = 222222.22$$

$$\text{Axis 4 VEL} = \left[\frac{100000}{\sqrt{((25000^2)+(200000^2)+(100000^2))}}\right] \text{X } (250000) = 111111.11$$

Axes 1, 3, and 4 will start moving at the same time, reach their peak velocity at the same time, start decelerating at the same time, and stop at the same time. Axis 2 will not participate in the linear interpolated move, it does a preset move of 50,000.

☛ NOTE: The Path velocity for a contour is evaluated when a Path is compiled. Changes in the **VEL PATH** will not effect the velocity of precompiled paths.

See Also: **MOVE**, **MOVI**

Example

Statement	Description
LINT MODE YES NO YES YES	'Axis #2 is not participating in LINT move
ACCEL PATH 250000	
DECEL PATH 250000	
VEL PATH 250000	
MOVE 25000 50000 200000 100000	'These are distances used in the above 'calculation

C H A P T E R ③

Multi-Tasking

Product Description

Multi-Tasking functionality is only present in the -CFM (or Following) option. The **TASK** statement has been added to allow the 4000 to run as many as 5 programs at the same time. Each concurrently executing program is called a task. The first task is started using the **RUN** command or the **START** command, just as in the standard Model 4000. Once a program is started, it can start other tasks using the **TASK** statement. Any of the 5 tasks are masters in the sense that each can start, stop and resume any of the 4 other tasks, although it is usually convenient to think of one task as the master when programming.

Starting Another Task

When editing on the front panel, a default name **PROG1** is provided as the program name to start or stop. If the options: **START**, **RESUME** or **STOP** are not present when the enter key is pressed, **START** is assumed. The **FND_PROG** option is provided over F6, this option allows the names of the programs already created to be recalled. The **FND_PROG** option will place previously defined program names where ever the cursor is located. After pressing **FND_PROG**, the up and down arrow keys then scroll through all the existing program names.

The **TASK** *program-name* **START** statement should only be executed once for each program you want to run as a separate task. If the same program is started a second time, and it is still running, an execution error will result. After starting another task, statement execution alternates among all the tasks. If a task is waiting for a **MOVE** to complete or an input to become active, the other tasks continue executing.

Stopping A Task

If a **TASK** *program-name* **STOP** is executed, it does not stop a **MOVE** in progress on that task. A **TASK** *program-name* **STOP** suspends the task from executing more statements, and it stops statements that were in progress. The **TASK** stop interrupts any **WAIT** statement, **IN LCD**, **IN PORT**, Jog or Joystick modes in progress, just as if an **ON** statement had become true. Because the **TASK STOP** does not stop motion, the task that executes the **TASK STOP** should stop the motion after the **TASK STOP** is executed. A better way to suspend a task is to use the combination of **MATH Qn = 1** in a

master task and the **ON Qn = 1** in a slave task to allow the master task to communicate to another task that it must do something. The example at the end of this discussion uses this combination to allow the main task to tell the other tasks to stop. An **ON** condition interrupt is better than the **TASK STOP** when using an **IN LCD** or **IN PORT** statement because the **IN** statements are terminated without error instead of just being suspended. A **MOVE STOP** should be executed to stop motion if that is the desired functionality.

Interaction Between Tasks

Many statements when executed affect the other currently running programs. The programs must be written to prevent one task from changing parameters of the other tasks. If the tasks are programmed to control different axes, then using the asterisk option to prevent changing another tasks' set-up parameters is one way to prevent problems. An example of this is the **UNIT POS 125000 * * ***, if the program controls only axis 1 then the other axes' parameters should not be set. Another way to prevent problems is to program all the tasks to use the same parameters so that they are not changed by other tasks. An example of this is **ENABLE ON NO** (Since it affects all tasks it should not be changed by different tasks.) The first word of the statements that can affect other tasks are:

ACCEL	**MODE**	**LINT**	**VEL**	**ENABLE**
DECEL	**POSM**	**SLIMIT**	**DEFINE**	**RS232**
ENCO	**SEG**	**UNIT**		

Keep all the various options these in mind when programming.

Since most statements execute in less than 2 milliseconds, the swapping of programs will not appear to slow the operation of any one program. However, there are a few statements that will slow the operation of all programs. These are: the **DISPLAY**, **ON**, **MATH SQRT**, **MATH** division, **MATH** multiplication, and **MATH** trigonometric statements. Also, there are statements which slow down only the programs that use them. These are: **OUT LCD**, **IN LCD**, **OUT PORT**, **IN PORT**, **MOVE**, **PATH COMPILE**, and **WAIT** statements. **ON** and **WAIT** statements are checked after all of the tasks execute a statement, but if no **ON** or **WAIT** statements are ever executed then they are not checked, and thus more processing time is available for the execution of statements.

If a statement can complete execution before the program must be swapped, then that statement will not slow down another program which executes that same statement. Statements which can not complete execution before the program must be swapped include: **MOVE**, **MOVI**, **PDEF**, **MODE**, **PATH**, **FOL**, **OUT LCD**, and the **IN LCD** statement. When more than one program execute these statements at the same time, then the later programs are delayed until that resource becomes available. For example, if one task executes a **MOVE** on axis 1, then any other **TASK**'s which also execute a **MOVE** on axis 1 will have to wait for the first **TASK**'s move to finish before they continue. But those other tasks could have **MOVE**s on axis 2, 3, and 4 and they would *not* have to wait for the move on axis 1 to finish.

Only one task may execute either an **IN LCD** or an **IN PORT** at one time, not one of each, one of either. Subsequent tasks that issue an **IN LCD** or **IN PORT** are suspended until the first task's **IN** statement is complete. Only one **TASK** may execute an **ENABLE JOG**, **ENABLE JOY** or be in Teach mode at one time. Subsequent **TASKS** are suspended until the first **TASK**s' **JOG** or **JOY** mode is complete, i.e., Multiple tasks using **JOY** and **JOG TEACH** mode can cause task suspension if they overlap each other.

Contouring

Contouring statements can be executed by one task **ONLY**. Two tasks can NOT execute contouring statements at the same time, even if they are commanding different axes.

Wait

All 5 tasks may execute any of the **WAIT** statement options at the same time. For instance, all 5 tasks can each issue a **WAIT FOR n SECONDS** statement with n different for each task: the tasks will wait independent of each other.

Interrupts

Only one of the 5 tasks may execute one of the approximately 98 individual **ON** statements at a time. If another task executes an **ON** statement that was already activated, then only the second task will be interrupted if the **ON** condition becomes true. An execution error is not generated when a second task executes the same **ON** as a previous task. All 5 tasks may execute **ON** statements, just not the same exact type. See the **ON** statement for all of the 98 individual **ON** types. The **ENABLE ON** statement enables or disables **ON** statement checking for all 5 tasks.

ON statements will interrupt the task that was running when they were initiated. If that task is **STOP**ed, and the interrupt then becomes true, the interrupt is ignored and disabled (after each **ON** becomes true it must be re-initiated to be active again). If the task is **RESUMED**, all **ON** statements previously initiated remain enabled. Keep in mind that if an **ON** condition becomes true but is not serviced because the task is **STOP**ed, and then a subsequent **ON** condition becomes true **ONLY** the last **ON** condition that became true is serviced.

Communication Between Tasks

Variables are global to all tasks. There are no variables that are local to any task. Variables are a good way to communicate between tasks. The **DEFINE VAR** statement allows the number and type of variables to be increased or decreased. An **ON Qn** (n= 1-10) statement will interrupt a task based upon a variable having a certain value.

Each **TASK** can **GOSUB** or **GOTO** many different programs. A **TASK** starts executing one program, but it may branch to another program. Tasks and programs are not correlated to the axes in any particular way. Any task or program can use the **MOVE**, **PATH**, **SEG** statements etc. All 5 tasks have a **GOSUB** stack, and **LOOP** stack. The **BREAK** statement clears the **GOSUB** and **LOOP** stacks for the task that executes it, it does not clear the **GOSUB** and **LOOP** stacks of the other programs.

The **DATA** statements are searched for starting with the first line of the program named in the **TASK** program name **START** statement. The **RESTORE** statement resets the **IN Qn = DATA** pointer for the task the issued it, not for the other tasks.

Memory Upgrade

An optional memory size of 256K bytes can be purchased. The 256K byte memory is only available with the -CFM software option and allows much larger program storage and variable storage. The **DEFINE VAR** statement allows up to 32,764 numeric and 32,763 string variables if the 256K memory is installed. This option can be purchased through Compumotor's Custom Product Group.

Multi-Tasking Example

This example waits for the user to press F-Keys 1 through 6 to cause moves on axis 1, 2 and 3. Also, the arrow up and down keys increase and decrease the velocity of the 4th axis. The following 5 example programs show how 5 programs can be run at the same time. The first program is the master program which is the program name to run using the **RUN** command on the front panel. When running the master program receives operator input from the front panel and then uses variables to control the other 4 programs.

The following statements are for the first task, the program is named **TASKMAST**

```
Q1 = 0                                    'Default to no moves commanded on axis 1.
Q21 = 0                                   'Default to no moves commanded on axis 2.
Q2 = 0                                    'Default to no moves commanded on axis 2.
Q22 = 0                                   'Default to no moves commanded on axis 2.
Q3 = 0                                    'Default to no moves commanded on axis 3.
Q23 = 0                                   'Default to no moves commanded on axis 2.
Q4 = 0                                    'Default to no velocity commanded on axis 4.
Q5 = 0                                    'Default to no velocity commanded on axis 4.
Q10 = 0                                   'Default to zero velocity on axis 4 slew move.
UNIT POS 25000 25000 25000 25000          'Setup scale factors for all axis
UNIT VEL 25000 25000 25000 25000
UNIT ACCEL 25000 25000 25000 25000
VEL 8 4 2 0
ACCEL 99 50 25 10
DECEL 50 75 50 50
GOSUB BLANK                               'Clear the LCD.
OUT LCD1,1 ^  THIS IS A MULTI-TASKING MODEL 4000    ^ 'Menu.
OUT LCD2,1 ^  Arrow up&down control axis 4 velocity.^
OUT LCD4,1 ^ START  STOP  START  STOP   START  STOP ^
'
TASK AXIS1 START                          'Start a second task running program: AXIS1
TASK AXIS2 START                          'Start a third task running program: AXIS2
TASK AXIS3 START                          'Start a fourth task running program: AXIS3
TASK AXIS4 START                          'Start a fifth task running program: AXIS4
'
LABEL BEGIN                               'Main loop starts here, display the menu on LCD.
OUT LCD3,1 BLANK
OUT LCD3,1 ^PART1 ^ Q1                    'Display commanded number of part 1.
OUT LCD3,12 ^^ Q21                        'Display completed number of part 1.
OUT LCD3,15 ^PART2 ^ Q2                   'Display commanded number of part 2.
OUT LCD3,25 ^^ Q22                        'Display completed number of part 2.
OUT LCD3,29 ^PART3 ^ Q3                   'Display commanded number of part 3.
OUT LCD3,38 ^^ Q23                        'Display completed number of part 3.
'
ON F-KEY1 GOTO START1                     'Enable F-key 1 interrupt to start axis 1 move.
ON F-KEY2 GOTO STOP1                      'Enable F-key 2 interrupt to stop  axis 1 move.
ON F-KEY3 GOTO START2                     'Enable F-key 3 interrupt to start axis 2 move.
ON F-KEY4 GOTO STOP2                      'Enable F-key 4 interrupt to stop  axis 2 move.
ON F-KEY5 GOTO START3                     'Enable F-key 5 interrupt to start axis 3 move.
ON F-KEY6 GOTO STOP3                      'Enable F-key 6 interrupt to stop  axis 3 move.
ON TIME = 1.25 GOTO BEGIN                 'After 1.25 sec goto begin to update the LCD.
LABEL WAITK                               'Program waits in this loop until a key is
                                          'pressed.
IF ARROWU GOTO INC_VEL                    'Check if arrow up key pressed to inc axis 4
                                          'vel.
IF ARROWD GOTO DEC_VEL                    'Check if arrow down pressed to dec axis 4 vel.
GOTO WAITK
'
LABEL START1
     MATH Q1 = Q1 + 1                     'Another task running program AXIS1 uses Q1 as a
     GOTO BEGIN                           'command counter of moves to make.
LABEL STOP1
     MATH Q1 = -1                         'Reset command counter of AXIS1 moves.
     WAIT FOR .5
     GOTO BEGIN
LABEL START2
     MATH Q2 = Q2 + 1                     'Another task running program AXIS2 uses Q2 as a
     GOTO BEGIN                           'command counter of moves to make.
LABEL STOP2
     MATH Q2 = -1                         'Reset command counter of AXIS2 moves.
     WAIT FOR .5
     GOTO BEGIN
LABEL START3
     MATH Q3 = Q3 + 1                     'Another task running program AXIS3 uses Q3 as a
     GOTO BEGIN                           'command counter of moves to make.
LABEL STOP3
     MATH Q3 = -1                         'Reset command counter of AXIS3 moves.
     WAIT FOR .5
     GOTO BEGIN
'
'
```

```
'
'
LABEL BLANK                                 'Clear all lines of the LCD.
     OUT LCD1,1 BLANK
     OUT LCD2,1 BLANK
     OUT LCD3,1 BLANK
     OUT LCD4,1 BLANK
RETURN
'
'
LABEL INC_VEL                               'Another task running program AXIS4 uses Q4 to
MATH Q4 = Q4 + 1                            'change the velocity of a slew move on axis 4.
GOTO BEGIN

LABEL DEC_VEL                               'Another task running program AXIS4 uses Q4 to
IF Q4 <= -1 GOTO BEGIN                      'If Q4 is already at -1, cannot be decreased.
MATH Q4 = Q4 - 1                            'change the velocity of a slew move on axis 4.
GOTO BEGIN
'
'
' The following statements are for the second task, the program is named AXIS1
'
ON Q1 = -1 GOTO STOP1                       'The main program sets Q1=1 when axis 1 should
                                            'stop.
IF Q1 > Q21 GOTO DO_MOVE1                   'Check if commanded number is greater than
                                            'completed.
GOTO BEG                                    'BEG is a reserved label name, the program jumps
                                            'to the first program statement.
LABEL STOP1
MATH Q1 = Q21                               'Set commanded number of part 1 to completed
                                            'number of part 1.
MOVE STOP * * *                             'Stop move on axis 1 only.
GOTO BEG
LABEL DO_MOVE1
MATH Q21 = Q21 + 1                          'Increase counter of completed part 1 parts.
MOVE 2 * * *                                'Now start making part 1.
WAIT FOR .1
MOVE 3.2 * * *
WAIT FOR .1
MOVE 4.4 * * *
WAIT FOR .1
MOVE 0 * * *
GOTO BEG                                    'Now go back to beginning and see if there any
                                            'more to make.
'
'
' The following statements are for the third task, the program is named AXIS2
'
ON Q2 = -1 GOTO STOP
IF Q2 > Q22 GOTO DO_MOVE
GOTO BEG
LABEL STOP
MATH Q2 = Q22                               'Set commanded number of part 2 to completed
                                            'number of part 2.
MOVE *  STOP * *
GOTO BEG
LABEL DO_MOVE
MATH Q22 = Q22 + 1
MOVE * 4 * *
WAIT FOR .1
MOVE * 3.2 * *
WAIT FOR .1
MOVE * 2.4 * *
WAIT FOR .1
MOVE * 0 * *
GOTO BEG
'
'
' The following statements are for the 4th task: program named AXIS3
'
ON Q3 = -1 GOTO STOP
IF Q3 > Q23 GOTO DO_MOVE
GOTO BEG
LABEL STOP
MATH Q3 = Q23                               'Set commanded number of part 3 to completed
                                            'number of part 3.
MOVE * * STOP *
GOTO BEG
LABEL DO_MOVE
MATH Q23 = Q23 + 1
MOVE * *  5 *
WAIT FOR .1
MOVE * * 6.2 *
WAIT FOR .1
MOVE * * 5.4 *
```

```
WAIT FOR .1
MOVE * * 0  *
GOTO BEG
'
'
' The following statements are for the 5th task: program named AXIS4
'
MATH Q5 = Q4
Q10 = 0
LABEL MAIN
ON Q4 = -1 GOTO STOP
IF Q4 > Q5 GOTO DO_INC
IF Q4 < Q5 GOTO DO_DEC
GOTO MAIN
LABEL STOP
MOVE * * * STOP
MATH Q4 = 0
GOTO BEG
LABEL DO_INC
IF Q10 >= 20 GOTO TOO_HIGH                'Don't allow any further increase beyond max.
Q5 = Q5 + 1                               'Increase local velocity change counter.
MATH Q10 = Q10 + .2                       'Increase axis 4 velocity by .2 rps.
IF Q10 <= 20 GOTO VEL_OK
MATH Q10 = 20                             'Set to maximum possible.
LABEL VEL_OK
VEL * * * Q10
MOVI * * * SLEWCW
GOTO MAIN
LABEL DO_DEC
IF Q10 <= 0 GOTO TOO_LOW                  'Don't allow any further decrease below min.
Q5 = Q5 - 1                               'Decrease local velocity change counter.
MATH Q10 = Q10 - .2                       'Decrease axis 4 velocity by .2 rps.
IF Q10 >= 0 GOTO VEL_OK2
MATH Q10 = 0                              'Set to minimum possible.
LABEL VEL_OK2
VEL * * * Q10
MOVI * * * SLEWCW
GOTO MAIN
LABEL TOO_HIGH
LABEL TOO_LOW
MATH Q4 = Q5
GOTO MAIN
```

Multi-Tasking—Statements

This statement is designed to be used with the Multi-Tasking Option.

TASK

Name	**TASK**
Descriptor	Control other TASKs
Type	Set-Up
Default	TASK PROG1
Syntax	TASK PROG1 START

Options	TAB	RESUME	START	ALPHA	STOP	FND_PROG
	F1	**F2**	**F3**	**F4**	**F5**	**F6**

Description

The **TASK** statement causes another program to start running at the same time the program that executes it runs. The program name to start running is specified in the **TASK** statement. Once another program is started, it is called another task. Five **TASK**s can run at the same time.

The **TASK** statement can also stop and resume **TASK**s that have been started. To stop a **TASK**, place **STOP** after the program name. The **STOP** parameter only stops the **TASK** specified from executing more statements, it does not stop statements in progress. For example, if a **MOVE** was in progress, the **TASK program name STOP** does not stop the move. Also, if a **TASK program name RESUME** is then issued, it does not re-execute the move, and it does not execute a **MOVE RESUME**.

When editing on the front panel, the **FND_PROG** option is provided over F6. This option recalls program names and replaces PROG1 with them. The

cursor should be under PROG1 before the **FND_PROG** option is used. The **ALPHA** option is used to create new program names.

Example 1

This example shows how two tasks can command moves on different axes at the same time.

```
TASK PROG2 START              'Program named prog2 starts running now as a
                              'second TASK
MOVE 5000 * * *               'Start a move on axis 1
OUT LCD1,1^AXIS1 DONE^        'When finished moving display message to LCD1
```

Example 2

Program PROG2

```
MOVE *1000 * *            'Start a move on axis 2
OUT LCD2,1^AXIS2 DONE^    'When finished moving display message to LCD2
```

DISPLAY ON PORTn TRACE

Name	**DISPLAY ON PORTn TRACE**
Descriptor	Display Program Statements
Type	Set-Up
Default	DISPLAY ON PORTn TRACE
Syntax	DISPLAY ON PORT1 TRACE

Options	TAB	PORT	LNUM	OFF	LCD	
	F1	**F2**	**F3**	**F4**	**F5**	**F6**

Description

To help debug programs, the **DISPLAY** statement has been expanded. To see a program trace for each task on the LCD, use the **DISPLAY ON LCDn TRACE** statement. Place a **DISPLAY ON LCDn** statement in each program that is multi-tasking, and use a different LCDn number in each program. The 4000 will display only that tasks statements on the display as they are executed. Not every statement is displayed on the LCD when executed because that would slow down program execution to about 5% of the speed of program execution without the LCD Trace option. Instead, the 4000 samples each tasks' program execution every 250 milliseconds and displays whatever statement was executing at that time.

Another version of program trace mode is the **DISPLAY ON PORTn TRACE** statement. Each task that executes the **DISPLAY ON PORTn** statement will have its statements displayed to the port specified as they are executed. So that each tasks' statements can be distinguished, a Tn where n= 1 - 5 is displayed just in front of each statement, that allows all the tasks to specify a single port to which to have the strings displayed. A difference between the LCD trace mode and the **PORT** trace mode is that program execution is slowed down much more in **PORT** trace mode. This is because every statement that executes is displayed to the port, not just those sampled every 250 milliseconds. Because serial communications take about 40 milliseconds per statement, and normal statement execution take 2 to 3 milliseconds to execute, the program is slowed to about 5% of the normal execution rate.

DEFINE ON RET

Name	**DEFINE ON RET**
Descriptor	Configure how an interrupt returns
Type	Set-Up
Default	DEFINE ON RET *
Syntax	DEFINE ON RET YES

Options	TAB	YES	NO	NULL		
	F1	**F2**	**F3**	**F4**	**F5**	**F6**

Description

Another statement added is **DEFINE ON RET**. This statement only affects **RETURN** statements that are returning from an **ON GOSUB** (the **RETURN** from an interrupt destination). This statement has no affect on **GOSUB** or **IF GOSUB** statements. This statement tells a **RETURN** statement whether to return to the statement interrupted or the statement following the one interrupted. The default for this statement is **NO**: return to the statement following the one interrupted. This statement remains in effect until it is disabled with the **NO** option or power is cycled. All **TASKS** and programs are affected by it. The statement can be executed within an interrupt service routine. This allows the return destination from that interrupt to be configured by the interrupt.

This statement can be used to return to re-execute a **WAIT** or **IN** statement but probably would not be used to return to a **MOVE** statement (unless the **MODE** was **M_ABS** or **E_ABS** so that no additional move would be commanded).

ENABLE TRIG REV

Name	**ENABLE TRIG REV**
Descriptor	Enable inversion of active trigger level
Type	Set-Up
Default	ENABLE TRIG REV * * * *
Syntax	ENABLE TRIG REV YES * * NO

Options	TAB	YES	NULL	NO		
	F1	**F2**	**F3**	**F4**	**F5**	**F6**

Description

The **ENABLE TRIG REV** statement enables or disables the inversion of the active state of the specified trigger input. The power-up default for all four trigger inputs is active high, i.e, a transition from a low to a high TTL level is required for activation of the functions associated with a trigger. These functions may include registration and several functions associated with Following. Refer to the **Model 4000 Options Guide** for a description of these Following functions. If the active level of a trigger is inverted with the **ENABLE TRIG REV YES** statement, a transition from a high to a low TTL level is required for activation of the functions associated with that trigger.

See Also: **SEG REG**, **ENABLE REG**, **DEFINE TRIGDB**, **FOL**, **FOLM**

ENABLE REGSRV

Name	**ENABLE REGSRV**
Descriptor	Enable registration compensation of servo following error
Type	Set-Up
Default	ENABLE REGSRV * * * *
Syntax	ENABLE REGSRV YES * * NO
Options	TAB YES NULL NO
	F1 F2 F3 F4 F5 F6

Description

The **ENABLE REGSRV** statement enables or disables servo following error compensation in registration moves. When a Model 4000 is controlling a servo drive axis, that axis is usually in Motor Step mode. This allows the drive to close its own position control loop. While in motion, however, there will often be drive following error, i.e., the actual load position may not match the commanded position. In registration applications, it is the actual load motion which triggers the registration input. When a Motor Step mode registration move input occurs, the Model 4000 normally issues the additional commanded steps to the drives, as specified in the registration move definition. If drive following error exists when the input occurs, this error becomes static overshoot after the move finishes and the drive corrects its following error.

This problem may be avoided by using the registration compensation feature of the Model 4000. To use this feature, the actual position of the load (or shaft) must be available to the Model 4000 through the encoder input of that axis, and the **ENCO MRES** and **ENCO ERES** values must be correctly specified. Although the encoder position is not used to *control* the move profile, it is captured along with the motor step position upon receipt of the registration input. If servo following error compensation is enabled (**ENABLE REGSRV YES**), these values are used along with the motor and encoder resolutions to calculate the servo drive following error in terms of motor steps. The registration moves (Motor Step mode only) are adjusted by this error to result in correct positioning of the load after the drive has corrected its following error. The *reference* encoder position is captured when motion begins, so *it important that the load be stable and in position before the next move which results in registration actually starts.*

See Also: **SEG REG**, **ENABLE REG**, **MODE**, **ENCO MRES**, **ENCO ERES**

DEFINE TRIGDB

Name	**DEFINE TRIGDB**
Descriptor	Define trigger debounce time
Type	Set-Up
Default	ENABLE TRIGDB* * * *
Syntax	ENABLE TRIGDB 80 * Q1 *
Options	TAB Q NULL
	F1 F2 F3 F4 F5 F6

Description

The **DEFINE TRIGDB** statement defines the the total debounce time for the four trigger inputs. The debounce prevents noise or mechanical switch bounce from causing a false interrupt on the trigger input. A trigger is initially recognized on the rising edge of the input. That trigger will not be recognized again until it has gone low again and the debounce time, measured from the rising edge, has been exceeded. This debounce time

affects registration and all of the **FOL** and **FOLM** statements which include triggers as a parameter. The initial value of 80 milliseconds will usually be long enough to debounce most mechanical and electronic switches, but this time may be lengthened if needed. In some applications, registration marks or master/slave synchronization marks may occur more frequently than 80 milliseconds. In these cases, the debounce time may be shortened, provided the signal *bounce* is short enough. The debounce times are only accurate to ±2 milliseconds of the specified value, and the actual values used will always be between 4 and 1000 milliseconds. The debounce times are specified for triggers 1, 2, 3, and 4 left to right on the statement line.

See Also: **FOL MOVEWT**, **FOL NEWCYC**, **FOL WAIT**, **FOL M_SYNC**, **FOL S_SYNC**, **FOLM DEF**

C H A P T E R ④

Following

Product Description

The Following option with the Model 4000 provides to users the functionality and programmability to solve applications requiring encoder or motor step ratioing. This section of the manual is designed to highlight the capabilities of the Following option with useful descriptions and examples, however, if you need more details on the operation or syntax of a particular command, please refer to the *Statements* section of this document.

For all the functionality, following with the 4000 can be categorized into one of three main application types: Ratio Following, Cam Profiling, and Moving Positioning System (MPS). Ratio following includes simple concepts such as an electronic gearbox, trackball, slave feed-to-length, as well as complex changes of ratio as a function of master position. Cam profiling allows rapid execution of predefined complex profiles. The moving positioning system allows users to super-impose standard positioning moves, like point-to-point or contouring, on top of ratio following.

Ratio following can include continuous, preset, and registration-like moves in which the velocity is replaced with a ratio. The slave may follow in either direction and change ratio while moving, with phase shifts allowed during motion at otherwise constant ratio. Ratio changes or new moves may be dependent on master position or based on receipt of a trigger input. Also, a slave axis may perform following moves or normal time-based moves in the same application because following can be enabled and disabled at will. In ratio following, acceleration ramps between ratios may be either time-based or dependent upon a specified master distance. Product cycles can be easily specified with the master cycle concept.

Complex applications such as coil winding and cam profiling can be solved with the definition of slave profiles which are precompiled. By predefining the profiles, the requirement for real time execution of statements is eliminated. This allows complex profiles with frequent ratio changes to be run at very high rates.

The moving positioning system allows familiar move functions to be performed on moving targets, but programmed as if the targets were stationary. For example, if a conveyor belt carries trays of parts which are to be unloaded, the 4000 can detect and track motion of the tray as it performs pick and place operations on those parts. The stationary reference is also maintained, so the parts can be placed on a stationary target. The pick and place moves on the tray are programmed using positions on the tray

and velocities with respect to the tray. In a similar fashion, complex contours can be applied to a part on a moving production line using standard **PATH** definition and execution.

Before delving into the specifics of ratio following, cam profiling, and moving positioning system, lets take a look at how the Model 4000 follows. First of all, what can be a master, and what can be a slave? The 4000 allows standard incremental or absolute encoder input, as well as the commanded step output of an axis to act as the master signal. Any axis can be slaved to the encoder input or motor output of any other axis. Also, an axis may be slaved to its own encoder input, as long as that axis does not need encoder feedback for encoder positioning or stall detection. Up to 4 axes can be following at the same time with the same or different inputs as the master signals. For more information on assigning a master for a particular slave, refer to the **FOL MASTER** statement.

Technical Overview

When a slave is following a master, the 4000 does not simply measure the master velocity to derive slave velocity. Instead, the Model 4000 samples master position every two milliseconds and corresponding slave setpoint positions are calculated. This is true even if the slave is in the process of changing ratios. A slave is not simply following velocity, but rather position. With this algorithm, the master and slave position or phase relationship is maintained indefinitely, without any drift over time due to velocity measurement errors.

The 4000 also measures master velocity by measuring the change in master position over a number of sample periods. The present master velocity and position may be used to calculate the next commanded slave position, so the slave has no velocity dependent phase delay. This concept is known as Velocity Feed Forward and may be enabled or disabled as needed.

The 4000 even allows the user to change the default following algorithm if their application requires. By default, the master's velocity is calculated over two sample periods (4 msec), and for most applications this will prove to work very well. Suppose, however, that the speed of the master is very slow, or has some vibration. For a case like this, the 4000 allows the user to extend the number of sample periods used to calculate the master velocity. This is known as Velocity Smoothing and more information can be found in the section titled Velocity Smoothing and the **FOL SMOOTH** statement.

The major points of emphasis in the design of the Model 4000's following features are its ability to maintain the programmed phase relationships between master and slave, and easy to use statements for the design of precise motion profiles.

More details on the technical aspects of following with the 4000 can be found at the end of this chapter, in the section titled Technical Considerations.

Installation Instructions

Use the steps below to properly install the -CFM option into your Model 4000 if you have purchased the option separately. If you purchased the following option factory installed, proceed to Ratio Following.

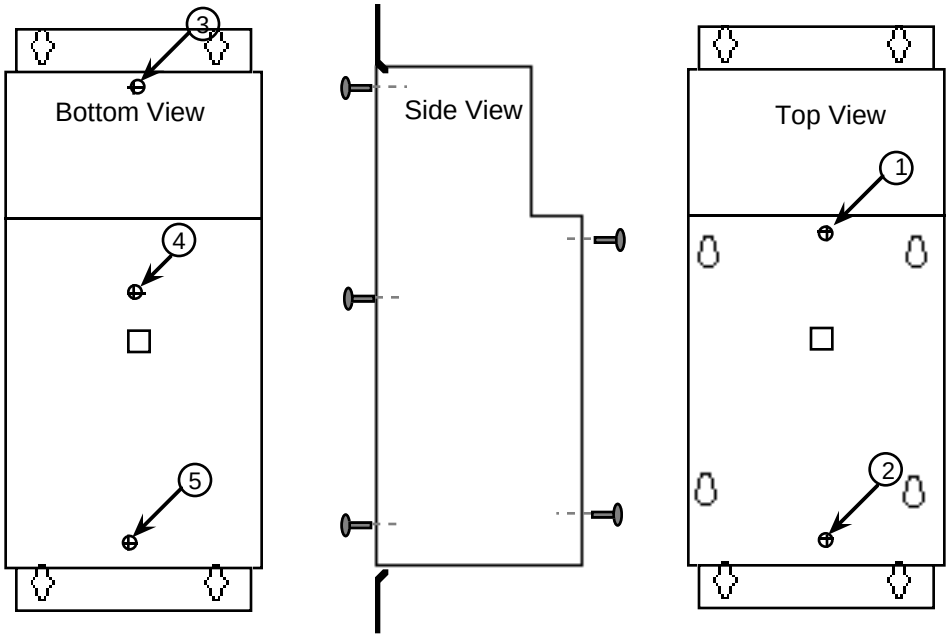
☛ These devices are sensitive to static discharge.

A grounding strap should be worn when performing this installation. If you do not have a grounding strap available you may discharge any buildup of static by touching a grounded piece of metal before opening the Model 4000.

Step ①

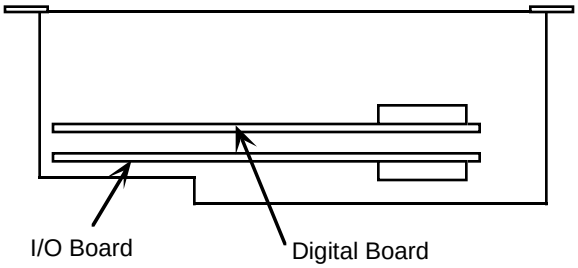
Remove AC Power.

To open the Model 4000 enclosure you must disconnect the phone cord and remove screws 1 through 5. Slide internal assembly off, by pushing on the fan side and remove completely.



Step ②

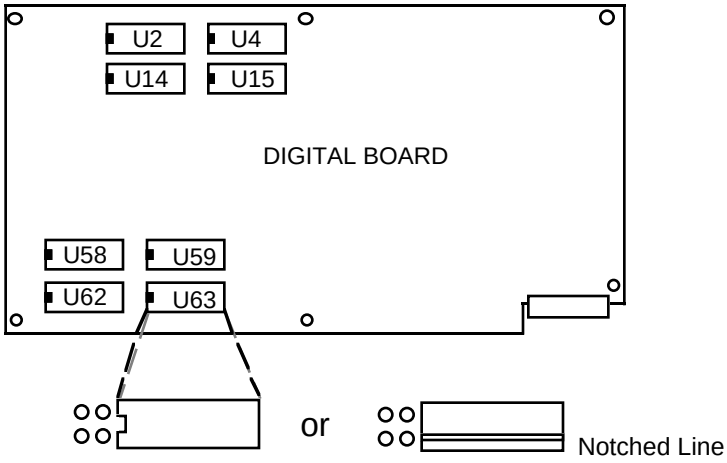
Refer to the following tables to determine the IC's you will need to remove. Carefully turn unit over and remove the appropriate IC's with a small screwdriver between the IC and the socket.



Use the following table if the sticker on the outside sheet metal of your unit has a serial number greater than or equal to 91-1114XXXXX. The four unmarked IC's are all interchangeable and go in slots U14, U15 U58, and U59.

IC #	Following Upgrade
U2	92-013550-01
U4	92-013550-11
U62	92-013083-01
U63	92-013083-11

Install the IC's from the upgrade kit. Verify that the IC's have been installed with the notched position as shown and that no pins are bent. If the serial number of your unit is less than 91-1114XXXXX, the Following Option cannot be installed. Please contact your distributor or Compumotor Applications for more information.



Both RAM and PROM IC's should be installed
so that the empty slots are positioned as above.

Step ③ Reassemble the Model 4000. Slide enclosure cover over the fan end and install the five screws. Plug in the Control Panel connector (phone cord) and AC cord. Apply power.

Step ④ When starting up your Model 4000 for the first time after installing your new PROMS, a message may appear on the front panel saying Your battery has failed. You will need to press the following keys in order to reset the Model 4000. This will clear the error message.

```
ACCESS
4000
ENTER
ETC  (F6)
RESET  (F2)
ENTER
```

Ratio Following

Ratio following is the most basic type of following. It provides the ability to follow a master continuously or for preset distances, it may take place at one fixed ratio or through many different ratios, and a slave can follow in the same or opposite direction as a master.

Ratio Following Statements

In order to command a following move, there are several set-up parameters which must be specified prior to the move taking place. First, it is useful to define master and slave multipliers so later programming can take place in user units. The **UNIT POS** statement defines the slave's scale factor, and the **UNIT MASTER** statement defines the master's scale factor. Usually, these scale factors are both programmed so that both master and slave units are the same, but this is not required. For example, if the slave is a 25000 step/rev microstepper mounted to a 4 pitch leadscrew, a **UNIT POS** of 100000 allows programming in inches. Similarly, if the master is a 1000 line encoder (which gives 4,000 steps per revolution after quadrature) mounted to a motor on a similar leadscrew, a **UNIT MASTER** of 16000 (4 * 4000) would allow the same programming units.

If the scale factor values are not immediately obvious, it is possible to set up some simple tests in the 4000 to provide those numbers. To find the **UNIT MASTER** parameter, use the **IN POS** or **DISPLAY** statements for the master axis absolute position (either **E_ABS** or **M_ABS**) to read positions before and after the master moves. Be sure **UNIT POS** for that axis is set to 1, so the positions will be read in steps. Now move the master a known distance in the desired units, for example 50 inches. The difference between master position

before and after that move divided by the move distance in the desired units is the **UNIT MASTER** parameter. The **UNIT POS** could similarly be determined by commanding the slave axis to move a known distance (i.e. 1 inch) in a simple motor mode move. The difference between motor position before and after that move is the **UNIT POS** parameter.

The **FOL MASTER** statement defines the masters and the slaves. Any incremental or absolute encoder input or any motor step output on the 4000 can be used as the master. If the master is an encoder input, the axis number of the slave need not match the axis number of the encoder input. For example axis #1 can use the encoder input connections for axis #3. If mechanics dictate that the master pulses are counting in the negative direction, the user can enter a minus (-) sign on the argument of the **FOL MASTER** statement and the 4000 will reverse the counting direction.

☛ Important Terminology

*The **FOL MASTER** statement configures an axis as a slave, and is required before any other **FOL** or **FOLM** statement may be executed. Execution errors will result if this rule is violated. In the remainder of this text, the term Following Mode will be used frequently. A slave is in following mode if **FOL MASTER** and **FOL ENABLE YES** statements have been executed. It means that it's motion is dependent on the master. If a **FOL ENABLE NO** is executed, the slave is not in following mode, and its move will be the normal time based on moves independent of the master. Most **FOL** and **FOLM** statements may be executed regardless of whether the slave is in following mode. However, a **FOL MASTER** must be executed before any **FOL** or **FOLM** statement may be executed.*

To allow subsequent moves to be completed as a slave to the specified master, it is necessary to use the **FOL ENABLE** statement to enable following mode. In order to enable following mode, it is necessary to have previously defined a master.

The **FOL RATIO** statement establishes the maximum allowed ratio for a preset move, or the final ratio for a continuous move. This statement defines the relationship between master and slave velocities and positions after acceleration to the ratio has been completed. The format of the statement is *slave value:master value* with each of the values scaled by their respective unit multipliers. If the scale factors are set up to program in inches, and a user has an **FOL RATIO** of 1:1.5, the slave will travel 1 inch for every 1.5 inches the master travels.

The **FOL MDIST** statement defines the master distance over which a preset slave move will take place, or the master distance over which a continuous slave move will accelerate to commanded ratio. Acceleration for either type of slave move can alternately be defined with the **ACCEL** statement. Whichever statement has most recently been specified prior to a slave move, **ACCEL** or **FOL MDIST**, will be the parameter used to determine the move's acceleration and deceleration ramps. The **FOL MDIST** value is specified in user units and is scaled by the **UNIT MASTER** parameter. Examples and more information on this topic can be found below in the section titled *Slave vs. Master Move Profiles*.

When a slave is following a master continuously, it may be necessary to adjust the slave's position with respect to the master while maintaining an otherwise constant ratio. The **FOL SHIFT** statement allows time-based slave moves to be super-imposed upon continuous following moves. Both continuous and preset distance moves can be executed with the **FOL SHIFT** statement. The most recently defined velocity and acceleration for the slave will determine the shift move profile. Commanded velocity will be added to the current velocity at which the slave is performing the following move.

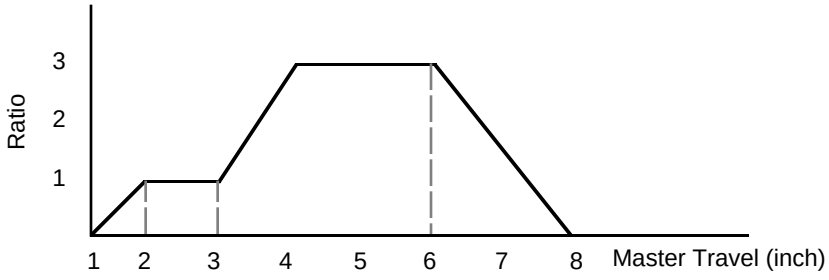
The current slave position (denoted **SLV_P**) and the net slave shift accumulated since being at a constant ratio (denoted **SHIFT**) may be read

into Q variables using the **IN Qn = FOL AXISn** statement, and may also be used for subsequent decision making.

Slave vs. Master Move Profiles

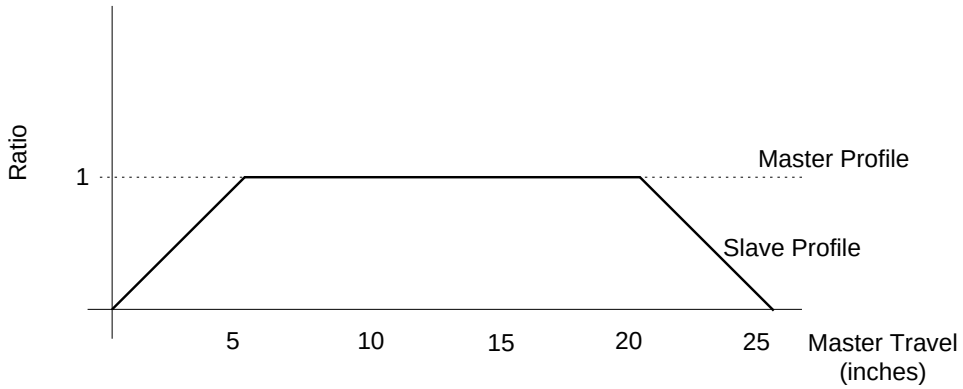
Before an axis begins moving in Following mode, the program should define how the slave will reach the commanded ratio. The acceleration of the slave can be defined in two ways: a time-based acceleration ramp with the **ACCEL** statement, or acceleration over a certain master distance with the **FOL MDIST** statement. If the slave is to accelerate over a specified master distance, a precise position relationship is maintained throughout the acceleration ramp and the constant ratio portion of the move. If a time-based acceleration is specified, the slave accelerates to the commanded ratio, but no position relationship to the master is maintained until the commanded ratio is reached.

For continuous moves, **FOL MDIST** specifies the exact master travel over which the slave ratio changes. This will be required for any application which uses multiple ratios and continuous moves for the construction of precisely defined multi segment moves. In the profile below, the first two moves change ratio over one master inch, and the final ramp to zero takes place over two master inches.

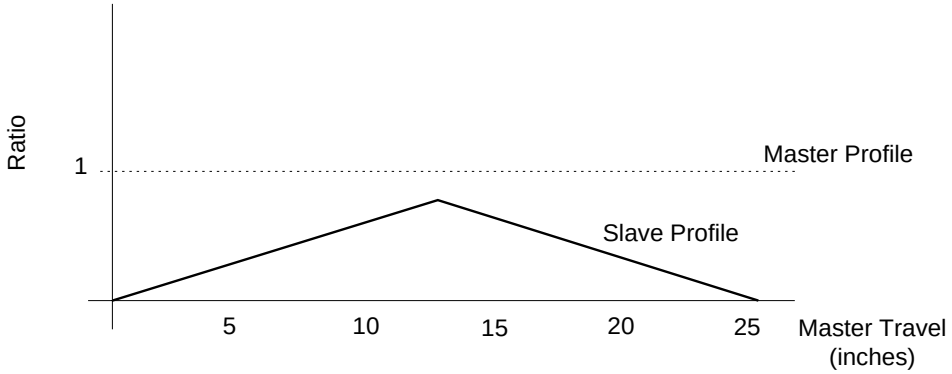


In the example above, **FOL MDIST** was executed *after* an **ACCEL** statement. If the **ACCEL** statement had been the most recent, its slave would use that acceleration to ramp to the new ratio, but the corresponding master travel would be unknown. This may be preferred in velocity based applications such as mixing or pumping, in order to match the acceleration to the motor's torque and load.

For preset moves, the **FOL MDIST** parameter has a different purpose, it is the master distance over which the entire slave move is to take place. If a slave is to move 20 inches over a master distance of 25 inches with a maximum ratio of 1:1, the diagram below illustrates the move profiles:



If the master distance specified is too large for the slave distance and **FOL RATIO** commanded, the slave will never actually reach the commanded **FOL RATIO**, and the move profile will look similar to that below. Here, the **FOL MDIST** is 25 inches and the slave is commanded to move 10 inches:



If the master distance is too small for the slave distance and **FOL RATIO** commanded, the 4000 will not perform the move at all. For example if the **FOL MDIST** is 25, the **FOL RATIO** is 1:1, and the slave is commanded to move 30 inches, the move will not even be attempted.

Summary of Ratio Following Statements

```
UNIT POS                 'Sets the slave distance scale factor
UNIT MASTER              'Sets the master distance scale factor
FOL MASTER               'Defines masters for slave axes
FOL RATIO                'Establishes the maximum ratio for a preset move
                         or the 'final ratio for a continuous move
FOL MDIST                'Defines the master distance over which slave
                         'acceleration or moves are to take place
FOL SHIFT                'Allows adjustment of slave position on the fly
                         during 'continuous following moves
FOL ENABLE               'Enables or disables following mode
IN Qn=FOL AXISn SLV_P    'reads the current slave posiiton
IN Qn=FOL AXISn SHIFT    'reads the net shift since constant ratio
```

The examples below will help to clarify the concept of Ratio Following.

Electronic Gearbox

An electronic gearbox is a classic application for Ratio Following, and very easy to program. Suppose we need a three output gearbox, with all three outputs geared off the same input. Also, each gear ratio must be individually programmed. In this example, a 1000 line encoder is mounted to the shaft of a master giving 4000 master counts per revolution after quadrature. This encoder is fed into the encoder input on axis #1 of the Model 4000. The motors on axes 1, 2, and 3 have resolutions of 200, 1000, and 25000 steps/revolution.

In this example, a precise position relationship is not required between master and slaves, so a standard acceleration will be specified. The slaves will accelerate to a 1 to 1 ratio (in terms of revolutions), and after 10 seconds, the gear ratio on each axis will change to 10 slave revolutions for each master revolution.

☛ The external master encoder is wired to Axis #1

In this example **ENC1** is specified as the master. This means that the *external* master encoder is wired to the Model 4000's axis #1 encoder input. It does not mean that an encoder driven by Axis #1 is the master. That would result in a circular following becoming unstable or running away.

```
UNIT POS 200 1000 25000 *          'Slave scale factors set to motor
                                   'resolutions
UNIT ACCEL 200 1000 25000 *        'Acceleration scale factors to motor
                                   'resolutions
UNIT MASTER 4000 4000 4000 *       'Master scale factor to number of pulses
                                   'per rev
FOL MASTER ENC1 ENC1 ENC1 *        'Encoder #1 assigned as master to all
                                   'three inputs
FOL ENABLE YES YES YES *           'Enable slaves to follow
```

```
ACCEL 5 5 5 *                    'Acceleration rate from current ratio to
                                 'new
FOL RATIO 1:1 1:1 1:1 *          'Initial following ratio is 1 to 1 for
                                 'each axis
MOVE SLEWCW SLEWCW SLEWCW *      'Begin slave continuous following move
WAIT FOR 10 SECONDS              'Wait to change to new ratio
FOL RATIO 10:1 10:1 10:1 *       'Assign new following ratio
MOVE SLEWCW SLEWCW SLEWCW *      'Start moving to new ratio
```

Trackball

Another example of Ratio Following is that of trackball. A trackball is a two axis, two dimensional positioning device. Its operation is similar to that of a two axis joystick, except it controls position rather than velocity. Just as a mouse is used to position the cursor on a computer screen, a trackball could be used to position an X-Y stage.

In this example, a two axis trackball is needed which can do fine and coarse positioning of an X-Y stage. The fine or coarse setting is selected by the user with a two position switch connected to trigger #1 on the 4000. A second trigger on the 4000 is used to transfer the stage back and forth from trackball to standard point-to-point positioning mode. Unlocking the stage from the trackball is necessary because of other point-to-point move requirements elsewhere in the 4000 program.

The trackball housing has two encoders mounted at 90 degrees to each other which are driven by rubber wheels in contact with the ball. The stage is driven by motors and leadscrews, and for one inch of trackball motion to result in one inch of stage motion, the slave to master ratio must be 10 to 1. This will be the ratio for coarse positioning. The fine positioning ratio will be one tenth of that. When trigger #1 is low, coarse positioning is selected, and when trigger #2 goes low, the stage becomes locked to the trackball. Each change of state of the triggers calls a different subroutine in the Model 4000 program, however, the ratios can only change if the stage is locked to the trackball. The trackball is initially unlocked and fine positioning is selected.

```
FOL MASTER ENC1 ENC2 * *         'Master encoders are assigned
FOL RATIO 1 1 * *                'Initial ratio is set to fine positioning
FOL ENABLE NO NO * *             'Following is initially disabled
ON TRIG1 = 0 GOSUB COARSE        'Interrupt to allow change to coarse
                                 'positioning
ON TRIG2 = 0 GOSUB LOCK          'Interrupt to allow trackball to be
                                 'engaged
LABEL MAINLOOP                   'Start of users main program loop
.                                'Other program operation takes place
.                                'Within this loop
.
GOTO MAINLOOP                    'Repeat main program loop

LABEL COARSE                     'Subroutine to assign coarse positioning
IF TRIG2 = 1 GOTO C_EXIT         'Allow ratio change only if stage is
                                 'locked to trackball
FOL RATIO 10 10 * *              'Coarse positioning ratio of 10 to 1
MOVE SLEWCW SLEWCW * *           'Move to begin travel at new ratio
LABEL C_EXIT                     'Label to exit subroutine if not locked
ON TRIG1 = 1 GOSUB FINE          'When switch #1 changes state, change to
                                 'fine
RETURN                           'Return to main program loop

LABEL FINE                       'Subroutine to assign fine positioning
IF TRIG2 = 1 GOTO F_EXIT         'Allow ratio change only if stage is
                                 'locked to trackball
FOL RATIO 1 1 * *                'Fine positioning ratio is 1 to 1
MOVE SLEWCW SLEWCW * *           'Move to begin travel at new ratio
LABEL F_EXIT                     'Label to exit subroutine if not locked
ON TRIG1 = 0 GOSUB COARSE        'When switch #1 changes state, change to
                                 'coarse
RETURN                           'Return to main programming loop
```

```
LABEL LOCK                      'Subroutine to lock stage to trackball and
                                'enter following mode
FOL ENABLE YES YES * *          'Enable following on axes #1 and 2
MOVE SLEWCW SLEWCW * *          'Start move at current ratio
ON TRIG2 = 1 GOSUB UNLOCK       'If switch #2 changes state, unlock the
                                'stage
RETURN                          'Return to main programming loop

LABEL UNLOCK                    'Subroutine to unlock the stage and enter
                                'standard positioning mode
MOVE STOP STOP * *              'Stop current move in progress
FOL ENABLE NO NO * *            'Disable following on both axes
ON TRIG2 = 0 GOSUB LOCK         'If switch #2 changes state, lock into
                                'following
RETURN                          'Return to main programming loop
```

The Master Cycle Concept

The previous examples illustrated the basics of Ratio Following but did not address applications which require precise programming synchronization between moves and I/O control based on master positions or external conditions. The concept of the master cycle will greatly simplify the required synchronization.

A master cycle is simply an amount of master travel over which one or more related slave events take place. The distance traveled by the master in a master cycle is called the master cycle *length*. A master cycle *position* is the master position relative to the start of the current master cycle. The value of master cycle position increases as positive master cycle counts are received, until it reaches the value specified for master cycle length. At that point, the master cycle position becomes zero, and the master cycle *number* is increased by one. This condition is called *rollover*.

The master cycle concept is analogous to minutes and hours on a clock. If the master cycle is considered an hour, then the master cycle length is 60 minutes, the number of minutes past the hour is the master cycle position, and current hour is the master cycle number. The master cycle position goes from 59 to zero as the hour increases by one.

By specifying a master cycle length, periodic actions may be programmed in a loop or with subroutines which refer to cycle positions, even though the master may be running continuously. To accommodate applications where the feed of the product is random, the start of the master cycle may be defined with trigger inputs. Two types of *waits* are also programmable to allow suspension of program operation or slave moves based on master positions or external conditions.

Master Cycle Statements

The **FOL MAS_CYC** statement is used to define the length of the master cycle in user units. This statement is scaled by the **UNIT MASTER** parameter to get the master cycle length in steps. For periodic master cycle operation, this parameter must be defined before those statements which wait for certain master positions are executed. The default value of **FOL MAS_CYC** is zero, which means the master cycle length is practically infinite. (It is an extremely large number i.e., 4,294,967,246 steps.) If a value of zero is chosen the master cycle position will keep increasing until this value is exceeded or a new cycle is defined with the **FOL NEWCYC** statement described below. If a non-zero value for **FOL MAS_CYC** is chosen, the internally maintained master cycle position will keep increasing until it reaches the value of **FOL MAS_CYC**. At this point, it immediately rolls over to zero and continues to count.

Once the length of the master cycle has been specified, the **FOL NEWCYC** statement is used to define the start of a master cycle. A new cycle can be started either immediately when this statement is executed, or when a specified trigger input becomes true. If a trigger is to define a new master

cycle, the 4000 program will not wait for the trigger to occur before continuing on with normal program execution. In this case, master cycle definition is pending the trigger input, and some statements which use master cycle position will not operate correctly until the trigger occurs. To halt program operation, one of the wait statements mentioned below can be used. A new master cycle will also start automatically when the full master cycle length is reached. This will be useful in continuous feed applications.

By using the **FOL CYC_OFF** statement it is possible to assign for the first cycle only, an initial Master Cycle Position to be a value other than zero. When a master cycle is defined with the **FOL NEWCYC** statement or the trigger specified in the **FOL NEWCYC** statement, the master cycle position takes the initial value previously specified with the **FOL CYC_OFF** statement. The value for **FOL CYC_OFF** is given in user units and scaled by **UNIT MASTER**. **FOL CYC_OFF** was designed to accommodate situations in which the trigger that defines the new cycle occurs either before the desired cycle start, or somewhere in the middle of what is to be the first cycle. In the former case, the **FOL CYC_OFF** value would be negative. The master cycle position would be initialized with that value, and would increase right through zero until it reached master cycle length. At that point, it would roll over to zero as usual. The continuous cut to length example illustrates the use of a negative **FOL CYC_OFF**. If it is desired that the first cycle is defined as already partially complete, the **FOL CYC_OFF** value would be greater than zero, but less than the master cycle length.

To give a value for **FOL CYC_OFF** which is greater than master cycle length is meaningless since master cycle positions are always less than the master cycle length. The 4000 responds to this case as soon as a new cycle begins by using zero instead of the initial value specified with **FOL CYC_OFF**.

The concept of an initial master cycle position is useful when the first cycle of a periodic operation must begin at some place other than the beginning of the cycle. A typical example is when a trigger that senses the motion of the master is physically offset from master position at which some action must take place. This is illustrated in the *Cut to Length* example.

The master cycle length may be changed with the **FOL MAS_CYC** statement, even after a master cycle has been started. The new master cycle length takes affect as soon as it is issued. If the new master cycle length is greater than the current master cycle position, the cycle position will not change, but will rollover when the new master cycle length is reached. If the new master cycle length is less than the current master cycle position, the new master cycle position becomes equal to the old cycle position minus one or more multiples of the new cycle length.

The current master cycle position, denoted **MAS_P**, and the current master cycle number, denoted **MAS_C**, may both be read into Q variables at any time using the **IN Qn = FOL AXISn** statement. Very often, the master cycle number will be directly related to the quantity of product produced in a manufacturing run, and the master cycle position can be used to determine what portion of a current cycle is complete.

Following Wait Statements

Two types of wait statements may be used with following in the Model 4000, they are intended for distinctly different uses. The first type of wait that can be specified, **FOL WAIT**, will cause the 4000 program to halt operation until the condition specified is satisfied. The **FOL WAIT** statement allows program operation to halt based on a specified trigger becoming active, or until a particular master cycle position has been reached. If a master position is specified, it is to be entered in user units, as the value is scaled by the **UNIT MASTER** parameter. This function will be useful in delaying subsequent I/O activation until the master has achieved the required position or an object has been sensed.

The second type of wait, **FOL MOVEWT**, will not cause immediate program suspension, but rather the next slave move specified will not begin until the condition specified is satisfied. The **FOL MOVEWT** statement is useful in synchronizing subsequent moves with a master cycle position, or a specified trigger input. As with the **FOL WAIT** statement, a master cycle position should be entered in user units. The **FOL MOVEWT** statement is preferred over the **FOL WAIT** statement if precise synchronization of master and slave moves is required. If the slave's move is to take place over a certain number of master steps (see **FOL MDIST**), that master step count starts with the master position specified by this statement, or the master position is latched when the trigger occurs. This eliminates variations in move profiles due to processing delays.

Both **FOL WAIT** and **FOL MOVEWT** may wait on a master cycle position. It is possible for the wait caused by either of these statements to occur while a master cycle is pending definition on a trigger. In that case, the corresponding wait will include waiting for the trigger and the cycle position in the master cycle defined by that trigger. If the pending status is cleared by either a **FOL NEWCYC NO** or a **FOL NEWCYC IMMED**, the wait is cleared also. It is also possible for specified master cycle position to have been already exceeded by the time the wait takes place. Please refer to the **FOL WAIT** and **FOL MOVEWT** descriptions for the respective responses to this situation.

Summary of Master Cycle and Wait Statements

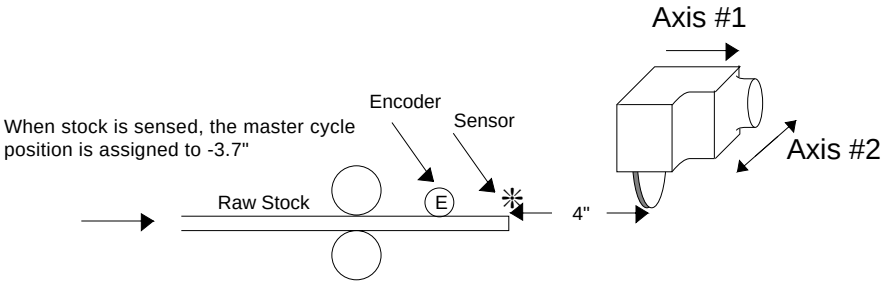
```
FOL MAS_CYC     'Defines the length of the master cycle
FOL NEWCYC      'Allows a new master cycle to begin immediately or upon a
                'trigger input
FOL CYC_OFF     'Defines the initial position of a new master cycle
FOL WAIT        'Suspends program execution until a specified master
                'position or trigger input is activated
FOL MOVEWT      'Suspends execution of the next slave move until a
                'specified master position or trigger input is activated
```

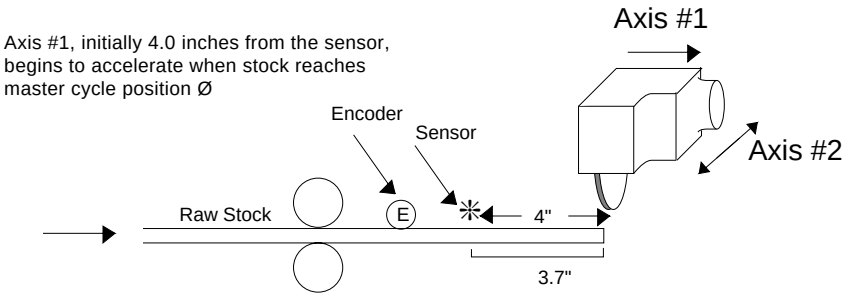
The next example illustrates the use of the master cycle concept and the commands above.

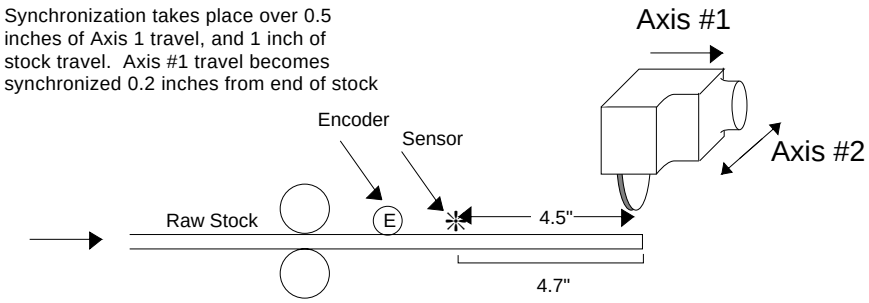
Continuous Cut to Length

This application calls for automobile trim to be cut to a pre-defined length. The saw is controlled by axes #1 and #2 on the 4000. It must be moving with the material while the cut is being made (axis #1), and also move perpendicular to the trim (axis #2) to actually make the cut. The trim comes in long stock which moves continuously under the cutting area.

The leading edge of the trim stock is detected with a sensor connected to trigger #1 which is located 4 inches from the home position of the saw. Axis #1 will be following the trim based on an encoder mounted on the trim via a friction wheel. The encoder is a 1000 line encoder and the wheel is geared to give 2 revolutions per inch of trim. This results in 8,000 post quadrature steps per inch of trim. Axis #1 has a resolution of 25000 steps per rev, and is connected to a 2 pitch leadscrew 24" in length. Axis #2 is similar in mechanics but its length is 10". The travel on Axis #1 will be controlled by the speed at which axis #2 makes its cut. Limit switches are in place for safety.







Below, the master cycle length will be set equal to the desired cut length (36" in the example below), which the operator can change by modifying variable Q1. The cut cycle will be a continuous loop, but the first cut will be made 0.2" from the end of the stock to ensure an even first edge. Axis #1 will accelerate to the desired tracking ratio over 1 inch of master travel for all cuts. Assume that the home position of both axes is at position 0".

The Cut to Length example takes advantage of being able to change master cycle length, while being careful to change it only at the beginning of a current cycle this ensures that the current master cycle position will be less than the new master cycle length, and will not change as a result of a change in cycle length. In this example the master cycle length and corresponding waits are redefined every cycle to the current value of Q1. The value of Q1 becomes the cut length, and can be changed via remote command during program execution. With minor modifications, cut lengths, and number of iterations could be read from **DATA** statements.

```
UNIT POS 50000 50000 * *        'Set axes #1 and #2 scale factors for
                                'programming in inches
UNIT VEL * 50000 * *            'Axis #2 velocity scale factor for
                                'inches/sec
UNIT ACCEL * 50000 * *          'Axis #2 accel scale factor for
                                'inches/sec/sec
```

```
UNIT MASTER 8000 * * *          'Master scale factor for programming in
                                'inches
ACCEL * 20 * *                  'Accel for axis #2 (axis #1 accel set with
                                'FOL MDIST)
VEL * 5 * *                     'Velocity on axis #2 to 5 inches/sec
MODE M_ABS M_ABS * *            'Absolute positioning mode for non-
                                'following moves
MATH Q1 = 36                    'Desired cut length is 36"
MATH Q2 = 4                     'Sensor is 4" from home position of axis #1
MATH Q2 = Q2 + 0.2              '1st cut to be 0.2" from end of stock
MATH Q3 = 1                     'FOL MDIST to be set to 1"
MATH Q4 = Q3 / 2                'Slave will travel 1/2" when accelerating
                                'to 1:1 ratio while master travels 1"
MATH Q2 = Q2 - Q4               'Take distance slave travels during accel
                                'into account so we'll be up to speed at
                                'position = 0.2" from end of stock.  Then
                                'the cut will be made.
MATH Q2 = -Q2                   'Initial master cycle position will be
                                'negative of distance traveled during
                                'slave wait and accel
FOL MASTER ENC1 * * *           'Encoder input #1 is the master for axis #1
FOL CYC_OFF Q2 * * *            'Set initial master cycle position to wait
                                'length
FOL MDIST Q3 * * *              'Acceleration to constant ratio will take
                                'place over Q3" (1" here) of master travel
FOL RATIO 1:1 * * *             'Following ratio is 1 to 1
FOL ENABLE YES * * *            'Enable following mode on axis #1
FOL NEWCYC TRIG1 * * *          'Define a new master cycle on trigger input
                                '#1
FOL WAIT TRIG1 * * *            'Suspend program execution until stock is
                                'sensed
OUT BIT7 = 1                    'Turn on output for saw blade to move down
                                'into position
FOL MOVEWT Ø * * *              'Wait on first move for necessary master
                                'travel.  This will ensure being at 1:1
                                'ratio at exactly 0.2" from end of stock.
LABEL NEW_CUT                   'Subroutine label for continuous cutting
MOVE SLEWCW * * *               'Start move on axis #1
MATH Q5 = Q1                    'Set Q5 = Q1 (Snapshot of Q1)
FOL MAS_CYC Q5 * * *            'New master cycle length is cut length
MOVE * 10 * *                   'Once axis #1 is up to speed, move axis #2
                                'across stock to make the cut
MOVE STOP * * *                 'Stop following move on axis #1
FOL ENABLE NO * * *             'Exit following mode
OUT BIT7 = 0                    'Raise the saw blade
MOVE 0 0 * *                    'Move both axes back to home positions
OUT BIT7 = 1                    'Move saw blade into position for next cut
FOL ENABLE YES * * *            'Enable following on axis #1
FOL MOVEWT Q5 * * *             'Wait for next move to start until next
                                'master cycle
GOTO NEW_CUT                    'Repeat the cut cycle
```

Following Performance and Measurement

Following performance includes such topics as following error, following smoothness, response to changes in master speed, and safety limitations. The overall following performance is determined by how the motion of the master is measured and constraints on the slave. These are parameters which may be specified by the user, and the resulting performance may be monitored.

Following Performance Statements

The **FOL SMOOTH** and **FOL VELFF** statements allow users to modify the default Following algorithm as necessary to optimize performance in their applications. Values of 1, 2, 3, or 4 for the **FOL SMOOTH** parameter correspond

to velocity averaging periods of 4, 8, 16, or 32 milliseconds. Applications where the master speed is slow or if the master changes velocity rapidly may see smoother slave motion if the **FOL SMOOTH** parameter is larger than the default of 1. **FOL VELFF** allows the user to enable or disable velocity feed forward. Velocity feed forward eliminates the dependence of following error on master velocity, but may result in rough motion. Master velocity is measured for this feature, and may be read using the **IN Qn=FOL AXISn MAS_V** statement. The **DEFINE TRIGDB FOR DIRSET**, and **FOL LEAD** statements allow the user to account for the specific requirements or performance of the motor drive and load. **DEFINE TRIGDB** allows the user to individually specify the debounce times for the trigger inputs. **FOL DIRSET** lets the user specify if a change in the direction output signal requires a set-up time. **FOL LEAD** lets the Model 4000 compensate a velocity dependent lag in the drive or load. The *Technical Considerations* section of this manual contains more details on these topics and others of the Following algorithm.

The **FOL MAXVEL** and **FOL MAXACC** statements define the maximum velocity and acceleration at which the slave will be allowed to move. Preset following moves and **FOL SHIFT** moves may command velocities and accelerations that the slave axis is physically not able to complete. The **FOL MAXVEL** and **FOL MAXACC** statements allow the user to set these limits. If the slave is commanded to move at rates beyond the defined maximums, the slave will begin falling behind it's commanded position. If this happens, the position error is made up as soon as the commanded velocity and acceleration fall within the limitation of **FOL MAXVEL** and **FOL MAXACC**.

These statements should be used only to protect against worst case conditions, and should be avoided altogether if they are not needed. If an axis is not able to follow its profile because of limitations imposed by these statements, some correction motion will occur when a **MOVE** statement is completed. This is due to the following error and the resulting *Dynamic Position Maintenance*. Refer to *Technical Considerations for Following* in this chapter. If the maximum acceleration is set very low, some oscillation about the setpoint may occur. This is because the slave is not allowed to decelerate fast enough to prevent overshoot.

Monitoring Following Error

As soon as an axis becomes configured as a slave, the slave setpoint position is continuously updated and maintained. The setpoint position is calculated from the master position and velocity and the current ratio or velocity of the slave. This continuously updated setpoint is used as the target position for the dynamic position maintenance, which is described in detail in the *Technical Considerations for Following* section of this chapter. Whenever the setpoint position is not equal to the actual slave position, a following error exists. This following error, if any, may be positive or negative, depending on both the reason for the error and the direction of slave travel. The following error is defined as the difference between the setpoint position and the actual position.

```
Following Error = Setpoint position - Actual position
```

If the slave is traveling in the positive direction and the actual position lags the setpoint position, the error will be positive. If the slave is traveling in the negative direction and the actual position lags the setpoint position, the error will be negative. This error is always monitored, and may be read into a variable at any time using the **IN Qn = FOL AXISn FOL_ERR** statement. The error value in slave steps is inversely scaled by **UNIT POS** for the axis, so the resulting value in the variable is the error expressed in the user's units. This value may be used for subsequent decision making, or simply recording the error corresponding to some other event.

Although it is useful to be able to read the current error, it is not a fast and convenient method of continuously monitoring error. The 4000 does this for the user, and the user may specify a following error tolerance using the **FOL**

PTOL statement. The tolerance value is given in slave position units and scaled by **UNIT POS**. This statement allows the user to specify the magnitude, or absolute value, of acceptable slave following error. Although the sign of the following error is reported when it is read into a variable, it is ignored during the continuous comparison to the value specified with **FOL PTOL**. If the magnitude of the actual following error ever exceeds the specified tolerance, the 4000 latches the condition of *following error tolerance exceeded*. The **ON FOL_ERR** statement allows the user's program to detect this condition, and allows the user complete flexibility in the response. If the following error tolerance is exceeded and the **ON FOL_ERR** statement has been executed, program execution will branch (**GOTO** or **GOSUB**) to the destination specified in that statement. This condition is cleared only by re-executing the **FOL PTOL** statement, executing a **FOL MASTER NO** statement, or when the program finishes. If the user's program requires that the 4000 respond to a new occurrence of excess following error, the **FOL PTOL** statement should first be executed to clear the old error, and then the **ON FOL_ERR** statement executed to allow detection of the condition.

Error Detection Windows

The discussion so far has considered only the *continuous* detection of excessive following error. There are many applications, however, in which a periodic repetitive operation takes place, and precise synchronization is only important during a portion of the cycle. For example, consider a printing application in which a continuous sheet of paper moves under a rotating print drum. Only a portion of the circumference of the drum has the print pattern, which is raised to make contact with the paper. The application may require a very tight tolerance during the portion of print drum rotation in which the drum contacts the moving paper. For the remainder of the drum rotation, there may be accelerations on either the drum or the paper. During this time, holding a tight following error tolerance is not required, and may even be impossible due to large loads and mechanical constraints. For such an application, it should be possible to define a portion, or window of a cycle in which excess following error is detected, and ignored in the remainder of the cycle. The 4000 provides for this by allowing for definition of a *error detection window within a master cycle*. The master cycle concept has been discussed in great detail earlier in this chapter, and is important to the understanding of an error detection window.

The 4000 allows the user to define the starting position of this window within the master cycle using the **FOL WIN_P** statement. This window starting position is given as a master cycle position, scaled by **UNIT MASTER**. Because it is a position within a master cycle, its value must be less the master cycle length to be valid, i.e., meaningful. The width of the window may be defined using the **FOL WIN_W** statement. This window width is given as a master distance, scaled by **UNIT MASTER**. Because it is a distance within a master cycle, its value must also be less the master cycle length to be valid, i.e., meaningful. If either of these values are greater than the master cycle length, the error detection window will not be valid, and *error detection will occur continuously*. Even if these values are larger than the master cycle length, however, they are saved as given. If the master cycle length is subsequently made larger than the window values, (using the **FOL MAS_CYC** statement), the window definition becomes valid. This allows the three statements, **FOL MAS_CYC**, **FOL WIN_P**, and **FOL WIN_W** to be given in any order.

As a special case, it is possible to give a window width (**FOL WIN_W**) of zero. This is also its value on power-up and after a **FOL MASTER** statement. The result of this is that the error detection window will not be valid, and *error detection will occur continuously*. It is also possible and valid to specify a window starting position (**FOL WIN_P**) close enough to the end of a cycle

that the end of the window (start position plus width) exceeds the end of the master cycle. In this case, the remainder of the window simply applies to the start of the next cycle. It is perfectly valid for an error detection window to span the end of one cycle and the start of the next.

Motor and Encoder Comparison

The 4000 has a feature (**FOL ENCCHK**) which allows a comparison between the commanded position in motor steps, i.e. controlled in motor step mode, and the actual position in encoder steps. This is especially useful when the slave is a Compumotor servo drive such as the Z Drive or the Dynaserv. Its first major purpose is to assist in the tuning of the drive for minimum following error. It also facilitates rapid electronic response to excess following error.

The previous discussion about following error holds true whether the slave is in motor step mode or encoder step mode. When the slave is encoder step mode, the measured following error is the difference between the commanded and actual encoder step positions, and this error is corrected with dynamic position maintenance (refer to *Technical Considerations for Following*). But Compumotor servo drives have their own position control feedback loop, and it is usually not desirable to have the 4000 doing position maintenance. For this reason, the 4000 should be in motor step mode with these servo drives.

These servo drives accept step and direction input and provide pseudo-quadrature encoder output to indicate the actual position. They can be configured to make the input step resolution and encoder feedback resolution the same. The **FOL ENCCHK** feature assumes the input step resolution and encoder feedback resolution are the same when comparing the commanded position in motor steps to the actual position in encoder steps. This allows the 4000 to continuously monitor the following error of the servo drive without interfering with the drive's own position control. The error detection window, error tolerance, and **ON FOL_ERR** features may thus be applied to the servo drive's following error. In addition, when this feature is enabled, the POB output for that slave axis becomes an indicator for following error out of tolerance. When error detection is occurring (either in a window or continuously) and the following error is greater than the specified tolerance, the output will be ON (low). Otherwise it will be OFF (high). Using this output in combination with an oscilloscope could assist in tuning a drive for minimum following error.

Summary of Following Performance and Measurement Statements

```
FOL MAXVEL                'Sets the maximum velocity at which a slave may
                          'travel
FOL MAXACC                'Sets the maximum acceleration a slave may use
                          'to change ratio
FOL SMOOTH                'Sets the sample time over which master velocity
                          'is calculated
FOL VELFF                 'Allows velocity feed forward to be enabled or
                          'disabled
FOL WIN_P                 'Defines master window position
FOL WIN_W                 'Defines master window width
FOL PTOL                  'Defines following error tolerance
IN Qn=FOL AXISn FOL_ERR   'Reads the current following error
ON FOL_ERR                'Interrupts program when following error
                          'tolerance is exceeded.
IN Qn=FOL AXISn MAS_V     'read current master velocity
DEFINE TRIGDB             'sets debounce time for trigger inputs
FOL DIRSET                'enables or disables direction change setup time
FOL ENCCHK                'enable or disable encoder/motor step check
FOL LEAD                  'advance setpoint proportional to slave speed
```

Periodic Master/Slave Synchronization

For many applications discussed so far, simply maintaining constant ratio or performing a sequence of moves with respect to a master cycle positions

and triggers will solve the application needs. If these operations are repeatably periodic in nature, master cycle positions may be used to achieve the required synchronization between master and slave. There are other *applications in which periodic operations must occur in intervals which are not perfectly repeatable. For these, the master and slave must be re-synchronized every cycle.* These applications will be able to make use of the 4000's master/slave synchronization features.

Two examples may illustrate this concept. In many packaging operations, a product may enter on a conveyor with non-repeatable, or random timing, yet must leave on a conveyor with perfect spacing and position. Such applications will be referred to here as *Random Timing Infeed.* In these applications, a major spacing correction may occur every cycle. In other applications, the operation may be nearly perfectly repeatable, but vary slightly over the course of many cycles. For example, a web may have periodic registration marks which must be aligned with a periodic operation such as printing or cutting. Small variations in the location of the registration mark may occur, particularly if the web can stretch or slip. These types of applications will be referred to here as *Web Processing.* In both applications the actual slave position may differ from the expected slave position when a synchronization input occurs.

Master and Slave Marks, Synchronization Offset, Sync Error

The 4000 allows the user to define two external events, or *marks*, which capture the slave position. These are called *Master Sync Mark* and *Slave Sync Mark*, and are defined with the **FOL M_SYNC** and **FOL S_SYNC** statements respectively. The nomenclature helps to distinguish one mark from the other, and also helps when it is assumed that the Master Sync Mark occurs as a result of motion from the master, and the Slave Sync Mark occurs as a result of motion from the slave. Each time either mark occurs, the corresponding slave position is captured and saved internally. These positions may be read into Q variables using the **IN Qn = FOL AXISn M_SYNC** and **IN Qn = FOL AXISn S_SYNC** statements. The user may also pre-define an expected difference between these captured slave positions. This expected difference is called the *Slave Synchronization Offset* and is defined using the **FOL SYNC_OFF** statement. There is an important reason for defining the offset expected between two positions instead of defining the position expected at a single synchronization mark. It allows continuous motion in one direction without requiring a continuous re-calculation of the expected slave position. The difference between the actual offset and the expected offset is called the Sync Error. This error may be read into a Q variable using the **IN Qn = FOL AXISn SYNC_ERR** statement. To understand exactly how to use this, more precise definitions of actual synchronization offset and sync error are required.

Synchronization Offset and Synchronization Error Definitions

Although these features may be used in a variety of combinations, the synchronization marks and offset are named with the idea that the master sync mark records a slave position reference, and the slave sync mark records a slave position measurement. The difference between the measurement and reference is the actual offset.

```
Actual sync offset = slave position at slave mark - slave position at
master mark.
```

The sign of the actual sync offset will depend on the order in which the marks are encountered, and the direction of slave travel. If the slave is traveling in the positive direction, and the master sync mark occurs first, the actual sync offset will be positive. Of course, the slave may travel in the negative direction, and the master sync mark could occur either before, after, or at the same time as the slave sync mark. It is important to understand how the actual sync offset is calculated, so that the expected sync offset may be programmed correctly. The difference between the actual offset and the expected offset is called the *Sync Error.*

```
Sync Error = Actual sync offset - Expected sync offset
```

Again, this is defined with the idea that the master sync mark records a slave position reference, and the slave sync mark records a slave position measurement. Under this assumption, motion of the slave either represents or actually drives an object toward the sensor defined as the slave sync mark. After both master and slave sync marks have been received for a given cycle, the error may be read. The Q variable used with the **IN Qn = FOL AXISn SYNC_ERR** statement may be used without modification in a subsequent **FOL SHIFT** statement to bring the error to zero. Alternatively, comparisons or modifications may be made to the variable, depending on the needs of the application.

Master and Slave Sync Mark Definitions

So far, the master and slave sync marks have simply been referred to as external events which capture the slave position. *These external events may be any of the four trigger inputs TRIG1 through TRIG4, or a master cycle position.* In many cases, one sync mark will be a TRIG input, while the other is a master cycle position. If the sync mark is defined as a trigger, the slave position is captured on each occurrence of that trigger. Typically, a sensor which detects a passing object or registration mark would be connected to the trigger input. The characteristics of trigger inputs are described in the **Model 4000 User Guide**.

If the sync mark is defined a master cycle position, the slave position is captured *on only the first occurrence of that cycle position* each cycle. This happens each time a new cycle is defined, or a master cycle has completed and rolled over to start a new cycle. In most cases, the master will be moving continuously in positive direction, so a given cycle position will occur only once per cycle anyway. If rollovers do not occur because the cycle length is nearly infinite (i.e., defined as zero), or a cycle definition is pending a trigger input, the slave position will not be captured. Also, if the master position specified as a sync mark is greater than the master cycle length, it will never occur. For a complete understanding of a sync mark defined as a master cycle position, it is important to understand master cycles. Please refer to the section titled *The Master Cycle Concept* earlier in this chapter.

Using Periodic Synchronization Features

The basic features of periodic synchronization and their use have been discussed in a general manner. *The program must define master and slave sync marks and the expected synchronization offset.* The sync error may be requested and used in subsequent decision making. In order to ensure maximum flexibility in their use, very few restrictions and error checks have been placed in these statements. The only requirement is that the sync marks must have been *defined* before a request for the captured position or the sync error is given with the **IN Qn = FOL AXISn** statements.

By contrast, the requests do *not* check to see that the sync marks have actually occurred, assume any order of occurrence, or attempt to determine if the reported error is a realistic value. These things must be done by the user through thoughtful application design and programming. For example, to ensure that both marks have occurred in a given cycle, the **FOL WAIT** statement may be used to wait for either a trigger or a master cycle position. This solution also requires the application to be designed so the marks always occur in the same order. If this is not possible, the request could wait for some other event which is known to take place after both marks occur. Another solution is to poll for cycle position using the **IN Qn = FOL AXISn MAS_P** and using **IN** or **ON** statements to detect the transition of trigger inputs. In any case, it will be up to the program to determine whether and when to correct the error, and whether to correct all or some portion of the error.

The expected synchronization offset may be changed at any time using the **FOL SYNC_OFF** statement, even while moves are in progress. The sync

error request (**IN Qn = FOL AXISn SYNC_ERR**) uses the expected offset value in currently in effect to calculate the sync error. This means that the desired synchronization offset must be established before the sync error is requested, but not necessarily before the slave positions are captured with the sync marks. The ability to change expected offset independently of other synchronization parameters makes it possible to write a program which allows independent manual correction of automatic alignment routines. The **FOL SHIFT** statement with a correction distance would be used for the automatic correction routine, and the **FOL SHIFT CW** or **CCW** statement would probably be used for the manual correction routine. The manual routine initiated by the operator could read the slave net shift value before and after manual correction and modify the expected offset with the difference. This would ensure that the automatic correction routine did not undo the operator's manual intervention in the next cycle. This is illustrated in the *Random Timing Infeed* example below.

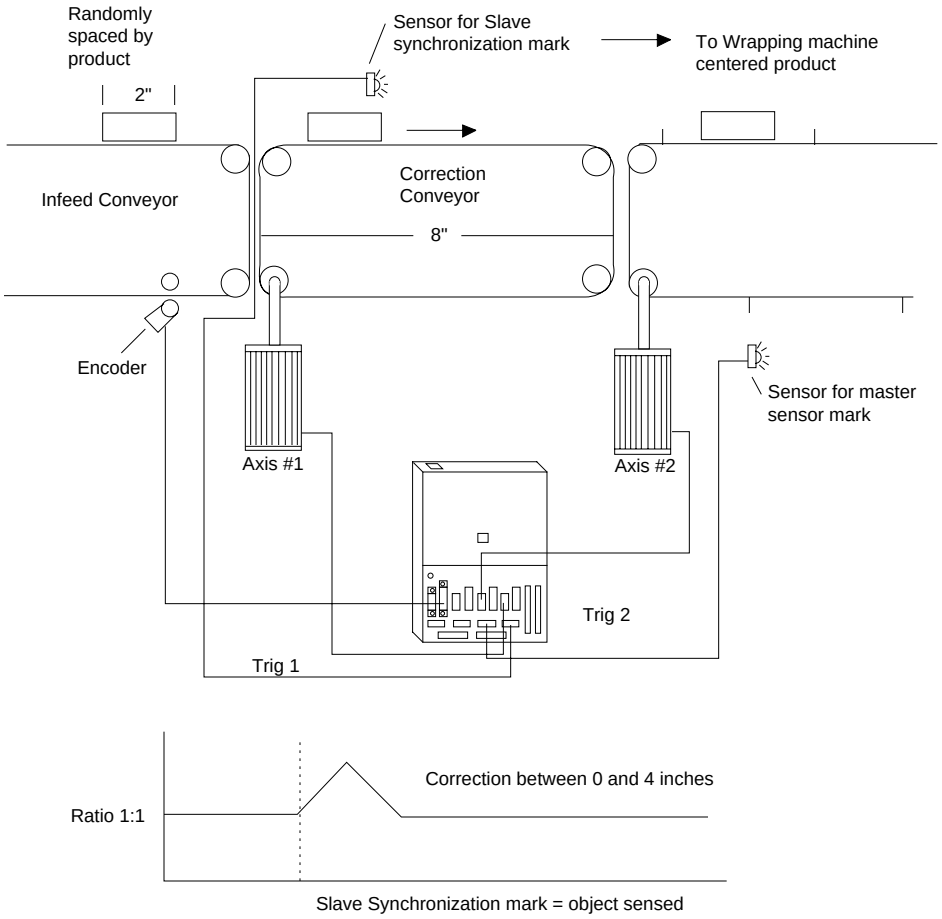
Summary of Periodic Master/Slave Synchronization Statements

```
FOL M_SYNC                      'Define Master Synchronization Mark
FOL S_SYNC                      'Define Slave Synchronization Mark
FOL SYNC_OFF                    'Define expected Synchronization Position
                                'Difference
IN Qn = FOL AXISn SYNC_ERR      'Read Following Synchronization Error
IN Qn = FOL AXISn M_SYNC        'Read Slave position captured by master
                                'synchronization mark
IN Qn = FOL AXISn S_SYNC        'Read Slave position captured by slave
                                'synchronization mark
```

Random Timing Infeed

Random timing infeed refers to operations in which a product may enter on a conveyor with non-repeatable, or random timing, yet must leave on a conveyor with perfect spacing and position. Typically, there may be an infeed conveyor on which products are randomly spaced, a short conveyor on which the correction move is made, and an exit conveyor with dividers called *flights*. As the products move onto the exit conveyer, they must be correctly positioned between flights. All three conveyers must have the same line speed while the product moves from one to the other.

In this example, a previous operation has placed product on the infeed conveyer for subsequent wrapping. The product dimension in the direction of travel is 2 inches. The wrapping machine on the exit conveyer is mechanically triggered by the raised flights, and expects the products to be centered between the flights. The flights are 4 inches apart, so the maximum correction move required will be 4 inches. The length of the correction conveyer must accommodate this move length (4 inches), plus the product dimension (2 inches), plus the travel due to following which occurs during the correction move. The conveyer will move slowly while following, so an 8 inch overall length will be adequate.



Although the product arrives randomly, it will never be closer than 9 inches apart, ensuring that each product can enter and exit the correction conveyer before the next arrives, even if no correction move occurs. As a result of longer spacing, however, some flights will be empty, a condition detected by the wrapping machine. When the product has moved completely onto the correction conveyer, it is detected by a sensor connected to TRIG1, which will be used as the slave sync mark. The sensor is mounted 2.5 inches from the beginning of the conveyer, ensuring that the entire product is on the conveyer before any correction move occurs. The flights on the exit conveyer are detected by a sensor connected to TRIG2, which will be used as the master sync mark. The master sync mark sensor is mounted so that the expected synchronization offset will be zero inches. The fact that product will never be closer than 9 inches apart also ensures that at least two master marks will occur for every slave mark, so the program may simply wait for the slave mark before determining the appropriate correction move.

The infeed conveyer is controlled by another machine, and its motion is measured by an encoder which is geared to give 5000 steps per inch. This encoder is connected to the axis 4 encoder input and will be the master for the other axes. The correction and exit conveyers are both controlled by motors geared to give 50000 steps per inch, and are connected to axes 1 and 2 respectively. The exit conveyer simply tracks the infeed motion at a constant 1:1 ratio. The correction conveyer will also starting moving at constant 1:1 ratio, but may perform correction **`FOL SHIFT`**s which are superimposed on the ratio. The correction axis is the only axis which has synchronization parameters specified, because its position is used as a measurement of the product position. Notice that in this example, no master cycle is defined, because trigger inputs are used for both sync marks.

The mechanically triggered wrapping machine is manually adjusted, assuming products will centered between the flights. Over time, it tends to drift out of adjustment, making the ideal product position something other than centered between flights. Because of this, the operator must be able to

manually and visually adjust the target of the correction moves so that product arrives on the exit conveyer in the desired position with respect to the flights. In the example below, the operator may shift the product after the correction move. A button connected to IN24 BIT1 commands a shift in the CW direction, and a button connected to IN24 BIT2 commands a shift in the CCW direction. The amount of shift is measured and the expected synchronization offset is adjusted accordingly. As a result, the next correction move will place the product in an adjusted position with respect to the exit conveyer flights.

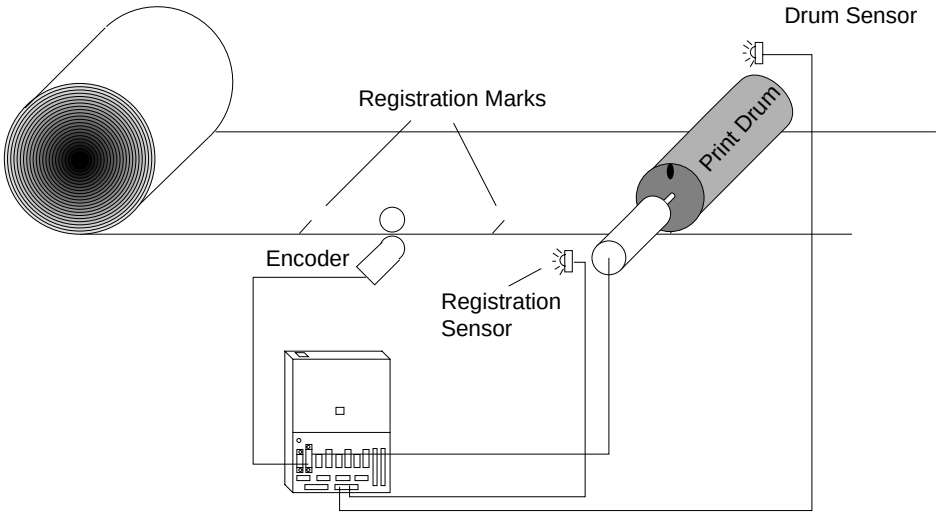
```
UNIT POS 50000 50000 * *       '50000 steps per inch, both conveyers
UNIT MASTER 5000 5000 * *      '5000 encoder steps per inch
FOL MASTER ENC4 ENC4 * *       'axes 1,2 use encoder 4 as master input
FOL RATIO 1:1 1:1 * *          'both conveyers track master infeed at 1:1
                               'ratio
FOL ENABLE YES YES * *         'correction and exit conveyers will both
                               'follow
ACCEL 10 10 * *                'conveyers will simply accel to ratio
VEL 5 * * *                    'super-imposed correction velocity
FOL M_SYNC TRIG2 * * *         'correction conveyer master sync mark
FOL S_SYNC TRIG1 * * *         'correction conveyer slave sync mark
MATH Q1 = 0                    'initial expected synchronization offset
FOL SYNC_OFF Q1 * * *          'expected synchronization offset
MOVE SLEWCW SLEWCW * *         'start the conveyers to track infeed
LABEL NEXT_P                   'main loop for each product
FOL WAIT TRIG1 * * *           'wait for product on correction belt
IN Q5 = FOL AXIS1 SYNC_ERR     'read correction amount
FOL SHIFT Q5 * * *             'perform correction move
IF BIT1 = 1 GOTO SHF_CW        'check for CW shift command
IF BIT2 = 1 GOTO SHF_CCW       'check for CCW shift command
GOTO NEXT_P                    'no shifts, wait for next product
LABEL SHF_CW                   'routine to adjust target move CW
VEL .1 * * *                   'use low shift velocity
IN Q2 = FOL AXIS1 SHIFT        'get current net shift
FOL SHIFT CW * * *             'start shifting
LABEL CW_LOOP                  'tight loop while checking input
IF BIT1 = 1 GOTO CW_LOOP       'shift as long as input active
GOTO SHF_OK                    'go to common exit
LABEL SHF_CCW                  'routine to adjust target move CCW
VEL .1 * * *                   'use low shift velocity
IN Q2 = FOL AXIS1 SHIFT        'get current net shift
FOL SHIFT CCW * * *            'start shifting
LABEL CCW_LOOP                 'tight loop while checking input
IF BIT1 = 1 GOTO CCW_LOOP      'shift as long as input active
LABEL SHF_OK                   'common routine after shifting
FOL SHIFT STOP * * *           'stop shift now
IN Q3 = FOL AXIS1 SHIFT        'get new net shift
MATH Q3  =  Q3 - Q2            'change in net shift
MATH Q1 = Q1 - Q3              'change in expected offset
FOL SYNC_OFF Q1 * * *          'new expected synchronization offset
VEL 5 * * *                    'super-imposed correction velocity
GOTO NEXT_P                    'wait for next product
```

Web Processing

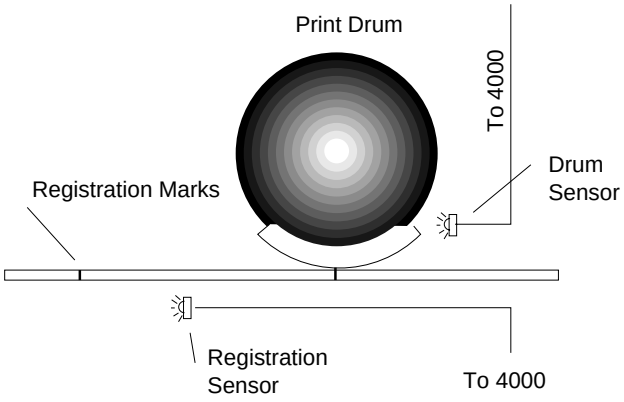
In a typical web processing application the master is the web which has periodic registration marks, and the slave drives a rotary print drum. The drum has a raised print portion which contacts the web during a certain segment of its rotation. During this portion, the drum surface speed must match the web speed, and the print portion must be aligned with the registration mark on the web. During the remainder of the drum rotation, the drum may rotate at a higher or lower ratio, depending on the cycle length of the pattern on the web below. During that portion, the drum may be shifted to correct for errors in registration.

A program could be developed which could handle variable web cycle lengths and print portion lengths, making extensive use of Q variables in the following parameters. For clarity in this example, however, we will assume a fixed drum circumference of 12 inches, and a print portion of 4 inches. The nominal distance between registration marks (web cycle length) will be 19 inches, requiring that the slave ratio be lower than 1:1 during the non-print portion of the cycle. The slave ratio changes will take place twice per cycle, each over 1 inch of web travel. As a result of these numbers, the non-print

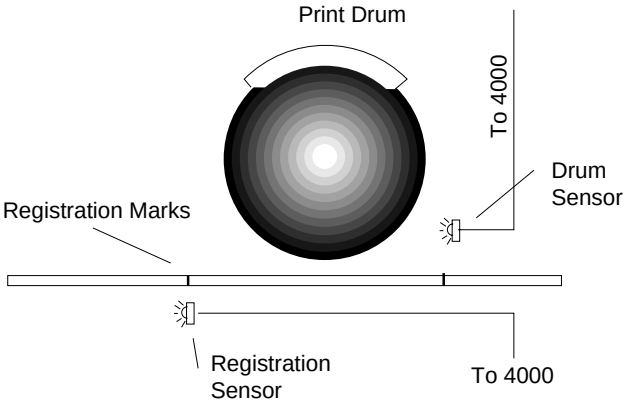
portion of the slave travel takes place at a ratio of 0.5:1 and covers 6.5 inches of drum circumference. If no registration correction is required, this 6.5 inches of drum travel occurs over 13 inches of web. If the registration spacing varies from its nominal 19 inches, however, a shift in drum position must be introduced to compensate.



Drum is in the center of 1:1 constant ratio print portion when drum sensor occurs



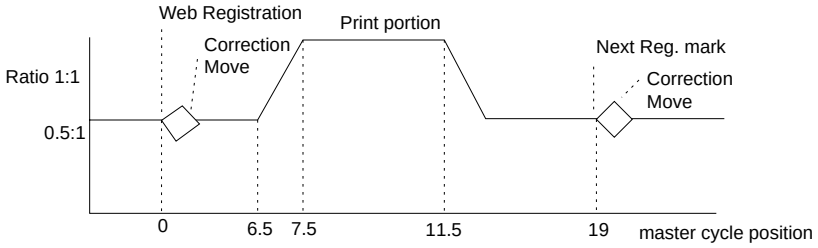
Drum is near the center of non-print portion when web registration occurs, and correction made



The slave sync mark will be a sensor connected to TRIG1 and mounted to detect the leading edge of the print portion of the drum. It is mounted so that the edge is detected exactly halfway into the print portion of the cycle. By then, the drum is well into the constant 1:1 ratio phase of its rotation. This ensures that the sensor measures the current corrected alignment of the drum, not its alignment during the correction portion of the previous cycle. The master sync mark will be a sensor connected to TRIG2 and mounted to detect the web registration mark exactly one half web cycle after the leading edge of the print portion of the drum is detected.

With this arrangement, perfect registration would result in the TRIG1 and TRIG2 occurring one half cycle apart, and an expected synchronization offset of negative 6 inches, just one half of the drum circumference. This value could be modified slightly during an initial alignment procedure to compensate for misalignment in mounting the upstream master sync mark sensor. If the registration mark arrives early, the sync error will be positive, and if the mark is late, the sync error will be negative. The sync error for the next cycle may be read as soon as the web registration mark in received. By that time, the drum will be near the center of the non-print portion of the cycle, and positive or negative corrections may be made safely. The sync error may be used in a subsequent **`FOL SHIFT`** statement to correct the drum alignment.

The sensor which detects the web registration will double as the input which is used to define the beginning of a master cycle. This master cycle reference in turn defines when the drum should change ratios during the cycle. In this example, the web registration mark should be detected exactly halfway through the non-print portion of the drum cycle. At that time, the master cycle is defined with an initial position of zero. If the web sensor is slightly misaligned, the initial master cycle position established with **FOL CYC_OFF** could be changed to a small non-zero value. When this mark is received, the drum alignment is corrected for the next cycle and the master cycle definition takes place. The remainder of the drum cycle ratio changes take place at master cycle positions which represent positions within that web cycle.



The sensor on the drum will double as a registration input which is used for the initial positioning of the drum. The registration distance is calculated so that if the drum begins it ramp to the 0.5:1 ratio when the first web mark is detected, the initial alignment will be correct. If the sensors are mounted correctly and registration is perfect, the web registration mark will normally occur exactly 6 inches of drum travel after the drum sensor, with the drum already at 0.5:1 ratio. During startup however, the drum will only travel 0.25 inches during the ramp to ratio, so the registration distance must be 6.25 inches.

The example below illustrates the use of periodic master/slave synchronization statements for the situation described above. For the sake of brevity, assume all units are in inches, an encoder which measures the web is the master, and the velocity and acceleration are appropriate for the slave shift. Assume that the initial drum positioning has already occurred, and that the drum is now stationary and waiting for the first web registration mark.

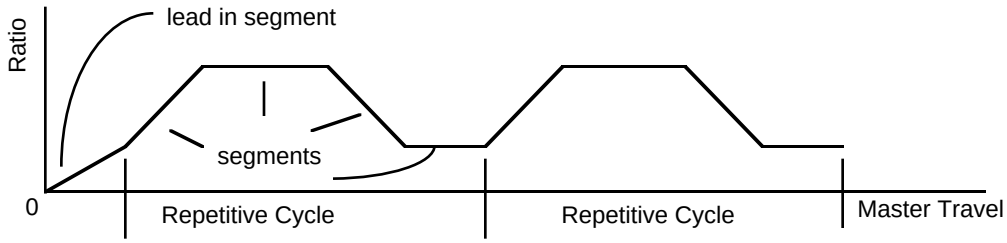
```
FOL MASTER ENC4 * * *          'use encoder 4 as master input
FOL RATIO 0.5:1 * * *          'track master initially at .5:1 ratio
FOL ENABLE YES * * *           'drum will follow web
FOL M_SYNC TRIG2 * * *         'master sync mark on web
FOL S_SYNC TRIG1 * * *         'slave sync mark on drum
MATH Q1 = 6                    'initial expected synchronization offset
FOL SYNC_OFF Q1 * * *          'expected synchronization offset
MATH Q2 = 0                    'initial master cycle position
FOL CYC_OFF Q2 * * *           'assume initial cycle position
FOL MAS_CYC 0 * * *            'zero cycle length ensures long cycle
FOL NEWCYC TRIG2 * * *         'cycle starts on TRIG2
FOL MDIST 1 * * *              'all ramps over 1 inch of web
FOL MOVEWT TRIG2 * * *         'sync startup move with first reg mark
MOVE SLEWCW * * *              'start the drum, no initial correction
LABEL NEXT_P                   'main loop for each product
FOL MOVEWT 6.5 * * *           'wait for correct position to change ratio
FOL RATIO 1:1 * * *            'new ratio of 1:1 during print
MOVE SLEWCW * * *              'make the ratio change at 6.5 web inches
FOL MOVEWT 11.5 * * *          'results in 4 inches of 1:1 ratio
FOL RATIO 0.5:1 * * *          'new ratio of 0.5:1 for non-print portion
MOVE SLEWCW * * *              'make ratio change at 11.5 web inches
FOL NEWCYC TRIG2 * * *         'cycle starts again at TRIG2
FOL WAIT TRIG2 * * *           'wait for the new cycle
IN Q5 = FOL AXIS1 SYNC_ERR     'read correction amount
FOL SHIFT Q5 * * *             'perform correction move
GOTO NEXT_P                    'no shifts, wait for next product
```

Cam Profiling

Some following applications will require very rapid cycle times with several ratio changes during a cycle. It may not be possible to execute individual

statements that are required for ratio changes rapidly enough to meet the cycle time requirements. For these cases, it is useful to be able to pre-define a profile of master and slave position relationships and ratios. This profile is defined with statements before it is actually run, and it is saved in a compiled form. In this sense, it is the same as a single complex move command. Only a single statement is required to start the move, when the profile is run, it is not affected by the execution speed of concurrent program statements.

A cam *profile* consists of a set of *segments* that describe the motion of the slave (the cam) pertaining to the motion of the master over a specified range of travel. A profile is constructed with sequential motion segments. Each segment describes a portion of the overall profile with data for master and slave travel and segment end ratio. Because the segments are sequential, the ending ratio of each segment will be the starting ratio of each subsequent segment. The starting ratio of the first segment defined will automatically be zero. The final ratio of the last segment may be non-zero if desired. The resulting profile has no sudden changes in ratio, and therefore no sudden changes in slave velocity.



These profiles may be executed as one-shot moves, or as repetitive cycles, as appropriate for the application. Progress through the profile may take place either forwards or backwards, following the direction of the master. If the profile is executed as a repetitive cycle, it is possible to designate some segment other than the first segment as the start of the repetitive portion of the profile. This allows one or more *lead-in* segments to precede a repetitive pattern. Once master travel has gone beyond the lead in segment(s), repetition will take place only within the repetitive cycle, even if the master moves backwards.

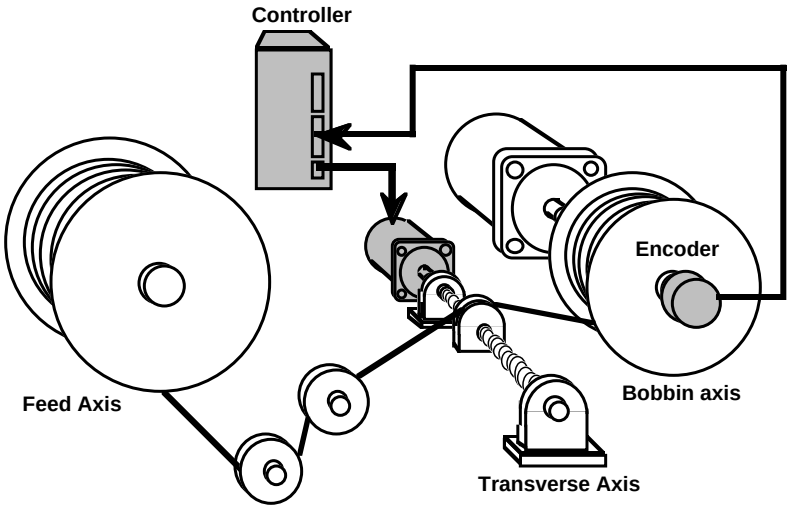
Profiling Applications

Several different types of applications benefit from the rapid cycle times available with cam profiling and the ability to progress either forward and backward through the profiles. Included in these are mechanical cam replacements, rotary knife, and coil winding. Each of these applications uses the concepts of master travel, slave travel, and ratio, but they correspond to different physical parameters for each application.

The term *cam profiling* is derived from the fact that many applications that require motion along a prescribed profile have been solved with the use of mechanical cams. These are often rotary cams, which have no beginning and no end. For an application that replaces a rotary cam, the profile is designed to execute repetitively. The angular travel corresponds to master travel, the radius at a given angle corresponds to slave position, and the slope of the cam surface for a given angle corresponds to ratio. In designing a profile that replaces an actual mechanical rotary cam, it is important that the final ratio and final position of the profile match the starting ratio and starting position of the first repetitive segment. A mismatch in these *ratios* would result in an abrupt change in velocity at the end of the cam cycle. A mismatch in these *positions* would result in a net position change for each cycle execution. A graph of slave versus master *positions* for a repetitive cycle of a rotary cam profile would have no sharp corners, and the starting and ending slopes and slave positions would be the same.

Another application that uses repetitive profiles with no net slave position change is coil winding. A typical configuration for winding is called bobbin

winding. The bobbin is a core onto which one or more layers of wire or filament is wound. The bobbin is usually the master, and its motion is measured in *turns*. The slave is a wire guide that traverses back and forth along the length of the coil. In this case, the ratio of the traverse of the wire guide to the rotation of the bobbin determines the pitch of the wind on the coil. The pitch is expressed in turns per inch, and would be the inverse of the ratio if the bobbin were the master.



A repetitive cycle of the profile would be two complete layers, one down the length of the coil and another back. By using a cam profile, complex patterns may be wound reliably at high speeds. These might include variable pitch within a layer, or special end of layer variations to accommodate the reversal of the wire guide. Layers could be designed to be offset by one half of the width of the wire, to tightly pack the layers. A graph of slave versus master *positions* for several repetitive cycles of a coil winding profile may look like a zigzag with rounded corners. Like the cam profile, the starting and ending slopes and slave positions would be the same.

By contrast, there are many applications that require a repetitive profile, but require a net change in slave position from the start to the end of a cycle. A rotary knife that makes periodic cuts in a moving web is a good example. The word *web* is a general purpose term for a continuously fed material. It may be paper, glass, plastic, or many other types of material. The web travel corresponds to master travel, the rotary knife position around the circumference corresponds to slave position, and the ratio refers to the ratio of knife travel to web travel. This is similar in concept to the web printing application described earlier. During the cut portion, the speed of the knife tip must match the speed of the web. During the remainder of the cycle, the knife must go through a profile of changing ratio to match the cut length with the circumference of the wheel. In designing a profile that controls a rotary knife, it is necessary for the final slave position of the final repetitive segment to be one full revolution from the starting slave position of the first repetitive segment. This allows continuous rotation of the knife as the web material moves through its cut cycles. If the slave cycle length is off even by one step, the cut portion of the profile will gradually drift away from the required physical position. A detailed example of a rotary knife application is given at the end of this section on *Cam Profiling*.

This software allows one profile per axis to be defined and stored. Once a profile is defined, it may be executed repeatedly without re-definition. The stored profile is automatically deleted if another profile is defined on that axis. This arrangement is similar to segmented moves on the 4000. The profiles are stored in the same data area used for **PATH** storage, therefore, no **PATH**s may be defined if any profiles are defined. Definition of a cam profile on an axis begins with the **FOL CAM YES** statement on that axis. To delete a profile, the **FOL CAM NO** statement must be executed. As soon as a cam profile is defined on any axis, all existing **PATH** definitions will be deleted. The resulting RAM is divided into four sequential blocks of 120 segments each for cam profile storage, one for each axis. This allows a maximum of 120 segments for each axis, if all four axes have profiles defined. It is possible, however, for the segment storage of a lower numbered axis to continue into the area allocated for a higher numbered axis. This flexibility allows better use of the RAM when fewer than four axes have profile definitions. For example, axis 1 could use all 480 segments, or use 360 segments with 120 left for axis 4. Many other permutations are possible, but all involve blocks of 120 segments.

Each segment describes sequential portions of the overall profile and contains three pieces of data. They must be given with profiling enabled (**FOL CAM YES**), *and* following enabled (**FOL ENABLE YES**). The first is the master distance (not position) over which the segment motion takes place. The **FOL MDIST** statement establishes this value for the axis on which a profile is being defined. The second is the ending ratio of that segment. The ratio value may be positive or negative, and is established with the **FOL RATIO** statement. The third is the slave distance or final slave position of that segment. It is specified with a **MOVE** or **MOVI** statement containing a distance or position. As long as profiling and following are enabled, these statements define segments but do not cause motion. If either profiling or following are not enabled, these statements will cause motion. It is this slave distance or position specification that causes the 4000 to compile and save the segment, using the previously established master distance and ratio values. The segments are compiled and linked one segment at a time, no separate statement is required for compilation.

The master and slave data for each segment of a profile are stored as starting and ending positions, from which the distances are calculated internally. This allows a profile to be started at any master and slave position, and facilitates repetitive cycles. When specifying the slave *positions* for the profile, however, the current position of the slave is important. If the slave is in an absolute mode (i.e., **M_ABS** or **E_ABS**) the value in the **MOVE** or **MOVI** statement is interpreted as the commanded segment end *position*. In this case, the slave's current position and commanded segment end position are used to calculate the distance for the profile's *first segment only*. Subsequent slave distances will be calculated using the previous commanded segment end position and the new commanded segment end position. In many cases it will be convenient to give the first slave position data as zero, even if the current position is not zero. In this case, a **PDEF** statement may be used before defining the profile, or the slave data may be offset by the current position using **IN** and **MATH** statements. If the slave is in an incremental mode (i.e., **M_INC** or **E_INC**) the value in the **MOVE** or **MOVI** statement is interpreted as the commanded segment *distance*. In this case, the slave's current position does not affect profile definition.

Profiles and Master Cycles

When profiling is enabled, the commands that establish master cycle parameters outside profiling are either ignored, invalid, or have new meaning. Please refer to the section in this chapter titled *The Master Cycle Concept* for a discussion of master cycles. The previous discussion has referred to lead in segments that precede the repetitive portion of a profile. The default number of lead in segments is zero, i.e., the entire profile is designated as the repetitive portion. This default takes effect when profiling is enabled, but any segment may be designated as the start of the repetitive portion. This is done by preceding that segment with the **FOL NEWCYC IMMED** statement. This does not immediately initiate a new master cycle. Instead, it indicates which segment of a profile will initiate a new master cycle. The sum of the master distances in the segments preceding this statement constitutes a negative value for master cycle offset position (**FOL CYC_OFF**), and the sum of the master distances in the repetitive segments after this statement constitute a value for master cycle length (**FOL MAS_CYC**). When executing a profile these parameters become defined automatically, and the repetitive portion of a profile will automatically constitute a master cycle. This allows the master cycle number to indicate the number of profile cycles that have been completed, and the master cycle position to indicate the progress into the current profile cycle. The fact that it is done automatically eliminates the application programming that would be required to calculate the values, and allows the master cycle to be synchronized with the start of the repetitive profile.

Executing a Profile

Execution of the profile is accomplished with the **MOVE SLEWCW** or **MOVE SLEWCCW** statements for one shot or repetitive executions respectively. The same profile may be executed in either manner. When these statements are given, the currently constructed profile is executed as is. Because a profile is not deleted until a **FOL CAM NO** statement is given, it is possible to add more segments to a profile even after having executed it. The profile may be started at a specific trigger by using the **FOL MOVEWT** statement. This is useful in synchronizing the start of two axes that are both profiling from the same master. The **FOL MOVEWT** may *not* use master cycle position as parameter when profiling is enabled, because the master cycle position is not defined until the profile is started.

Execution may take place forwards and backwards through the profiles. If the profile is started with **MOVE SLEWCW**, it is executed only once, and the profile is not complete until the master position reaches the end of the profile. If the master moves backward before the end of the profile is reached, the profile will also execute backwards, but only up to the beginning of the profile. If the profile is started with **MOVE SLEWCCW**, the lead in segments is executed as described in one shot mode. The last lead in segment (if any) links to the remainder of the profile, which is executed repetitively. If the master moves backward, the profile is executed backward until the beginning of the first repetitive segment. It then loops back to the end of the last repetitive segment. If the master moves forward, the profile is executed forward until the end of the last repetitive segment. It then loops back to the beginning of the first repetitive segment.

Statements Affected by Cam Profiling

Enabling cam profiling via the **FOL CAM YES** statement affects several statements. All master cycle definition statements are ignored except **FOL NEWCYC IMMED**. That will define the start of the repetitive portion of a profile, and **FOL NEWCYC TRIGn** will result in an error. It also changes the meaning of following mode (i.e., **FOL ENABLE YES**) **MOVE**s to define or execute a profile. **MOVE** statements executed out of following mode (i.e., **FOL ENABLE NO**) will be normal time based moves, even if profiling is enabled. This is useful for repositioning a slave between profile executions without losing the profile. The most recent values of **FOL MDIST** and **FOL RATIO** become the master travel and ending ratio values respectively for a segment. Negative ratios may be specified with **FOL RATIO**, and the sign is used to determine the direction of motion in that segment. In normal following moves, only the magnitude of the ratio is recorded, and the direction is determined by **SLEWCW** or **SLEWCCW** in the **MOVE** or **MOVI** statement. The following table shows the method of deleting, building and executing a profile.

Use of Statements in Cam Profiling

Statements	Function	Description
FOL CAM NO	Delete profile	If profiling is already disabled on this axis, the statement is ignored. If profiling is enabled on this axis, the profile is deleted.
FOL CAM YES	Enable profiling	If profiling is already enabled on this axis, the statement is ignored. If profiling is disabled on this axis, the profile is initialized
FOL MDIST # **FOL RATIO #** **MOVE # or Q**	Create a segment	These 3 statements provide the master travel, final ratio, and slave position or distance of the segment.
FOL NEWCYC IMMED	Define repetitive cycle	This marks the beginning of the repetitive portion of a profile.
FOL MDIST # **FOL RATIO #** **MOVE #**	Add another segment, etc.	These 3 statements provide the master travel, final ratio, and slave position or distance of the segment.
MOVE SLEWCW	Perform profile once	If the **MOVE** parameter is **SLEWCW**, the profile is performed once, i.e., the **MOVE** is complete when enough master travel has occurred to finish the profile.
MOVE SLEWCCW	Perform profile repetitively	If the **MOVE** parameter is **SLEWCCW**, the lead in segments are performed once, and the remainder of the profile is performed repetitively. When enough master travel has occurred to finish the profile, the profile starts over from the first repetitive segment if the master continues to move.
FOL MDIST # **FOL RATIO #** **MOVE #**	Add another segment, etc.	Even if the profile has already been performed, additional segments may be added to the end of the profile.

Practical Profile Design Issues

Some practical precautions must be taken when designing the cam profile. As discussed earlier, the starting and ending ratios of repetitive cycles must match to avoid abrupt changes in slave velocities. If there is no net change in slave position over the cycle, the starting and ending positions of repetitive cycles must also match.

The use of lead in segments allows a graceful entrance from zero ratio into a cycle that never comes to rest. This especially useful if the profile is to be started when the master is already in motion. If the master will not be in motion when the profile starts, it may be desirable to have the profile start with a specific non-zero ratio. This can be effectively accomplished with a lead in segment that has zero master travel, the specific final ratio, and zero slave distance. This segment has no effect other than initalize the starting

ratio of the subsequent segment. Although it takes zero master travel to finish this segment, some internal execution time is required to process it. These types of segments should not be constructed sequentially. Also because of process time considerations, the segments should average at least 8 milliseconds of travel at the maximum master speed.

For a given segment, the user effectively specifies the starting and ending ratios as well as both the master and slave travel of a segment. This means that the segment's average ratio (slave travel divided by master travel) may not be equal to the average of the starting and ending ratios. To satisfy all the user specifications, the segment is broken internally into two halves, and an intermediate ratio goal is reached halfway through the master travel of the segment. The 4000 calculates this ratio internally using the formula below.

$$R_{mid} = \frac{(2*S - M*Rave)}{M}$$

where:

R_{mid} = Intermediate ratio
R_{ave} = Average of starting and ending ratio
S = Slave travel
M = Master travel

If the user's data results in a value for R_{mid} after scaling by **UNIT POS** and **UNIT MASTER** that is greater than 127 or less than -127, the profile will not execute properly, but no error message is generated. To avoid this, care must be taken in the design of the individual segments to ensure that intermediate ratio magnitudes do not exceed ±127.

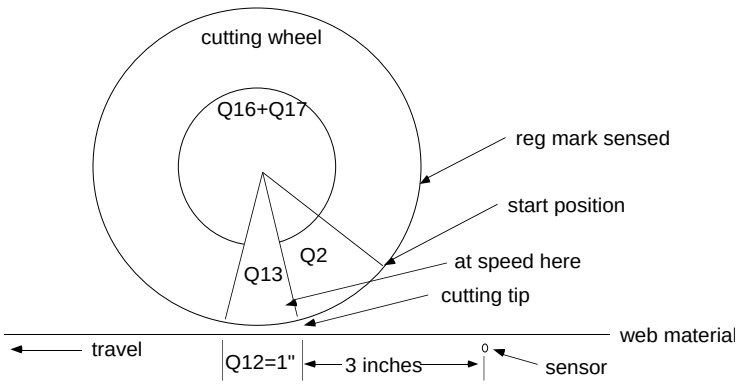
Another important precaution must be taken when cam profiling is used in applications that require a segment of constant ratio. Because all the user segments are divided internally into two segments, care must be taken in the specification of constant ratio segments. The starting and ending ratios must be the same, of course, but the master and slave travels must result in an average ratio that is equal to the starting and ending ratios. The requirement is:

S/M = starting and ending ratios
where:

S = Slave travel
M = Master travel

Rotary Knife Cut to Length

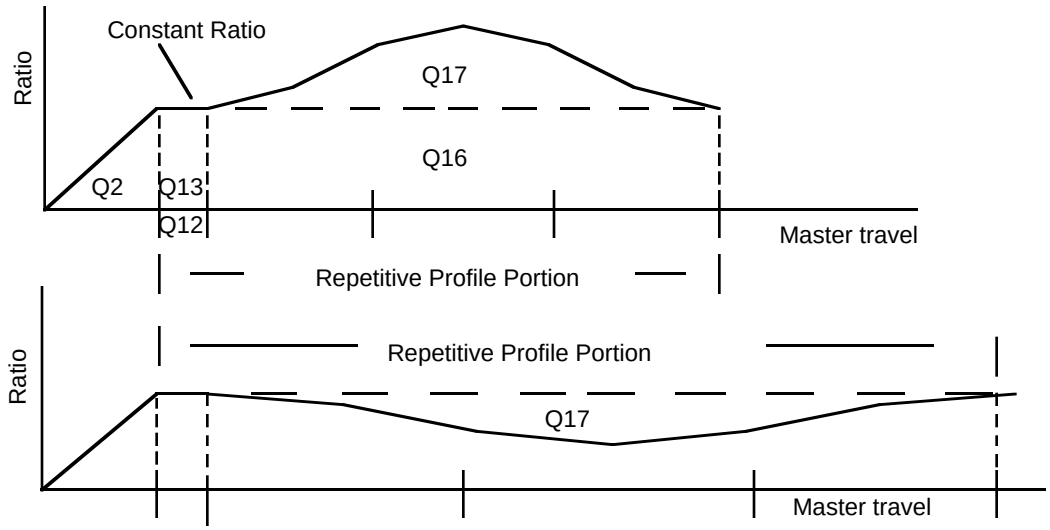
A continuous cut to length application uses a rotary knife to cut material moving under the knife. The motion of the material is measured with an encoder turned by a pinch roller on the material. To make a clean cut requires that the rotary knife match the material speed during 1 inch of the overall cut cycle. This prevents the knife from tearing or bunching the material. The circumference of the wheel at the tip of the knife is 25 inches, requiring that the knife tip move 24 inches during the remaining portion of cut cycle. The cut length varies from 10 inches to several feet, which means that a knife profile must be designed to match the knife circumference to the cut length. The material web is the master, and the knife is the slave.



The cutting wheel has a once per revolution sensor that is used for initial positioning of the wheel. This sensor is connected to trigger 3 and causes a registration move to bring the wheel to the start position. There is also a sensor mounted 3 inches away from the start of the cut position. It is connected to trigger 2 and is used to detect the leading edge of the material. The wheel is first positioned with the registration move, and then waits for the detection of the material. As soon as the material is detected, the wheel ramps to the cut speed and position. This occurs over 3 inches of material travel and Q2 steps of wheel travel. This positions the knife tip at the start of the material for the beginning of the repetitive portion of the cycle. The wheel distances are programmed in steps, to ensure that the sum of the steps in a cycle adds up to 25000. The material distances are programmed in inches, to allow the cut length to be entered in inches.

```
UNIT POS = 1            25000 step circumference
UNIT MASTER = 500       encoder steps per inch of material
```

The ratio during the one inch cutting portion of the cycle must be 1:1, expressed in terms of inches of knife travel to inches of material travel. The remainder of the cycle takes place over the cut length minus 1 inch. In the program below, this remaining master distance is called Q14. The average ratio during the remainder of the cycle must be (remaining 24 inch's circumference: Q14). The profiles shown below meet these requirements. The variable Q16 is the additional distance (beyond the 1 inch of constant ratio) the wheel would travel *if it stayed at constant ratio for the entire cut length*. This however, will usually result in the incorrect wheel cycle length. The variable Q17 is the component of travel required for the wheel to make one complete revolution during one cut length. The top profile illustrates the case in which the cut length is shorter than the circumference, and Q17 is positive. The bottom profile illustrates the case in which the cut length is somewhat longer than the circumference, and Q17 is negative. In both cases, the component of slave travel represented by Q17 requires acceleration and deceleration, and can be considered for this discussion as a separate motion profile.



This Q17 motion profile was designed to minimize the overall jerk, i.e., the magnitude of changes in acceleration. Low jerk allows the wheel to run smoothly, resulting in less servo drive oscillation and better accuracy in cut lengths. The Q17 profile uses three segments of equal master travel. The outer segments each command 1/5 of Q17, and the inner segment commands the remaining 3/5's of Q17. The 4000 automatically calculates the kinks in these segments to satisfy all the segment specifications. The resulting Q17 profile is a close approximation to a sine wave, which minimizes jerk.

The program below assumes that the slave is axis 1, and that the master is axis 2 encoder input. This allows axis 2 to be used as a controlled master for the purpose of demonstration. In actual application, the master may simply be measured, rather than controlled by the 4000. This application also takes

advantage of the automatic assignment of master cycle parameters during cam profiling. The master cycle length is simply the cut length, and the master cycle number is displayed as the number of cuts made.

```
UNIT POS 1 * * *                          'slave units in steps
UNIT VEL 25000 125 * *                    '
UNIT ACCEL * 25000 * *                    '
UNIT MASTER 500 * * *                     'encoder steps per inch
OUT LCD1,01 BLANK TO END OF LINE          '
OUT LCD2,01 BLANK TO END OF LINE          '
OUT LCD3,01 BLANK TO END OF LINE          '
OUT LCD4,01 BLANK TO END OF LINE          '
IN Q1 = LCD3,05 ^CUT SPEED? (inches/s) ^  'get cut speed
VEL 1 Q1 * *                              'cut speed
MATH Q3 = 2000                            'steps from reg mark to start
                                          'position
SEG REG3 Q3 * * *                         'defines reg move from TRIG3
ENABLE REG3 YES * * *                     'enable TRIG3 as  registration
                                          'input
MOVE SLEWCW * * *                         'find the reg mark
IN Q10 = LCD2,05 ^CUT LENGTH? (inches) ^  'get cut length
ENABLE REG3 NO * * *                      'disable TRIG3 as registration
                                          'input
MATH Q11 = 25000                          'total slave cycle (steps)
MATH Q12 = 1                              'master travel during cut
                                          'segment (inches)
MATH Q13 = 1000                           'slave travel during cut segment
                                          '(steps)
MATH Q14 = Q10 - Q12                      'master travel during non-cut
                                          'portion
MATH Q15 = Q11 - Q13                      'slave travel during non-cut
                                          'portion
MATH Q9 = Q15 * 1.9                       'intermediate calculation of
                                          'MRMAX
MATH Q9 = Q9 * Q12                        'intermediate calculation of
                                          'MRMAX
MATH Q9 = Q9 / Q13                        'final calculation of MRMAX
IF Q14 > Q9 GOTO LONGCUT                  'profile will come to rest
MATH Q16 = Q13 * Q14                      'intermediate calculation of
                                          'non-cut base
MATH Q16 = Q16 / Q12                      'base portion of slave non-cut
                                          'travel
MATH Q17 = Q15 - Q16                      'profile portion of slave non-
                                          'cut travel
MATH Q18 = Q16 / 3                        '1/3 slave base portion
MATH Q19 = Q14 / 3                        '1/3 master non-cut portion
MATH Q20 = Q17 / 5                        '1/5 slave profile portion
MATH Q20 = Q20 + Q18                      'intermediate calculation of
                                          'slave outer-profile seg
MATH Q23 = Q17 / 2                        'intermediate calculation
                                          'numerator of profile ratio
MATH Q23 = Q23 + Q18                      'final calculation numerator of
                                          'profile ratio
GOTO COMMON                               '
LABEL LONGCUT                             '
MATH Q19 = Q9 / 2                         'master outer profile seg
MATH Q22 = Q14 - Q19                      'intermediate calculation of
                                          'master mid-profile seg
MATH Q22 = Q22 - Q19                      'final calculation of master
                                          'mid-profile seg
MATH Q20 = Q15 / 2                        'final calculation of slave
                                          'outer-profile seg
MATH Q23 = 0                              'final calculation numerator of
                                          'profile ratio
LABEL COMMON                              '
MATH Q20 = Q20 / 10000                    'truncate all fraction
MATH Q20 = Q20 * 10000                    're-create integer
MATH Q21 = Q15 - Q20                      'intermediate calculation of
                                          'mid-profile seg
MATH Q21 = Q21 - Q20                      'final calculation of slave mid-
                                          'profile seg
MATH Q22 = Q14 - Q19                      'intermediate calculation of
                                          'master mid-profile seg
MATH Q22 = Q22 - Q19                      'final calculation of master
                                          'mid-profile seg
MATH Q2 = 1500                            '1.5  inch of tip travel during
                                          'ramp to ratio
MATH Q3 = 3                               '3 inches master travel during
                                          'ramp to ratio
OUT LCD3,01 BLANK TO END OF LINE          '
```

```
OUT LCD4,01 BLANK TO END OF LINE          '
OUT LCD3,01 ^Press when knife is          '
stopped^
OUT LCD4,01 ^    OK ^                     '
WAIT FOR F-KEY1 TO BE PRESSED             '
OUT LCD3,01 BLANK TO END OF LINE          '
OUT LCD4,01 BLANK TO END OF LINE          '
FOL MASTER ENC2 * * *                     'use axis 2 encoder input for
                                          'master.
FOL SMOOTH 3 * * *                        'this may require trial and
                                          'error
FOL ENABLE YES * * *                      'moves will be following type
FOL CAM YES * * *                         'generate cam profile
                                          '
FOL MDIST Q3 * * *                        'ramp from rest to ratio over Q3
                                          'inches glass
FOL RATIO Q13:Q12 * * *                   'ratio while tip contacts glass
MOVE Q2 * * *                             'create seg slave travel during
                                          'ramp
                                          '
FOL NEWCYC IMMED * * *                    'first repetitive cycle starts
                                          'on here
                                          '
FOL MDIST Q12 * * *                       'cut portion over Q12 inches
FOL RATIO Q13:Q12 * * *                   'ratio while tip contacts glass
MOVE Q13 * * *                            'create seg slave travel during
                                          'cut
                                          '
FOL MDIST Q19 * * *                       'outer profile seg portion over
                                          'Q19 inches
FOL RATIO Q23:Q19 * * *                   'final ratio of outer segment
MOVE Q20 * * *                            'create seg slave travel during
                                          'outer seg
                                          '
FOL MDIST Q22 * * *                       'mid profile seg portion over
                                          'Q22 inches
FOL RATIO Q23:Q19 * * *                   'final ratio of outer segment
MOVE Q21 * * *                            'create seg slave travel during
                                          'outer seg
                                          '
FOL MDIST Q19 * * *                       'outer profile seg portion over
                                          'Q19 inches
FOL RATIO Q13:Q12 * * *                   'final ratio of outer segment
MOVE Q20 * * *                            'create seg slave travel during
                                          'outer seg
                                          '
FOL MOVEWT TRIG2 * * *                    'first ramp starts on TRIG2
MOVI SLEWCCW * * *                        'start profile
MOVE * SLEWCW * * *                       'start master
OUT LCD3,01 ^Simulate detection of stock  '
TRIG2^
OUT LCD4,01 ^Use BIT1 = 1 to stop ^       '
FOL WAIT TRIG2 * * *                      '
OUT LCD3,01 BLANK TO END OF LINE
OUT LCD4,01 BLANK TO END OF LINE

LABEL NEXT_CUT                            'start of cycle loop
IF BIT1 = 1 GOTO EXIT                     'signal to exit cut loop
IN Q25 = FOL AXIS1 MAS_C                  'read cycle count
OUT LCD4,01 ^Cycle count is: ^ Q25
GOTO NEXT_CUT                             'do next cut when length passes

LABEL EXIT
MOVE * STOP * * *                         'stop master
MOVE STOP * * *                           'stop profile
DONE
*
```

The application example and program above can be briefly summarized. The main reason the cutting wheel can achieve high cycle rates with smooth motion is that the entire profile is defined and compiled before motion ever starts. The profile is also designed to minimize the jerk, i.e., changes in acceleration required, to match the wheel cycle with the cut length. The three segments that accelerate and decelerate the wheel roughly approximate a sinusoidal wave form, and all the math in the program is used to calculate the distances required for this.

The questions and answers below may be typical for this type of application.

Question	Answer
Is it possible to change the cut length on the fly, i.e, while moving?	It is not possible to change cut length on the fly. Everything about the profile is pre-compiled, including the master travel (i.e., cut length) over which the profile takes place. To change cut length, you must stop and compile a new profile.
How can I stop the profile at a pre-defined spot?	The only way to stop at a pre-defined spot is with a registration input. The program above already makes use of a registration input on the wheel for initial positioning. Define an **`ON STOPK`** destination in front of the main portion of the program. In the destination routine, simply enable registration on the wheel sensor input, and the registration move will exit the profile at the desired position. It is not possible to stop at a predefined spot as part of the pre-compiled profile definition.
Will there be any drift due to roundoff errors?	There will never be any drift due to roundoff errors. The profile is designed in terms of segments, as shown above. Each segment has the master travel in inches, and the exact travel of the slave in terms of steps. The sum of slave travel for the segments adds up to exactly 25,000. This was ensured by truncating any fractional results of all segments but the middle segment, and calculating a subsum. The program then assigns the middle segment a distance of 25000 less the subsum of the others. During execution, the 4000 adjusts its setpoint at the end of each segment, eliminating any accumulated roundoff error during that segment.
How can I synchronize a second identical wheel to do a sealing operation downsteam from the cut?	This wheel would use the same profile, only delayed to accommodate the physical separation from the first wheel. The profile for this second wheel could be designed using the same program and numbers, but would have one more lead in segment than the first wheel. This additional lead in segment would have zero slave travel and end in zero ratio, but would specify a master travel equal to the physical separation from the first wheel. This essentially *dwells* for a master distance.
Can I define a new profile on one axis while I'm running on another?	Yes. Because the profiles are pre-defined, only a single **`MOVE`** or **`MOVI`** statement is required to get them started. Because a repetitive profile never finishes, a **`MOVE`** statement would never finish. To simply start a profile and continue program execution, use a **`MOVI`** statement. Once a profile is started, no further statement execution is required for their continued operation. The program could be doing anything else, including definition of a profile on another axis.

Moving Positioning System

Up to this point, the discussion has been related to how the slave follows the master. The point of view has been that of a stationary observer watching both the master and slave move. With a ratio of 1:1, the two move together. This is referred to in the following discussion as the stationary positioning system (not moving means no motion at the motor). Suppose instead that the observer is moving with the master, watching the slave. From this point of view, the slave is not moving at all. This viewpoint is the moving positioning system (MPS). Standard moves which can be super-imposed on the MPS include point-to-point, contouring, and even ratio following. Implementation of the moving positioning system is so easy because each of the other moves is programmed as if in the standard positioning system.

In the stationary positioning system, moves may be either continuous or preset, with both types requiring acceleration and velocity specifications. The same is true in the moving positioning system, except that the velocity and acceleration specifications are concerning the moving positioning system, not a stationary point. In both positioning systems, preset moves may be either incremental or absolute. In the stationary positioning system, the absolute position is defined as zero on power-up, and defined elsewhere with the **PDEF** statement. The **MOVE HOME** statement is often used to establish the zero position based on a known physical location. In the moving positioning system, the **PDEF** statement may also be used to establish absolute position, as long as the slave is at rest within the MPS. No **MOVE HOME** statements may be issued while in the MPS, but initial absolute positions within the moving positioning system can be based on trigger inputs.

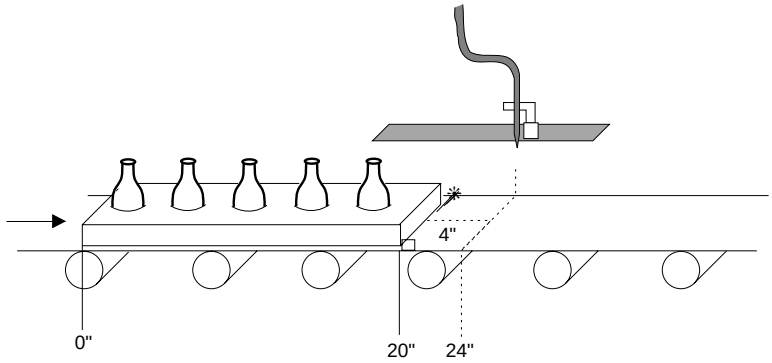
Defining and Entering the Moving Positioning System

The MPS can best be described with an example. Suppose the master represents a conveyor belt that holds trays of bottles, and the slave is controlling the position of a dispensing nozzle. From the *tray's* point of view, the nozzle must be at rest when filling the bottles. The application is such that the nozzle moves to the first bottle, stop and fill the bottle, move to the next bottle, stop and fill the bottle, etc. This is point-to-point positioning with output control, except that the positioning is being performed within the moving positioning system, not a stationary positioning system. The bottle filling machine must continually accept new trays of empty bottles at one end of the belt, and fill the bottles as the tray moves under the filling nozzle.

We have already noted that for the nozzle to be at rest in the MPS, it must be following the tray (the master) at a 1:1 ratio, moving in the same direction as the tray. To move to a particular position on the tray, it must also have established a position reference. The **FOLM RATIO** command forms one part of the definition of a moving positioning system, i.e., how to stay at rest in the MPS. Establishing the absolute position reference is the other part of the MPS definition. The **FOLM DEF** statement allows a trigger input to be used to establish the master and slave position reference. In the bottle filling example, trigger #1 is connected to a sensor that detects the leading edge of the tray and this initiates the recording of master and slave positions in the MPS.

The nozzle starts out at rest in the stationary positioning system as the tray approaches on the conveyor. The sensor is located 4 inches away from the nozzle, towards the approaching tray. The **FOLM PDEF** statement allows definition of the slave's position at the time the MPS is defined. If the moving positioning offset, **FOLM PDEF**, has not been defined for the slave, the initial slave position when the MPS is defined will be zero. When the tray gets to the position sensor, the 4000 defines the moving positioning system by reading the position counts of both the master and the slave, and the slave's position is set to that defined with **FOLM PDEF**. In this example,

since the tray is 20 inches long and the sensor detecting the tray is 4 inches from the nozzle, the slave's initial position will be defined as 24 inches. This way all position references will be with respect to the far edge of the tray, or position 0.



Once the MPS is defined, in this case when the trigger is made, master and slave positions are read, but the 4000 has not entered the reference point of view of the tray. The 4000 has separate commands for defining and entering the MPS. This allows switching between the moving and stationary positioning systems at will without losing position in either reference frame. The moving positioning system is entered with the **FOLM ENABLE** statement. After this is issued, all subsequent slave moves will be with reference to the defined MPS.

In the case of the bottle filling machine, the program will be designed so that the MPS is entered immediately after it is defined. The nozzle is at rest in the stationary reference frame when the trigger is sensed, but it is moving toward the tray in the moving reference frame. When the switch is made from stationary to moving positioning system, *the nozzle enters the MPS at a non-zero velocity with respect to that system, even though the motor shaft of that axis does not start to rotate. While in the MPS, to* ***stop*** *or position means come to rest with respect to the moving reference.* For the slave to stop within the MPS and track the master at the predefined ratio, a **MOVE STOP** must be issued, or the slave must be commanded to some position within the MPS. In this case it will stop with respect to the tray and track the master after the commanded position is reached.

The example above illustrates that when a moving positioning system is defined, there are two sets of positions to which the slave may be sent. Positions with respect to a stationary reference may be commanded when the slave axis is in the stationary positioning system (i.e., **FOLM ENABLE NO**). In this case, the **IN Qn = FOL AXISn SLV_P** statement will read a stationary position. Positions with respect to the moving reference, for example, the tray of bottles, may be commanded when the slave axis is in the moving positioning system (i.e., **FOLM ENABLE YES**). In this case, the **IN Qn = FOL AXISn SLV_P** statement will read a position with respect to the moving reference. In other words, the **FOLM ENABLE YES/NO** statement changes context for the position request as well as the positioning command.

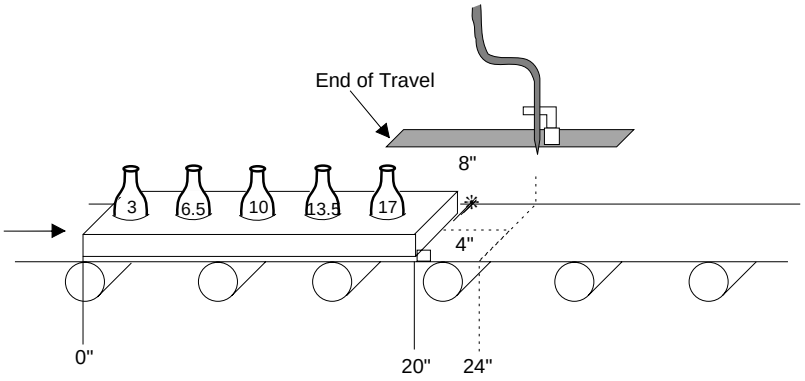
Below is a summary of the moving positioning system commands required for the bottle filling machine, assuming axis #1 is the axis controlling the nozzle:

```
FOLM RATIO 1:1 * * *    'Set slave to master ratio at 1:1, assuming unit
                        'scale factors previously defined
FOLM PDEF 24 * * *      'Slave position is 24 inches at time of MPS
                        'definition
FOLM DEF TRIG1 * * *    'Define the MPS on trigger #1
FOL WAIT TRIG1 * * *    'Halt program operation until trigger #1 active
FOLM ENABLE YES * * *   'Enable moving positioning system
```

Event Coordination and Master Cycle Positions

The fact that moves are simply super-imposed on ratio following makes programming within the MPS easier because slave profiles do not need to be calculated as a function of master position. There are situations, however, when slave event timing must be coordinated with respect to master position. One situation arises from limited travel on the slave axis.

☛ In this example the slave axis controls the nozzle position and velocity



In the bottle filling machine described above, the nozzle's initial position is 8 inches away from the physical end of travel closest to the tray. Suppose there are 5 bottles per tray, spaced every 3.5 inches, with the center of the first bottle 3 inches from the edge. At the time the MPS is defined, the slave is at position 24, moving toward the first bottle. After the MPS is enabled, the desired command sequence should simply move the nozzle to positions 17, 13.5, 10, 6.5, and 3, pausing at each long enough to fill the bottle. Notice, however, that if the nozzle's velocity is high enough, the nozzle may be commanded beyond its end of travel. To avoid this situation a master cycle of 24 inches should be defined using the same trigger input which defines MPS. Each slave's move should be preceded by a command to wait for a master position within that cycle which is 3.5 inches greater than the previous master position. This will ensure room for the slave's next move. While *waiting* for each master position, keep in mind that the slave will be tracking the master's velocity, remaining stationary in the moving reference frame, but moving away from the critical end of travel in the stationary reference frame.

The above need is accommodated by the ability to define a new master cycle upon the same trigger input which defines the MPS, and wait for positions within that cycle. The master cycle position and cycle count are set to zero when the new master cycle is defined (with the **FOL NEWCYC** statement). In the case of the bottle filling machine, the trays are not evenly spaced, so a new cycle is defined with each new tray. Once a tray is sensed, a new cycle begins and the MPS is defined. Waiting for positions within that cycle will take care of the limited travel situation described above.

The Multi-Axis Bottle Filling example below shows how the use of **FOL WAIT** statements solves the problem arising from limited travel on the bottle filling axis.

It is important to note that any command processing delay associated with the wait for a cycle position will have no effect on the positioning accuracy of the slave in the MPS. The position relationship between the master and slave remains locked while the slave waits for a master cycle position.

Summary of Moving Positioning System Statements

```
FOLM RATIO      'Defines the required ratio of slave to master
                'velocity for the slave to stay at rest in the MPS
FOLM PDEF       'Determines the slave's initial position in the MPS
FOLM DEF        'Establishes the moving positioning system at the time
                'of statement execution or upon a trigger input
FOLM ENABLE     'Enables or disables a slave's positioning within the MPS
```

The text above described in detail the relationship between the bottle tray and the fill nozzle, and the text below expands on that example to illustrate multi-axis coordination and the ability to position between moving and stationary positioning systems. In addition to filling each bottle, the machine must move the finished tray off the first belt onto a second moving belt. The second belt is moving perpendicular to the first belt at a velocity unrelated to that of the first belt.

Axis #1 controls the nozzle, and its motion requirements have already been described. Axis #2 and #3 move a clamping device that removes the tray from the first belt and deposits it onto the second belt. The 4000 program controlling the entire machine is described and listed below.

The motor resolution for all three axes will be 25,000 and the encoders will be 1000 line rotary encoders. Rack and pinion systems are used which yield one revolution per inch for both the motors and the encoders. This gives 25000 slave steps per inch and 4000 post quadrature master steps per inch. Programming units for master and slave positions will be inches, and slave velocity units will be inches/second.

It is important to note that when the nozzle is moving with respect to the moving positioning system, the defined velocity of 10 inches/second refers to the velocity of the nozzle with respect to the tray. Because the nozzle's moves will be in the negative direction and the tray is moving at 5 inches/second in the positive direction, the net velocity in the stationary positioning system will be 5 inches/second in the negative direction.

After defining the MPS parameters, the axes are sent home, positions are set to zero, and the nozzle and clamp outputs are turned off. After these statements are completed, the axes are enabled as slaves. The encoder which measures the speed for the first belt acts as a master to axes #1 and #2, and it is connected to encoder input #1 on the 4000. The encoder which measures the speed of the second belt is fed into encoder input #4 on the 4000, and acts as the master for axis #3. Stall detect is enabled on axes #2 and #3, with stalls detected by encoders mounted right on those motors.

```
UNIT POS 25000 25000 25000 *      'Slave scale factors set for steps/inch
UNIT MASTER 4000 4000 4000 *      'Master scale factors to encoder
                                  'steps/inch
UNIT VEL 25000 25000 25000 *      'Velocity scale factor set for inches/sec
UNIT ACCEL 25000 25000 25000 *    'Acceleration scale factor set to
                                  'inches/sec/sec
VEL 10 10 10 *                    'Velocity of 10 ips on all axes
ACCEL 20 20 20 *                  'Acceleration of 20 ips^2 on all axes
MOVE HOMECW  HOMECW  HOMECW *     'Start move towards home in CW direction
PDEF 0 0 0 *                      'Position at home set to 0
OUT POB 11XX                      'Set programmable outputs 1 and 2
FOL MASTER ENC1 ENC1 -ENC4 *      'Encoder input #1 to act as master for
                                  'axes 1 and 2, while encoder input on axis
                                  '4 to be master to axis #3.  Physical
                                  'limitations require the master encoder
                                  'for the second belt to be mounted such
                                  'that it rotates in the negative direction
                                  'while the belt moves forward.  Therefore,
                                  'the minus (-) sign is present on the axis
                                  '#3 argument.
```

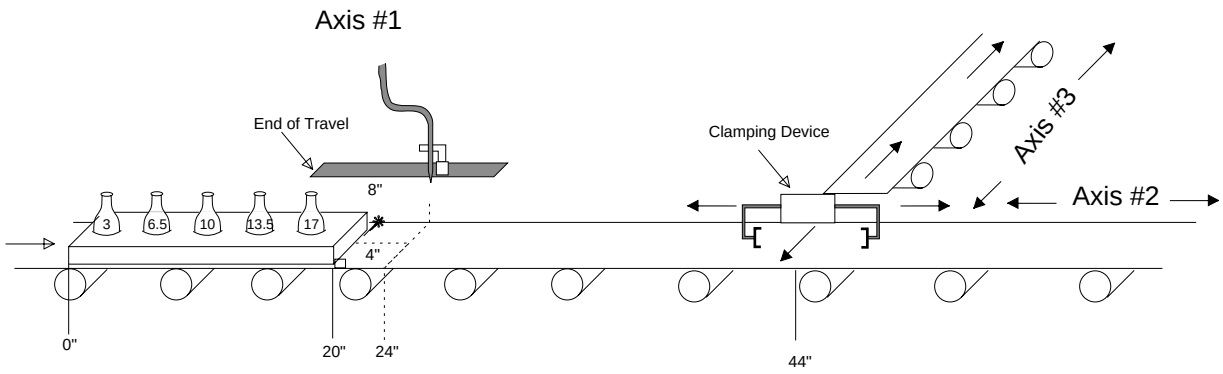
The next three statements initialize the remaining parameters required before the three moving positioning systems can be defined. These are the MPS ratio, offset, and master cycle length.

The nozzle's initial offset is 24 inches, allowing programming with respect to the far edge of the tray. The initial position of the clamp is 20 inches further away from the nozzle, allowing plenty of room for all the bottles to be filled before the tray moves into the area where it is to be clamped. To allow programming of the clamp positions to be done with respect to the far edge of

the tray, the moving positioning offset is set to 44 inches for axis #2. Notice that because the nozzle and clamp are in front of the tray as it is moving toward them on the first belt, their initial positions are positive. The second belt is moving away from the clamp, and because there is no particular destination the tray must be placed on, there is no need to define an initial offset position for axis #3.

Because the trays do not arrive with even spacing each tray represents a new cycle for all the axes. The cycle lengths for all the axes may be set to 0 to allow waiting for all necessary master cycle positions. With the **FOL MAS_CYC** value set to 0, the master cycle length is actually infinite. A new master cycle will be started with each tray that is sensed (**FOL NEWCYC**), so master positions will be referenced with respect to the leading edge of the tray. This is different than the slave axis reference point, remember its absolute positions will be referenced to the far edge of the tray.

```
FOLM RATIO 1:1 1:1 1:1 *      'Set MPS ratio to 1:1 on all slave axes
FOLM PDEF 24 44 0 *           'Initial positions when MPS is defined (in user
                              'defined units, inches)
FOL MAS_CYC 0 0 0 *           'Set master cycle lengths to 0 (infinite master
                              'cycle length)
```



At this point the machine is set up and ready to accept the trays of bottles to be filled. The next block of program statements performs those functions which were used to illustrate the concepts of the moving positioning system. For each tray, the first step is to wait for the sensor connected to trigger #1 to become active, signaling the leading edge of a tray. The moving positioning systems of axes #1 and #2 are defined at this time. Axis #1 enters the MPS, because it will begin moving to the bottles immediately. Axis #2 does not enter the MPS, because it remains at rest in the stationary positioning system until all the bottles have been filled. Axis #1 moves to tray positions 17, 13.5, 10, 6.5, and 3 inches. In between it toggles a programmable output for 1 second to fill each bottle and then waits for another 3.5 inches of master travel. It is still important to install end of travel limits, even though the slave waits for master positions to ensure that it never reaches it's negative end of travel. If the master is traveling more rapidly than expected, the slave may encounter its positive end of travel before all bottles are filled.

```
LABEL NEWTRAY                 'Subroutine label for each tray
FOL NEWCYC TRIG1 TRIG1 * *    'Begin new master cycle on trigger #1 for
                              'axes #1 and #2
FOLM DEF TRIG1 TRIG1 * *      'Define MPS on axes #1 and #2 upon trigger #1
                              'activation
FOL WAIT TRIG1 TRIG1 * *      'Pause here until the trigger goes active
FOLM ENABLE YES * * *         'Enter the moving positioning system
MOVE 17 * * *                 'Move axis #1 to position 17 inches on the
                              'tray
OUT POB 01XX                  'Output #1 activated
WAIT FOR 1 SECONDS            'Time delay for bottle filling
OUT POB 11XX                  'Release output #1
```

```
FOL WAIT 3.5 * * *          'Wait for 3.5 inches of master travel
MOVE 13.5 * * *             'Move to position 13.5 inches on the tray
OUT POB 01XX                'Output #1 activated
WAIT FOR 1 SECONDS          'Time delay for bottle filling
OUT POB 11XX                'Release output #1
FOL WAIT 7 * * *            'Wait for 3.5 more inches of master travel
MOVE 10 * * *               'Move to position 10 inches on the tray
OUT POB 01XX                'Output #1 activated
WAIT FOR 1 SECONDS          'Time delay for bottle filling
OUT POB 11XX                'Release output #1
FOL WAIT 10.5 * * *         'Wait for 3.5 more inches of master travel
MOVE 6.5 * * *              'Move to position 6.5 inches on the tray
OUT POB 01XX                'Output #1 activated
WAIT FOR 1 SECONDS          'Time delay for bottle filling
OUT POB 11XX                'Release output #1
FOL WAIT 14 * * *           'Wait for 3.5 more inches of master travel
MOVE 3 * * *                'Move to position 3 inch on the tray
OUT POB 01XX                'Output #1 activated
WAIT FOR 1 SECONDS          'Time delay for bottle filling
OUT POB 11XX                'Release output #1
```

At this point the nozzle has just finished filling the last bottle. The next task is to move the tray off the first moving belt onto the second moving belt. First, it is necessary to wait for the tray to come within range of the clamp. This is handled by axis #2 waiting for a master position of 44, the length of travel needed to clear the nozzle axis. Notice that because the moving positioning systems for axes #1 and #2 were both defined by trigger #1, their master cycle positions will always be the same.

Before the clamp is activated, both axes #2 and #3 must be positioned to the center of the tray. The home position of axis #3 has been placed so that it will be centered side to side. In order for axis #2 to become centered on the tray, it must enter the MPS, and then move to position 10. The same two statements take axis #1 back into the stationary positioning system, and return the nozzle to its starting position.

```
FOL WAIT * 44 * *           'Pause here until tray has moved into area for
                            'clamping
FOLM ENABLE NO YES * *      'Exit the MPS on axis #1 and enter on axis #2
MOVE 0 10 * *               'Move to stationary reference position 0 on
                            'axis #1 and MPS referenced position 10 inches
                            'on axis #2 (center of tray)
```

Setting programmable output #2 low causes the clamp to lower, close on the tray, and rise with the tray. A low signal on trigger #2 indicates that this is complete. The clamp must now place the tray on the second moving belt. Axis #3 must define and enter a moving positioning system based on the second belt, and move to position 0 within that reference frame. While this occurs, axis #2 enters the stationary reference frame, and moves to position 0, the center of the second belt. In order to be sure that the tray physically reaches the second belt, axis #3 must wait for a cycle position of the width of the tray, 6 inches in this case. Only then, the second programmable output bit can be set high, which causes the clamp to lower, release the tray, and rise. A high signal on trigger #2 indicates that this is complete. The last step is to prepare for the next tray. Axis #3 returns to the stationary positioning system and moves back to position 0.

```
OUT POB 10XX                'Set output #2 low
WAIT FOR TRIG2 = 0          'Wait for trigger #2 to go low - clamping
                            'complete
FOLM DEF * * IMMED *        'Define MPS on axis #3
FOL NEWCYC * * IMMED *      'Begin new master cycle immediately
FOLM ENABLE * NO YES *      'Exit the MPS on axis #2 and enter on axis #3
```

```
MOVE * 0 0 *                'Move to stationary reference position 0 on
                            'axis #2 and MPS referenced position 0 inches
                            'on axis #3
FOL WAIT * * 6 *            'Wait for 6 inches of master travel (width of
                            'tray)
OUT POB 11XX                'Set output #2 high to release clamp
WAIT FOR TRIG2 = 1          'Wait for trigger #2 to go high - clamp
                            'released
FOLM ENABLE * * NO *        'Exit the MPS on axis #3
MOVE * * 0 *                'Move to stationary reference position 0
GOTO NEWTRAY                'Goto label NEWTRAY and repeat the process
```

Special Features of the Moving Positioning System

Notice that in the bottle filling example, the first positioning command given on axis #1 after the tray is sensed is given while the slave is already in motion with respect to the moving reference frame. The 4000 allows normal time-based preset moves to be made in a moving positioning system while the slave is already moving, even if the slave's current direction is away from the new commanded position. This means that the slave does not need to waste time stopping before moving to the first commanded position.

A second special feature is that of super-imposed contouring on a moving target. Imagine an application where glue must be dispensed in a pattern onto a web as it moves through a machine. The obvious three axis solution would be to mount a two axis X-Y stage used for contouring onto a one axis table which would follow the web. A more cost effective and reliable solution would be to use one axis of the X-Y stage to follow the web. Now the mechanics are reduced and only two motors are required. The programming from the user's point of view would be identical to the contouring of the three axis approach, but the X axis would have entered the moving positioning system and started tracking the master before the contour is drawn.

In this contouring case, as with normal positioning, the path velocity of the contour with respect to the moving web is not dependent on the speed of the master. This allows uniform application of the glue along the pattern, regardless of master speed.

Another possibility to discuss is that of differential following. Imagine an application in which trackball positioning must be performed above a web as it moves through a machine. The same three axis versus two axis solutions exist as with the contouring example described above. Once again, the programming for the two solutions is the same, except that one axis of the trackball would be the same as the web following axis. This is a case where ratio following is super-imposed on a moving positioning system.

The obvious benefit to the use of the moving positioning system is the ability to program positioning on a moving target in exactly the same way it would be programmed in the stationary reference frame. In both cases, the programmer need not be concerned with velocity versus position along the move profile, only to command end positions, contours, following ratios, I/O, or whatever the application requires.

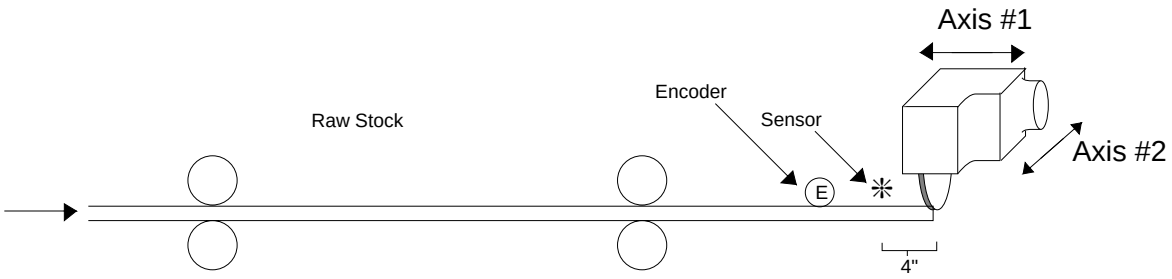
A second application of the moving positioning system is described below.

Continuous Cut to Length

Let's take another look at the automobile stock cutting application discussed earlier in this chapter. Below is another programming solution for that example using the moving positioning system.

Once again, this application calls for automobile trim to be cut to a pre-defined length. The saw is controlled by axes #1 and #2 on the 4000. It must be moving with the material while the cut is being made (axis #1), and also moving perpendicular to the trim (axis #2) to actually make the cut. The trim comes in long stock which moves continuously under the cutting area.

The leading edge of the trim stock is detected with a sensor connected to trigger #1 which is located 4 inches from the home position of the saw. Axis #1 will be following the trim based on an encoder mounted on the trim via a friction wheel. The encoder is a 1000 line encoder and the wheel is geared to give 2 revolutions per inch of trim. This results in 8,000 post quadrature steps per inch of trim. Axis #1 has a resolution of 25000 steps per rev, and is connected to a 2 pitch leadscrew 24" in length. Axis #2 is similar in mechanics but its length is 10". The travel on Axis #1 will be controlled by the speed at which axis #2 makes its cut. Limit switches are in place for safety.



Below, the initial cut length is 36", established with each cut by the current value of variable Q1. Minor modifications to this program could allow Q1 to be read from **DATA** statements or modified in other ways. The cut cycle will be a continuous loop, but the first cut will be made 0.2" from the end of the stock to ensure an even first edge. Assume that the home position of both axes is at position 0". The moving position system offset will be set to the distance from axis #1's home position to the desired cut position, 4.2". This means that the desired cut position will be position 0 in the MPS. This example differs slightly from the first *Cut to Length* example in that the initial master is set to be -4.2".The program below differs from the first look at this example in that as soon as axis #1 enters the MPS and moves to the moving position 0, it starts tracking the speed of the master. As soon as this happens, the cut with axis #2 takes place. Notice that as soon as axis #1 moves to the moving cut position, 0", the position is then defined as the cut length. This enables the loop to execute continuously.

```
UNIT POS 50000 50000 * *          'Set axes #1 and #2 scale factors for
                                  'programming in inches
UNIT VEL * 50000 * *              'Axis #2 velocity scale factor for
                                  'inches/sec
UNIT ACCEL * 50000 * *            'Axis #2 accel scale factor for
                                  'inches/sec/sec
UNIT MASTER 8000 * * *            'Master scale factor for programming in
                                  'inches
ACCEL * 20 * *                    'Accel for axis #2
VEL * 5 * *                       'Velocity on axis #2 to 5 inches/sec
MODE M_ABS M_ABS * *              'Absolute positioning mode for non-
                                  'following moves
MATH Q1 = 36                      'Desired cut length is 36"
MATH Q5 = Q1                      'Set Q5 cut length
MATH Q2 = 4                       'Sensor is 4" from home position of axis #1
MATH Q2 = Q2 + 0.2                '1st cut to be 0.2" from end of stock
MATH Q3 = -Q2                     'Q3 = offset for 1st cut
FOL MASTER ENC1 * * *             'Encoder input #1 is the master for axis #1
FOL CYC_OFF Q3 * * *              'Initial master cycle position
FOLM RATIO 1:1 * * *              'MPS tracking ratio is set to 1:1
FOLM PDEF Q2 * * *                'Slave position when the MPS is defined
                                  '(upon trigger #1) will be distance to
                                  'first cut location.
FOLM DEF TRIG1 * * *              'Define moving positioning system upon trig
                                  '#1
FOL NEWCYC TRIG1 * * *            'Also define a new master cycle on that
                                  'trigger
```

```
FOL WAIT TRIG1 * * *          'Suspend program until edge of product is
                              'sensed
OUT BIT7 = 1                  'Lower the saw blade into position
FOL WAIT Ø * * *              'Wait for desired master travel before
                              'executing first move
LABEL NEW_CUT                 'Subroutine label for continuous operation
FOLM ENABLE YES * * *         'Enter the MPS
MOVE 0 * * *                  'Move to position 0" within the MPS (the
                              'cut position
PDEF Q1 * * *                 'Set current position to desired cut length
MOVE * 10 * *                 'Move axis #2 to make cut
FOLM ENABLE NO * * *          'Exit the MPS
OUT BIT7 = 0                  'Raise the saw blade
MOVE 0 0 * *                  'Move both axes back to home positions
OUT BIT7 = 1                  'Move saw blade into position for next cut
FOL WAIT Q5 * * *             'Wait for end of cycle
MATH Q5 = Q1                  'Set Q5 = Q1 (new cut length)
FOL MAS_CYC Q5 * * *          'New master cycle length is cut length
GOTO NEW_CUT                  'Repeat the cut cycle
```

Technical Considerations for Following

In the Technical Overview section at the start of this chapter, the algorithm for Model 4000 Following was briefly discussed. Here we will address some of the more technical aspects of Following. Topics covered include Velocity Feed Forward, Velocity Smoothing, Dynamic Position Maintenance, Preset vs. Continuous Following Moves, and Master and Slave Distance Calculations. Keep in mind that in all cases, the slave position is calculated from a sampled master position.

Velocity Feed Forward

Velocity feed forward is simply a technique used to compensate for the fact a slave position setpoint can not be calculated and implemented infinitely fast. As noted in previous paragraphs, the 4000 measures master position every two milliseconds, and calculates a corresponding slave setpoint. This calculation and achieving the subsequent slave setpoint position require 4 milliseconds. If velocity feed forward is not turned on, this results in a slave position lag. In other words, by the time the slave reaches the position which corresponds to the sampled master position, 4 milliseconds have gone by, and the master may be at a new position. Measured in time, the lag is 4 milliseconds. Measured in position, the lag is 4 msec * current slave velocity. For example, suppose our slave is traveling at a speed of 25000 counts per second. Without velocity feed forward enabled, the slave will lag the master by 100 counts (25000 counts/sec * 4msec = 100 counts).

By measuring the change in master position over a number of sample periods, the master's velocity is calculated every two milliseconds. The present master velocity and position are used to predict future master position. If velocity feed forward is enabled, the predicted future master position is used to determine the commanded slave position setpoint. In this case the slave has no velocity dependent phase delay. The slave's velocity for a given sample will always be the velocity required to move from its current position to the next calculated setpoint. Velocity feed forward is activated by default in the Following algorithm, but can be turned off as desired with the **FOL VELFF** statement.

If the master velocity is fairly smooth and constant, the measurement of its recent velocity will be very accurate, and a good way of predicting future position. But the master motion may be rough, changing, or measured over a very short sample period (see Velocity Smoothing). In this case, the predicted master position and the corresponding slave setpoint will have some error, which may vary in sign and magnitude from one sample to the next. This

random variation in slave setpoint error results in rough motion. The problem is particularly pronounced if there is vibration on the master.

It may be desirable to deactivate velocity feed forward when maximum slave smoothness is important and minor phase delays can be accommodated.

Velocity Smoothing

In the default Following algorithm, invoked when the **FOL MASTER** statement is executed, the master's velocity is calculated over 2 two millisecond sample periods. This default algorithm will work well for most Following applications. If master pulses are received at a very slow rate or if the master is actually vibrating, it may be necessary to change the velocity smoothing factor. If coarse velocity measurement is used in velocity feed forward due to fluctuations in master velocity speed, there will be a corresponding roughness in the motion of the slave.

Increasing the velocity smoothing value from the default value of 1 up to its maximum value of 4 has two effects. One is to increase the filtering done by the 4000 on each individual master position measurement, and the other is to lengthen the velocity averaging sample period. The Model 4000 filters master position measurements by averaging the actual position with the master position which would be expected based on previous velocity measurements. Increasing the smoothing number increases the weight of the expected position in these calculations. This serves as a software damping for vibration on the master.

Lengthening the sample period by increasing the velocity smoothing number also increases the velocity measurement resolution (i.e. how many pulses are coming in per sample period). The 4000 allows sample periods of 4, 8, 16, or 32 milliseconds with the **FOL SMOOTH** statement values of 1, 2, 3, and 4 respectively. (These correspond to velocity measurement resolutions of 250Hz, 125Hz, 62Hz, and 31Hz.) This affects accuracy of the velocity read using the **IN Qn = FOL AXISn MAS_V** statement. For example, if the actual master velocity if 500Hz, and a value of **FOL SMOOTH 2** is chosen, the reported velocity will be ±25% accurate.

Lengthening the sample period gives smoother motion on the slave, but will also result in a slower response to master velocity changes. For those applications in which the master undergoes rapid velocity changes, it may not be desirable to have the velocity smoothing factor set very high. Here, a high velocity smoothing factor will cause sluggish velocity response on the slave.

If acceleration and deceleration of the master result in significant velocity changes within the sample period, temporary tracking error will occur if the 4000 is smoothing the velocity measurement. However, this does not mean that the overall position relationship between master and slave is lost. Because velocity measurement includes data from the previous sample period, slave position will temporarily *lag* during master acceleration, and *lead* during master deceleration. These temporary errors are corrected during constant master velocities, and minimized by using a low velocity smoothing factor.

Dynamic Position Maintenance

Even while following a master, a slave axis can be in encoder mode with stall detection, position maintenance, and/or deadband wait enabled. In this mode of operation, a difference between actual slave position and the desired slave position may arise just as it may with normal encoder mode moves. Systems with an encoder mounted on a load where mechanical backlash or product stretching is present will be most prone to these desired vs. actual position differences.

An axis becomes a slave by specifying a master with the **FOL MASTER** statement. When an axis in encoder step positioning becomes a slave axis, it automatically begins the equivalent of position maintenance. This remains

true even if the **ENABLE POSM NO** statement is given, and even while the axis is in motion. There is a very important reason for the continuous dynamic position maintenance. In regular time based moves, the axis has no defined position relationship with anything while it is moving but it does have a position goal while at rest. Therefore position maintenance is only meaningful when the axis is not moving. When a slave axis is following a master, there is always a defined position goal, calculated from the position of the master. Therefore, position maintenance occurs even while the axis is moving. The position error each sample period will usually be very small, so only a small correction velocity is added to that required for the slave to follow the master at the commanded ratio. The 4000 limits the maximum correction velocity to that specified with the **POSM MAXVEL** statement.

Setpoint Lead, Direction Change Set-Up, and Trigger Debounce

Most of the concepts discussed above will apply to all applications and all equipment used with those applications. There are, however, some differences in drive electronics and sensors which need to be addressed in following applications. One of the these is a velocity dependent lag in the controlled position in some drives. Drives which must convert step and direction input into the corresponding torque control via software may have some inherent delay in their response to these input. Among such drives are the Dynaserv step and direction versions, which have up to 6 milliseconds of lag.

In normal positioning applications, a delayed response to step and direction input will not cause a problem, because the final position is the only position of interest. In following, however, every position along the profile is important, because it must correspond to a master position. A velocity dependent lag causes a synchronization error. When the drive is a servo drive, it is not feasible to use the the dynamic position maintenance which is associated with encoder mode, because the drive is attempting its own position control. The setpoint lead feature allows the Model 4000 to dynamically advance the slave setpoint beyond what would be normally resulting from the profile. It is advanced by the product of the instantaneous velocity and the lead value specified with the **FOL LEAD** statement. This helps to compensate for lag in the drive, but at the expense of stabilty in commanded position. The commanded advance is a function of the velocity, but also affects the velocity. This can cause the profile to be skewed during slave acceleration and deceleration. It is best to avoid this feature when possible (i.e., use the default value of zero). If it is necessary to compensate for drive lag, the proper value will need to be determined empirically.

Another attribute of some drives is the requirement for a direction input change setup time. Most Compumotor steppers have this requirement. It simply means that some minimum time must be allowed for the drive to respond to a change on the direction input before any steps are given in the new direction. This mimimum time is generally much less than the 2 milliseconds allowed by the default condition of the Model 4000. In normal positioning applications, motion always stops before a move is commanded in the opposite direction, so this requirement is of no consequence. In following however, the master may change direction, resulting in a direction change on the slave without a new move command. When the set-up time is enabled, the Model 4000 temporarily saves the steps which would have been sent with the direction change. These steps are sent during the next 2 millesecond update. Some drives, such as the Dynaserv and Z Drive, do not have a direction change set-up requirement. In order to facilitate smooth following on these drives, the **FOL DIRSET** statement may be used to disable the default 2 millisecond direction change set-up.

Finally, the variety of sensors and their electronics demands that the debounce time for the trigger inputs be programmable. Debounce time refers to the time that trigger input change will be ignored after receipt of a rising edge on that input. The **DEFINE TRIGDB** statement allows the debounce times of the trigger inputs to be individually programmed. This allows flexibility in the choice of sensors, switches, and inputs. For example, a clean electronic signal could be used at a fairly high rate for Master/Slave Synchronization Sync Marks, while a mechanical switch with significant bounce could be used on some other input.

Factors Affecting Following Accuracy

Many references have been made throughout this chapter to the additional accuracy requirements of following applications beyond those of standard positioning. The slave must maintain positioning accuracy while in motion, not just at the end of moves, because it is trying to stay synchronized with the master. Assuming parameters such as master and slave scaling and ratios have been specified correctly, the overall positioning accuracy for an application depends on several factors. Just as with a mechanical arrangement, the accuracy errors can build up with every link from the beginning to the end. The overall worst case accuracy error will be the sum of all the sources of error listed below. The errors fall into two broad catagories, namely, master measurement errors and slave errors. These both ultimately affect slave accuracy, because the commanded slave position is based on the measured master position.

It is important to understand how master measurement errors result in slave position errors. In many applications, master and slave units will be the same, e.g., inches, millimeters, degrees. These applications will require linear speeds or surface speeds to be matched, i.e., a 1:1 ratio. For example, in the rotary knife application discussed in the *Cam Profiling* section, there were 500 master steps per inch of material, so an error in master measurement of one encoder step would result in .002 inches of slave position error. If the master and slave units are not the same, or the ratio is not 1:1, the master error times the ratio of the application gives the slave error. For example, suppose one revolution of a wheel gives 4000 master counts, and results in 10 inches of travel on the slave. The ratio is then 10 inches:one revolution. The slave error is which results from one step of master measurement error is (1/4000) * 10 inches:one revolution = .0025 inches.

① Resolution of the master. The best case master measurement precision is the inverse of the number of master steps per user's master unit. Even if all other sources of error are eliminated, slave accuracy will only be that which corresponds to 1 step of the master.

② The 4000's sampling accuracy. The 4000 has a nominal sampling rate of 2 milliseconds, but the precision of this rate may vary by as much as 100 microseconds from one sample to the next. This affect is may reduced to something less than 50 microseconds by using higher smoothing values . This means that measurement of master position may be off by as much as (50 to 100 microseconds * master speed). This may appear to be a significant value at high master speeds, but it should be noted that this error changes in value (and usually sign) every 2 milliseconds. It is effectively like a 500 Hz noise to the overall system. If the mechanical frequency response of the motor and load is much less than 500 Hz, the load can't respond to this error.

③ Velocity Feed Forward. This feature may be turned on or off, but each state contributes a different error. If velocity feed forward is turned off, the slave setpoint command is based on a master position that is 4 milliseconds old. This means that master measurement error due to velocity feed forward being off will be (4 milliseconds* master speed). If velocity feed forward is turned on (the default case), its accuracy is also affected by sampling accuracy, master speed and velocity smoothing. The error due to velocity feed forward being on is about

twice that due to sampling accuracy, i.e., (100 to 200 microseconds * master speed). As with the error due to sampling accuracy, error due to velocity feed forward being on is a 500 Hz error, which is not noticed by large loads.

④ Master Speed Variation, Velocity Feed Forward, and Velocity Smoothing. Although increasing velocity smoothing helps reduce the error due to sampling accuracy, it increases the error due to variations in master speed when velocity feed forward is on. Most applications keep a constant master speed, or change very slowly, so this effect is minimal. But if the master is changing rapidly, there is a significant master speed measurement error. This *master speed error* is about one half the master speed change over the velocity smoothing sample period. Please refer to the section titled *Velocity Smoothing* for a discussion of sample periods. This corresponding master measurement error will be (4 milliseconds * *master speed error*). This effect will always be smaller than that due to velocity feed forward being turned off.

⑤ Resolution of the slave. The best case slave precision is the inverse of the number of slave steps per user's position unit. Even if all other sources of error are eliminated, slave accuracy will only be that which corresponds to 1 step of the slave. This must be at least as great as the required precision.

⑥ Dynamic Position Maintenance. Even when the slave is in motor step mode, there may be one slave step of error inherent to the algorithm. When the slave is in encoder step mode, the user specified position maintenance gain is used to correct position. This gain is expressed as motor steps per second per encoder step error, and has a maximum value of 250. If a value lower than this is used, the position error in encoder steps due to low gain is given by: error = (250/gain) * (encoder resolution/motor resolution).

⑦ Accuracy of the slave motor and drive. The precision also depends on how accurately the drive follows its commanded position while moving. Even if master measurement were perfect, if the drive accuracy is poor, the precision will be poor. In the case of stepper drives, this amounts to the specifed motor/drive accuracy. In the case of servo drives, the better the drive is tuned for smoothness and zero following error, the better the precision of the positioning. Often, this really only matters for a specific portion of the profile, so the drive should be tuned for zero following error at that portion.

⑧ Accuracy of load mechanics. This is fairly self explanatory. The accuracy (not repeatability) of the load mechanics must be added to the overall build up of accuracy error. This includes backlash for applications which involve motion in both directions.

⑨ Repeatability of the trigger inputs and sensors. Some applications may use the trigger inputs for functions like registration moves, movewaits, new cycles, master/slave synchronization, or moving positioning system definition. For these applications, the repeatability of the trigger inputs and sensors add to the overall position error. In the 4000, the trigger inputs have 100 microsecond repeatability, and the sensor repeatability (SR) should be determined too. Velocity * time = distance, so the error due to repeatability is (SR +.0001 seconds) * speed = error. If the sensor repeatability is given in terms of distance, that value can be added directly.

Preset vs. Continuous Following Moves

When a slave performs a preset move in Following mode, the commanded position is either incremental or absolute in nature, but it does have a commanded endpoint. The direction traveled by the slave will be determined by the commanded endpoint position, and the direction the master is counting. Let's illustrate this with an example. Assume all necessary setup

statements have been previously issued for our slave (axis #1) and master so that distances specified are in revolutions:

```
FOL RATIO .75:1              'Following ratio of .75 rev on the slave to 1
                             'rev on the master
FOL ENABLE YES * * *         'Enable following on axis #1
FOL MDIST 10 * * *           'Preset move to take place over 10 master
                             'revolutions
MODE M_ABS * * *             'Slave in absolute mode
PDEF 0 * * *                 'Set current position to 0
MOVE 5 * * *                 'Move to position 5 revolutions
```

If the master is stationary when the **MOVE** statement is executed, the slave will remain stationary also. If the master begins to move and master pulses are *positive in direction*, the slave will begin the preset move in the positive direction. If the master pulses stop arriving before 10 master revolutions have been traveled, the slave will also stop moving, but that **MOVE** statement will not be completed. If the master then starts to count in the negative direction, the slave will follow in the negative direction, but only as far as it's starting position. If the master continues to count negative, the slave will remain stationary. The **MOVE** statement will not actually be completed until the master has traveled at least 10 revolutions in the positive direction from where it was at the time the **MOVE** command was executed. If the master oscillates back and forth between it's position at the start of the **MOVE** command to just under 10 revolutions, the slave will oscillate back and forth as well.

The master must be counting in the positive direction for any preset **MOVE**s commanded on the slave to be completed. If mechanics of the system dictate that the count on the source of the master pulses is negative, a minus (-) sign should be entered in the **FOL MASTER** statement so that the 4000 sees the master counts as positive.

Continuous slave moves react much differently to master pulse direction. Whereas a preset move will only start the profile if the master is moving in the positive direction, a continuous move will begin the ramp to its new ratio following the master in either direction. As long as the master is counting in the positive direction, the direction towards which the slave starts in a continuous move is determined by the argument of the **MOVE** statement. The slave direction is positive for **MOVE SLEWCW** and negative for **MOVE SLEWCCW**.

If the master is counting in the negative direction when slave begins a continuous move, the direction towards which the slave moves is opposite to that commanded with the **MOVE** statement. The slave direction is positive for **MOVE SLEWCCW** and negative for **MOVE SLEWCW**.

If the master changes direction during a continuous slave move, the slave will also reverse direction. As with standard continuous moves, the **MOVE SLEW** will continue until terminated by **MOVE STOP**, **MOVE KILL**, end of travel limits, stall condition, or command to go to zero velocity. As with preset moves, the sign on the **FOL MASTER** statement determines the direction of master pulses.

In both types of moves, acceleration to commanded ratio may take place with respect to master steps or a time-based acceleration. If **FOL MDIST** has been executed more recently than **ACCEL**, the ratio change will take place with respect to master steps. In this case, no change in ratio will result when a **MOVE** is commanded until some master pulses have been received. If a time-based **ACCEL** is used, the slave will accelerate until it reaches the velocity implied by the commanded ratio and the current master velocity. If the current master velocity is zero, the slave acceleration takes no time.

Master and Slave Distance Calculations

As described earlier in the chapter, a slave's acceleration to a commanded ratio can be defined in one of two ways. A standard time-based acceleration ramp may be defined with the **ACCEL** statement or the **FOL MDIST** statement can be used to determine the master distance over which the acceleration (mode continuous moves) or entire move (preset moves) takes place.

For the latter case, the formulas below show the relationship between master move distances and the corresponding slave move distances. These formulas may be re-arranged to solve for the desired parameters:

Mode Continuous Moves

$$D = \frac{MD * (R2 + R1)}{2}$$

where:

MD = Master Distance (**FOL MDIST**)

R2 = New Ratio

R1 = Current Ratio

D = Slave Distance Traveled During Ramp

Trapezoidal Preset Moves

$$Dmax = (MD * R_{max})$$

$$D1 = \frac{(Dmax - D)}{2}$$

$$MD1 = \frac{2 * D1}{R_{max}}$$

$$D2 = D - (2 * D1)$$

$$MD2 = \frac{D2}{R_{max}}$$

$$MD = (2 * MD1) + MD2$$

where:

MD = Master Distance (**FOL MDIST**)

MD1 = Master Distance During Accel and Decel Ramps

MD2 = Master Distance During Constant Ratio

D = Total Slave Preset Distance

D1 = Slave Travel During Accel and Decel Ramps

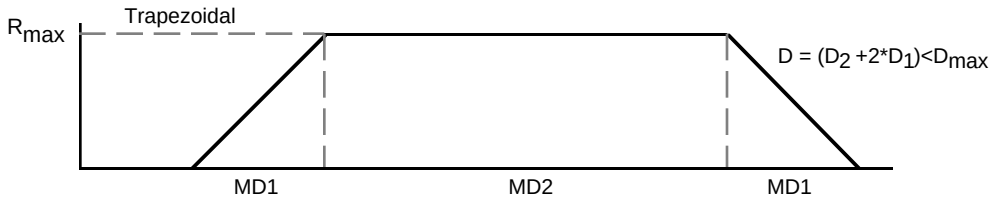
D2 = Slave Travel During Constant Ratio

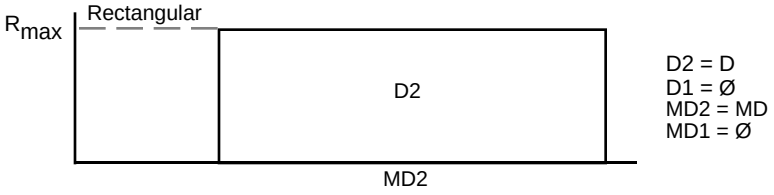
D_{max} = Maximum Slave Distance Possible

R_{max} = Maximum Ratio

☛ D_{max} results in rectangular move profile.

If (Dmax - D) > D, then the move will be triangular.





Triangular Preset Moves

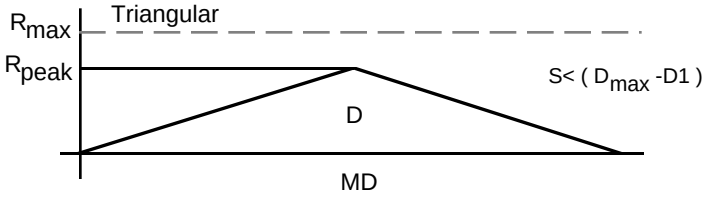
$$D = \frac{R_{peak} * MD}{2}$$

where:

MD = Master Distance (**FOL MDIST**)

R_{peak} = Peak Ratio Reached During Move

D = Total Slave Steps



These relationships may be used to assist in the design of following mode moves in which both the position and duration of constant ratio are important. In such calculations, it is very helpful to use **UNIT MASTER** and **UNIT POS** values which allow the master and slave distances to be expressed in the same units, e.g., inches or millimeters. In this case, many applications will be designed to reach a final ratio of 1:1, and the distances in these figures can be easily calculated. For a trapezoidal preset move with a maximum ratio of 1:1, the master and slave distances during the constant ratio portion will be the same. The slave travel during acceleration will be just half of the corresponding master travel, and will also occur during deceleration. In the example below, the desired travel during constant ratio is already contained in Q1, and may have been read from thumbwheels or a **DATA** statement. The corresponding **MOVE** distance and **FOL MDIST** are as shown:

```
MATH Q2 = 2          'desired slave travel during accel and decel combined
MATH Q3 = 2 * Q2     'required master travel during these ramps
MATH Q4 = Q1 + Q2    'move distance is constant ratio portion plus ramps
MATH Q5 = Q1 + Q3    'master travel for entire slave move
FOL MDIST Q5 * * *   'establish calculated master travel
MOVE Q4 * * *        'make desired slave move
```

Similar calculations may be done for a series of continuous move ramps to ratios, separated by **FOL MOVEWT** for master cycle positions. These ramps may be repeated in a loop to create a continuous cyclical slave profile. The *Web Processing* example earlier in this chapter uses this technique.

Some potential for roundoff error exists if the scaling of a move distance or master distance by **UNIT POS** and **UNIT MASTER** respectively do not result in an integer number of steps. Some additional care must be taken in the segment by segment construction of profile using ramps to continuous ratio. The 4000 maintains a slave position setpoint which is calculated from the commanded constant ratios, the ramps to the new ratios, and the master travel over which these take place. At the end of each ramp or constant ratio portion, this setpoint is calculated to the nearest integer slave step. If the ratios and master travel result in a non-integer slave travel for a segment, the fractional part of that segment's calculated travel will be lost. In a cyclical application, repeated truncations could build up to significant error. This may be avoided through the use of sync marks, as in the *Web Processing* example, or through careful attention to design of the profile.

Using Other Features with Following

The 4000 has many features which may be used in the same application as its following features are. In some cases, having configured an axis as a slave with the **FOL MASTER** statement will affect the operation of other features. These are described in the paragraphs below.

Setups used by FOL MASTER

The **FOL MASTER** statements uses the motor resolution (**ENCO MRES**), the velocity range (**VEL RANGE**), and the choice of motor step positioning or encoder step positioning **MODE** data to configure an axis as a slave. If encoder step positioning is chosen, the encoder resolution (**ENCO ERES**) data is also used. In order for this data to be used correctly, these statements must be given before **FOL MASTER**. *These statements will simply be ignored after* ***FOL MASTER*** *is given*. The **MODE** statement *may* be used to change from incremental to absolute positioning and back, but a change to encoder or motor step positioning will be ignored.

MOVE and MOVI

If a slave axis is in following mode (**FOL ENABLE YES**) moves will ramp to a ratio (**FOL RATIO**). If it is not in following mode, moves will ramp to a velocity (**VEL**). Switching in and out of following mode does not change the value for final ratio or final velocity goals, but simply changes which parameter is used as the goal.

MOVE STOP and MOVE KILL

These cause the slave to do a non-following decel, even if the slave is in following mode. If the slave is currently in the MPS, a **STOP** will decelerate the slave to a position within the MPS, so the slave may still be moving with respect to a stationary reference. A **KILL** will instantly stop motion with respect to a stationary reference, and automatically exit MPS. Both will clear a **FOL MOVEWT** pending condition, and both will clear (i.e., zero) any accumulated following error.

MOVEs while moving

In some cases, it is possible for a slave to perform a preset move, i.e., move to a position, while it is already moving. The need arises whenever a preset move must be made immediately after a switch is made to or from MPS. The move may always be done if the slave is not in following mode. If the target position is too close, the slave simply overshoots and comes back. If the slave *is* in following mode, the move may only be done if the profile is determined by **ACCEL**. In this case, if the target position is too close, the slave decelerates abruptly until it can reach the target with the commanded deceleration. If the profile is determined by **FOL MDIST**, the **MOVE** command will be ignored.

Not while in MPS

MOVE HOME moves and **MOVE SEG** moves will be ignored while the slave is in the Moving Positioning System. These moves are not normally allowed while the axis is already *moving*. In these cases, that includes being at rest in MPS.

Jog and Joystick

Entering jog mode and joystick are normally not allowed while the axis is in motion. In these cases, that does *not* includes being at rest in MPS. If an axis is at rest in MPS when either of these modes is entered, the axis automatically and immediately exits MPS, and implements the jog or joystick velocity with respect to a stationary reference. It does not return to MPS when jog or joystick mode is exited.

Registration Moves

Registration inputs may be enabled while an axis is a slave, and registration moves may interrupt either a following mode move or a time based move. They may also interrupt an axis which is either at rest or moving in MPS. The registration move itself, however, is always a time based move, and implements the registration velocity with respect to a stationary reference. It does not return to MPS when move is completed. Any trigger input may be used for a registration input or it may be used for any following feature which uses triggers, but not at the same time. A trigger input used for any following feature will disable that trigger as a registration move input and vice versa.

Entering and Exiting Following Mode while Moving

The **FOL ENABLE NO** command may be given while a slave is moving at constant ratio. In this case, the current velocity becomes the constant velocity, and the slave may accelerate or decelerate to other velocities. The **FOL ENABLE YES** command may not be given while the slave is moving out of following mode. Attempting to do so will result in an execution error.

Done, Stop and Resume

A program may be stopped and resumed, even if one or more axes is configured as a slave. Those axes do not lose track of the master input, even though motion is stopped. As usual, if a program finishes normally, is aborted, or if **MENU RECALL** is pressed, the program may not be resumed. In these cases, the equivalent of **FOL MASTER NO** is automatically executed, and no axis is configured as slave. As a result, the **POWER_UP** program may not configure a slave for a program which is to be run separately afterwards. If a program is resumed, partially completed following moves will be completed. If these moves were done using **FOL MDIST** however, the remainder of the move is completed over the entire original master distance.

Troubleshooting a Following Application

Following applications are often more complex than others, because motion of the slaves is programmed as a function of the motion of master. This requires the motion of the master to be well characterized, and accurately specified in the program, It often requires an unfamiliar way of thinking about the motion of the desired slave. The table below offers some possible reasons for troubles which may be encountered in achieving the desired slave motion.

Trouble Symptom	**Possible Causes**
Slaves do not follow master	•Improper **FOL MASTER** •Poor connection if master is encoder •Master running backward •No encoder power
Slave motion is rough	•**FOL SMOOTH** too low •Unnecessary **FOL VELFF** amplifies master roughness.
Ratio seems wrong	•**FOL RATIO** slave: master numbers reversed. •**UNIT POS** or **UNIT MASTER** wrong. •**ENCO MRES** or **ENCO ERES** wrong for encoder step slaves. •Following limited by **FOL MAXVEL** or **FOL MAXACC**
Slave *loses* previously specified following commands	•**FOL MASTER** has been re-executed, re-establishing defaults
Slave profile wrong, or unrepeatable	•Accel vs. **FOL MDIST** correct? •**FOL WAIT** used where **FOL MOVEWT** should be. •Too little master travel between **FOL MOVEWT** and **MOVE**, wait is missed.
Master/slave alignment drifts over many cycles	•Roundoff error due to fractional steps resulting from **UNIT POS** or **UNIT MASTER** and users parameters. •Ratios and master distances specified result in fractional slave steps covered during ramps, constant ratio.
Slaves ignore **ACCEL** statement	•**FOL MDIST** given more recently. •Slaves reached ratio at low master speed, then followed master acceleration. •Slaves inhibited by **FOL MAXACC**.
Excess following error constantly detected.	•**FOL PTOL** too small. •**FOL PTOL** statement not re-executed before new **ON FOL_ERR**.
Slave lags following position	•Inhibited by **FOL MAXVEL** •**FOL MAXACC** *clips* acceleration peaks resulting from attempt to follow rough master.

Slave dithers, or oscillates about desired position.

- **POSM GAIN** too high for encoder step slave.
- **FOL MAXACC** too low for **POSM MAXVEL**.

The Model 4000 does as much error checking as possible during the execution of a program. If an illegal parameter is discovered, the Model 4000 responds with an execution error message, and the program is aborted.

The table below lists all the error messages that relate to following, and indicates the statement and cause which may generate them.

Error Messages	Description
Big RAM not installed in U14, U15	This occurs if any **FOL** or **FOLM** statement or any **INQ** request for a following parameter is executed without the RAM required for the following option installed.
Invalid **FOL** master specified	This indicates that an illegal master was specified in **FOL MASTER**. A slave may never use its own motor step count as its master. A slave in encoder step mode or with stall detect enabled may not use its own encoder step count as master.
This **FOL** not valid while moving	This indicates the statement is not allowed while the slave is moving. *Moving* means moving with respect to the current positioning system. A slave may be stationary with respect to a stationary reference, yet be *moving* in the moving positioning system. **FOL MASTER** **FOL ENABLE**
FOL MASTER not executed	This indicates that no **FOL MASTER** for the axis is currently specified. It will occur if any **FOL** or **FOLM** statement defining or enabling parameters or any **INQ** request for a following parameter is executed and no **FOL MASTER** statement was executed, or **FOL MASTER NO** was executed.
FOLM DEF not completed	This indicates that the statement is not allowed if no moving positioning system is defined. It could occur if **FOLM DEF** was never executed, or if the trigger which defines the moving positioning system has not occurred. **FOLM ENABLE YES**
FOL parameter too large	This indicates that the numeric parameter supplied with the statement is too large. **FOL MDIST**— Error if: master steps > 999999999 **FOL MAS_CYC**—Error if: master steps>999999999 **FOL WIN_P**—Error if: master steps>999999999 **FOL WIN_W**—Error if: master steps>999999999 **FOL CYC_OFF**—Error if: master steps>999999999 or <-999999999 **FOL PDEF**—Error if: slave steps>999999999 or <-999999999 **FOL SHIFT**—Error if: slave steps>999999999 or <-999999999 **FOL SYNC_OFF**—Error if: slave steps>999999999 or <-999999999 **FOL PTOL**—Error if: slave steps>999999999 **FOL MOVEWT**—Error if: master steps >999999999 or <-999999999 **FOL WAIT**—Error if: master steps >999999999 or <-999999999 **FOL RATIO**—Error if specified or calculated ratio >127 or <-127 **FOLM RATIO**—Error if specified or calculated ratio >127 or <-127 **FOL WAIT**—Error if: master steps >999999999 or <-999999999 **FOL MSYNC**—Error if: master steps>999999999 **FOL SSYNC**—Error if: master steps>999999999
FOL parameter not valid	This indicates that the parameter supplied with the statement is not valid. **FOL MDIST** —Error if: master steps are negative **FOL MAS_CYC** —Error if: master steps are negative **FOL RATIO** —Error if: ratio denominator is negative **FOLM RATIO**—Error if: ratio denominator is negative **FOL SMOOTH** —Error if: smooth number is not 1-4 **FOL WIN_P**—Error if: master steps are negative **FOL WIN_W** —Error if: master steps are negative **FOL PTOL** —Error if: slave steps are negative **FOL MSYNC** —Error if: slave steps are negative **FOL SSYNC** —Error if: slave steps are negative
Master cycle definition pending	This indicates a master cycle definition is pending a trigger, the master cycle position is unknown. **IN Qn AXISn MAS_P** **IN Qn AXISn MAS_C** **IN Qn AXISn SYNC_ERR** - Error if either the slave sync mark or master sync mark is specified as a master cycle position, but master cycle definition is still pending. **FOL MSYNC** - Error if defined as a master position **FOL SSYNC** - Error if defined as a master position
FOL SHIFT cannot start move	This indicates that a command phase shift cannot be performed. **FOL SHIFT #** - Error is already shifting or performing other time based move or **VEL** or **ACCEL** is zero **FOL SHIFT CW,CCW** - Error if **ACCEL** is zero

Master sync mark undefined	This indicates that no master sync mark definition exists. This may be because the **FOL MSYNC** statement was never executed, or was executed with **NO** as the parameter. **IN Qn AXISn SYNC_ERR**
Slave sync mark undefined	Indicates that no slave sync mark definition exists. This may be because the **FOL SSYNC** statement was never executed, or was executed with **NO** as the parameter. **IN Qn AXISn SYNC_ERR**

These statements are designed to be used with the Model 4000-CFM Option.

FOL

Name	**FOL**
Descriptor	Following Parameters
Type	Set-Up
Defauxlt	N/A
Syntax	FOL

Options

TAB	MASTER	SHIFT	ENABLE	RATIO	ETC
TAB	MDIST	MOVEWT	NEWCYC	MAS_CYC	ETC
TAB	WAIT	SMOOTH	MAXACC	MAXVEL	ETC
TAB	VELFF	CYC_OFF	M_SYNC	S_SYNC	ETC
TAB	SYNC_OFF	WIN_P	WIN_W	PTOL	ETC
TAB	LEAD	DIRSET	ENCCHK	CAM	ETC
F1	**F2**	**F3**	**F4**	**F5**	**F6**

Description

FOL *statements can be used only if you have purchased the Model 4000's Following option.* **FOL** statements define master and slave parameters when one or more axes is involved in following an encoder or step output signal. Below is a summary of the Following statements.

FOL MASTER	Specify following master input.
FOL SHIFT	Execute time-based moves upon ratio following moves
FOL ENABLE	Enable and disable following mode.
FOL RATIO	Establish maximum allowed ratio for preset moves, or final ratio for continuous moves.
FOL MDIST	Set master distance for subsequent MOVE statements. MDIST specifies master distance over which preset moves take place, or master distance over which continuous moves change from one ratio to another.
FOL MOVEWT	Next move wait for trigger or master cycle position.
FOL NEWCYC	Define the start of a new master cycle, i.e., set the master cycle position to 0.
FOL MAS_CYC	Define master cycle length.
FOL WAIT	Wait for trigger or master cycle position.
FOL SMOOTH	Specify velocity measurement smoothing.
FOL MAXACC	Define following maximum acceleration
FOL MAXVEL	Define following maximum velocity
FOL VELFF	Enable or disable velocity feed forward
FOL CYC_OFF	Define initial master position
FOL M_SYNC	Define master synchronization mark
FOL S_SYNC	Define slave synchronization mark
FOL SYNC_OFF	Define expected synchronization position difference
FOL WIN_P	Define master window position
FOL WIN_W	Define master window width
FOL PTOL	Define following error tolerance
FOL CAM	Enable or disable cam profiling
DEFINE TRIGDB	Sets debounce time for trigger inputs
FOL DIRSET	Enables or disables direction change setup time
FOL ENCCHK	Enable or disable encoder/motor step check
FOL LEAD	Advance setpoint proportional to slave speed

See Also: **FOLM**, **UNIT MASTER**

FOL MASTER

Name	**FOL MASTER**
Descriptor	Assign Master to Slave
Type	Set-Up
Initial value	None
Range	± MOT or ENC, 1-4
Default	FOL MASTER * * * *
Syntax	FOL MASTER ENC2 MOT4 NO -ENC4

Options	TAB	ENC	NULL	MOT	NO	
	F1	**F2**	**F3**	**F4**	**F5**	**F6**

Description

The **FOL MASTER** statement configures an axis to be a slave, but *does not* automatically enable following. To enable following use the **FOL ENABLE YES** statement. Any incremental or absolute encoder input, or any motor step output can be used as the master for any axis. As soon as the master is specified with the **FOL MASTER** statement, a continuously updated relationship between the position of the slave and the position of the specified master is maintained. The slave motor resolution and velocity ranges are used in these calculations. If the slave is in encoder step positioning (i.e., **MODE E_ABS** or **MODE E_INC**) then the encoder resolution of the slave axis is also used. For that reason, *these parameters may not be changed after the **FOL MASTER** statement configures an axis as a slave.* **FOL MASTER NO** releases an axis from a slave configuration, and returns it to normal operation.

Notice that the master input axis number does not need to be the same as the slave axis number. Axis 1 uses the encoder input on axis #2 as the master, axis #2 is a slave to the step output of axis #4, axis #3 is not configured as a slave, and axis 4 is a slave to the encoder input of that axis. If a slave axis is in encoder mode (**MODE E_INC** or **E_ABS**), or if stall detect or position maintenance is enabled, that axis can not use its own encoder input as the master. Also, a slave can not use its own motor step output as the master input. On power-up, and at the end of every program, no axis is configured as a slave.

A minus sign is allowed as a parameter for the **FOL MASTER** when describing the encoder or motor step master. The minus sign will be needed for applications in which the desired direction of positive master motion results in negative counts on the master. This is particularly true for preset moves as described below. The master can be the motor step output or encoder step input of any axis. Putting a minus sign in front of the master parameter specification in the **FOL MASTER** statement causes the incoming count to be negated before it is used by the slave. The term *master count* refers to the count after negation, if any.

For preset slave moves, the direction the slave travels depends on the mode of operation (absolute or incremental) and the commanded position. However, once a preset slave move is commanded, it will only start moving if the master is counting up. This is true no matter the commanded direction of the slave move.

For continuous slave moves, the master count direction has a different affect. If the commanded move is positive in direction, **MOVE SLEWCW**, and the master is counting up, the actual slave travel direction will be positive. If the commanded move is positive in direction, **MOVE SLEWCW**, and the master is counting down, the actual slave travel direction will be negative. Similar cases exist for slave moves commanded in the negative direction.

*Each of the statements described below indicates the initial value taken as a result of the **FOL MASTER** statement*

The **FOL MASTER** statement re-initializes all **FOL** and **FOLM** parameters each time it is executed. More information about preset and continuous slave moves can be found in the *Technical Considerations* section of this chapter.

See Also: **FOL**, **FOLM**

FOL SHIFT

Name	**FOL SHIFT**
Descriptor	Following Phase Shift Move
Type	Motion
Initial value	None
Range	±99999999 steps after scaling
Default	FOL SHIFT * * * *
Syntax	FOL SHIFT 12500 CW Q1 -525.67

Options						
	TAB	Q	NULL	CW	CCW	ETC
	TAB	STOP	KILL			ETC
	F1	**F2**	**F3**	**F4**	**F5**	**F6**

Description

The **FOL SHIFT** statement allows time-based slave moves to be super-imposed on continuous following moves. Continuous shift moves in the CW or CCW direction, as well as preset shift moves of defined or variable distances may be commanded while a slave is performing a **SLEWCW** or **SLEWCCW** ratio move at any constant ratio. The velocity and direction of the **SHIFT** is independently super-imposed on whatever velocity and direction results from the ratio and motion of the master. The **SHIFT** is not a change in ratio. It is a velocity added to a ratio. Distances are scaled by **UNIT POS**. The **FOL SHIFT** parameters **STOP** and **KILL** can be used to halt a continuous or preset **FOL SHIFT** move (CW or CCW). The example below shows how to stop a **FOL SHIFT** continuous move. It should be noted that **FOL SHIFT** is similar in execution to **MOVE** and not **MOVI**. The entire preset distance shift or ramp to shift velocity must finish before the Model 4000 proceeds to the next statement. As with **MOVE**, however, a **FOL SHIFT** statement may be interrupted by an **ON** condition becoming true.

The most recently commanded **VEL** and **ACCEL** for the slave axis will determine the speed at which the **FOL SHIFT** move takes place. The velocity commanded will be added to the present speed at which the slave is moving, up to the velocity limit defined with the **FOL MAXVEL** statement. For example, assume a slave is traveling at 1 rps in the positive direction while following a master. If a **FOL SHIFT** move is commanded in the positive direction at 2 rps, the slave's actual velocity (after acceleration) will be 3 rps, assuming that **FOL MAXVEL** is greater than 3 rps.

A **FOL SHIFT** move may be needed to adjust slave position on the fly because of some load condition which changes during the continuous following move. For example, suppose an operator is visually inspecting the slave's motion with respect to the master. If they notice that the master and slave are out of synchronization, it may be desirable to have an interrupt programmed that will allow the operator to move the slave at a super-imposed correction speed until the operator chooses to have the slave start tracking the master again. The example below illustrates this.

See Also: **FOL** RATIO, FOL ENABLE

Example

Assume all scale factors and set-up parameters have been entered for the master and slave. In the example below, the slave (axis #1) is continually following the master at a 1:1 ratio. If the operator notices some mis-alignment between master and slave, there is 1 of 2 pushbuttons he can press to move the slave in the CW or CCW direction until the button is released. After the adjustment, the program continues on as before.

Statement	**Description**
`PATT1 XXXX ... XX10`	`'Define input pattern #1`
`PATT2 XXXX ... XX01`	`'Define input pattern #2`
`ON IN24 = PATT1 GOTO SHIFT+`	`'Interrupt to shift slave in the CW 'direction when pattern 1 active`
`ON IN24 = PATT2 GOTO SHIFT-`	`'Interrupt to shift slave in the CCW 'direction when pattern 2 active`
`FOL MASTER ENC 4 * * *`	`'Axis 4 Encoder input is the master`
`FOL RATIO 1:1 * * *`	`'Set slave to master following ratio`
`FOL ENABLE YES * * *`	`'Enable following mode on axis #1`
`MOVE SLEWCW * * *`	`'Start following master continually`
`LABEL MAINLOOP`	`'Main program loop`

```
        .                              'All other program operation takes place
                                       'within this loop
GOTO MAINLOOP                          'Repeat main program loop
DONE                                   'End of program
LABEL SHIFT+                           'Subroutine to shift in the CW 'direction
FOL SHIFT CW * * *                     'Start slave shift in CW directions
WAIT FOR BIT2 = 0                      'Continue shift until bit2 is 'deactivated
FOL SHIFT STOP * * *                   'Stop shift move
ON IN24 = PATT1 GOTO SHIFT+            'Re-enable interrupt for future shifts
GOTO MAINLOOP                          'Return to main program loop
LABEL SHIFT-                           'Subroutine to shift in the CCW 'direction
FOL SHIFT CCW * * *                    'Start slave shift in the CCW 'direction
WAIT FOR BIT1 = 0                      'Continue shift until bit #1 is
                                       'deactivated
FOL SHIFT STOP * * *                   'Stop shift move
ON IN24 = PATT2 GOTO SHIFT-            'Re-enable interrupt for future shifts
GOTO MAINLOOP                          'Return to main program loop
```

FOL ENABLE

Name	**FOL ENABLE**
Descriptor	Enable or Disable Following
Type	Programming
Initial value	NO
Default	FOL ENABLE * * * *
Syntax	FOL ENABLE NO * YES YES
Options	TAB YES NULL NO
	F1 F2 F3 F4 F5 F6

Description

The **FOL ENABLE** statement indicates whether subsequent moves will be following a master (**FOL ENABLE YES**) or normal time-based moves (**FOL ENABLE NO**). The term *Following mode* simply means that **FOL ENABLE YES** has been given, and that the motion of the slave is dependent on the motion of the master at all times. If **FOL ENABLE NO** is given the motion of the master is still monitored but the motion of the slave is independent of the master. In order to enable following mode, the master must have been previously specified with the **FOL MASTER** statement. The **FOL ENABLE** statement may be used in the standard stationary positioning system, or in the moving positioning system.

See Also: **FOL RATIO, FOL MDIST**

FOL RATIO

Name	**FOL RATIO**
Descriptor	Slave to Master Following Ratio
Type	Set-Up
Initial value	Ø
Range	Maximum of 127 slave steps per master step
Default	FOL RATIO * * * *
Syntax	FOL RATIO 10:1.4 Q5:1 * 1
Options	TAB Q NULL :
	F1 F2 F3 F4 F5 F6

Description

The **FOL RATIO** statement establishes the ratio between slave and master speed and position in terms of user units. For a preset move, it is the maximum allowed ratio, and for a continuous move, it is the final ratio reached by the slave. The ratio can be specified either with standard decimal numbers, or numeric Q variables. **FOL RATIO** is specified with two positive numbers, but it applies to moves in both directions. Actual slave direction will depend on commanded moves and master direction.

Assume **FOL RATIO** is set to 0.5:0.3 for an axis. The first parameter is scaled by the **UNIT POS** value to give slave steps. The second parameter is scaled by the **UNIT MASTER** value to give master steps. For a **UNIT POS** of 25000 and a **UNIT MASTER** of 4000, the slave to master step ratio would be 0.5*25000 to 0.3*4000, or 125 slave steps for every 12 master steps. If no second parameter is specified, it is assumed to be 1. Numeric Q variables can be used with this statement for slave and/or master parameters. The Model

4000 divides the scaled numerator and denominator to calculate the ratio. After scaling, the maximum value is 127 slave steps for every master step.

See Also: **FOL ENABLE**, **UNIT MASTER**, **UNIT POS**

Example

In the statements below, assume the master has a 1000 line incremental encoder on the back of a motor and programming units are to be rps. This yields a post quadrature **UNIT MASTER** scale factor of 4000. The motor resolution of the slave axis is 25000 steps/rev. A slave **UNIT POS** scale factor of 25000 provides consistent user units.

The slave will start ramping to a ratio of 1:1 when trigger #1 goes active. This means the actual step ratio of slave to master is 25000 to 4000, or 6.25 slave steps for every master. After 25 master revolutions, the slave will decelerate to a 0.5:1 ratio (3.125 slave steps for every master). After a total of 75 master revolutions, the slave will stop and repeat the cycle on trigger #1.

Statement	Description
`UNIT POS 25000 * * *`	`'Set slave scale factor to 25000 steps/rev`
`UNIT MASTER 4000 * * *`	`'Set master scale factor to 4000 steps/rev`
`FOL MASTER ENC1 * * *`	`'Assign encoder input #1 as master for axis '#1`
`FOL MDIST 1 * * *`	`'Slave should accelerate over 1 master 'revolution`
`FOL MAS_CYC 100 * * *`	`'Set master cycle length to 100 revs`
`FOL RATIO 1:1 * * *`	`'Initial slave to master ratio is 1 to 1`
`FOL ENABLE YES * * *`	`'Enable following on axis #1`
`LABEL ST_MOVE`	`'Label to repeat move`
`FOL NEW_CYC TRIG1 * * *`	`'New master cycle (counter at 0) on trigger '#1`
`FOL MOVEWT TRIG1 * * *`	`'Wait on next move until trigger 1 is 'active`
`FOL RATIO 1:1 * * *`	`'Following ratio is back to 1 to 1`
`MOVE SLEWCW * * *`	`'Start continuous following move on trigger '#1`
`FOL MOVEWT 25 * * *`	`'Wait on next move for master position 25`
`FOL RATIO .5:1 * * *`	`'Set new following ratio`
`MOVE SLEWCW * * *`	`'Move to new following ratio at master 'position 25`
`FOL MOVEWT 75 * * *`	`'Wait on next move for master position 75`
`FOL RATIO 0:1 * * *`	`'Set new following ratio for slave to stop`
`MOVE SLEWCW * * *`	`'Move to new following ratio at master 'position 75`
`GOTO ST_MOVE`	`'Repeat the cycle`
`DONE`	`'End of program`

FOL MDIST

Name	**FOL MDIST**
Descriptor	Master Move Distance
Type	Set-Up
Initial value	Ø
Range	±99999999 master steps
Default	FOL MDIST * * * *
Syntax	FOL MDIST 250.32 Q2 * 0.0
Options	TAB Q NULL
	F1 **F2** **F3** **F4** **F5** **F6**

Description

If a slave is doing preset moves, the **FOL MDIST** statement indicates the master distance over which the next preset moves will take place. Or, if a slave is in continuous mode, **FOL MDIST** is the master distance over

which acceleration or deceleration from the current ratio to the new ratio takes place. **FOL MDIST** is specified in user units and is scaled by the **UNIT MASTER** parameter. Numeric Q variables can be used with this statement.

In 4000 Following, acceleration for a slave can be specified either by master distance (**FOL MDIST**), or by the **ACCEL** statement. Whichever of these statements most recently precede the **MOVE** or **MOVI** statement will give the parameter used to determine the move profile. Specifying an acceleration for the slave means that the acceleration ramp is time-based and there is no position relationship between master and slave until the commanded following ratio is reached. Specifying a master distance for the slave's move profile ensures a precise position relationship between master and slave during all phases of the profile. Whenever the position relationship between master and slave is important, the **FOL MDIST** method should be used.

If a slave is in continuous mode and the master is starting from rest, setting **FOL MDIST** to 0 will ensure precise tracking of the master's acceleration ramp. The example below illustrates how this might take place.

See Also: **FOL ENABLE**, **UNIT MASTER**, **FOL RATIO**

Example

Statement	Description
`UNIT MASTER 4000 * * *`	`'Master scale factor is 4000 steps/rev`
`FOL MASTER ENC1 * * *`	
`FOL MDIST 0 * * *`	`'Assign following acceleration distance to 0 'master revs, i.e., instantaneous`
`FOL RATIO 1 * * *`	`'Set following ratio to 1 to 1`
`FOL ENABLE YES * * *`	`'Enable following on axis #1`
`MOVE SLEWCCW * * *`	`'Begin following master, if the master is 'not moving, slave will remain at rest until 'master moves. At this time it will track 'master precisely`

FOL MOVEWT

Name	**FOL MOVEWT**
Descriptor	Wait for Trigger or Cycle Position, Next Move
Type	Programming
Initial value	NO
Range	TRIG 1-4, ±99999999 master steps
Default	FOL MOVEWT * * * *
Syntax	FOL MOVEWT TRIG1 NO 236.50 Q4
Options	TAB Q NULL TRIG NO
	F1 F2 F3 F4 F5 F6

Description

The **FOL MOVEWT** statement does not cause the 4000 to suspend program execution immediately. It specifies that the next move will not start until the specified trigger or master cycle position has been reached. This statement is useful in delaying subsequent moves until the master has reached the required position or an object has been sensed. When a master cycle position is specified, it is always an *absolute* position relative to the start of the last master cycle.

If a **MOVEWT** has been specified, and the 4000 program reaches a **MOVE** statement, the following sequence of events takes place. Moves for axes which do not have a wait condition specified will start and complete their moves as normal. This includes axes which are not configured as slaves. Axes that have a wait condition specified will wait for that condition to be satisfied before beginning their moves. The next program statement will execute when all preset moves (following or non-following) are complete and/or the cruise velocity has been reached for continuous moves.

For a **MOVI** statement after a **MOVEWT**, the program action taken is quite different. Axes that have no wait condition specified, or are not configured as slaves, will begin their moves. Axes that have a wait condition specified will wait to begin their moves until the condition becomes true, but program execution will continue immediately.

The Model 4000 records the **MOVEWT** condition as soon as the statement is executed, but will not start checking and waiting for the **MOVEWT** condition until the **MOVE** or **MOVI** statement is executed. If the condition is a trigger, and the trigger input goes active before the **MOVE** or **MOVI** statement is executed, it will not be noticed, and the move will wait for the next trigger. If **FOL MOVEWT NO** is given, or if a **MOVE STOP** or **MOVE KILL** is executed, all wait conditions for that axis are cleared, and the next **MOVE** or **MOVI** for that axis begins immediately. If the **MOVEWT** condition is a master cycle position, and that position has already been exceeded before the **MOVE** or **MOVI** statement is executed, that axis flags a **MOVEWT MISSED** condition. This condition can be detected with the **ON WT_ERR** statement, and is cleared by the next **FOL MOVEWT** command for that axis.

If a master cycle position is used in a **FOL MOVEWT** statement, the wait may occur while a master cycle is pending definition or a trigger. In that case the wait will include the trigger, then the cycle position in the master cycle defined by that trigger. If the pending status is cleared by either a **FOL NEWCYC NO** or a **FOL NEWCYC IMMED**, the wait is cleared also.

If the master cycle position specified with **FOL MOVEWT** is larger than the master cycle length, (**FOL MAS_CYC**), the 4000 simply waits for the specified position relative to the start of the current cycle. For example, if the master cycle length is 10 inches, current master cycle position is 5, and an **FOL MOVEWT** is issued for master cycle position 38 inches, the 4000 will wait for 33 more master inches before starting the next move. The value 38 is not referenced from where the master is currently, but rather total master inches from the start of the current cycle, even if it is necessary to wait for more than one complete cycle length. The actual master cycle position counter is still being reset to 0 when the master cycle is complete, but the **FOL WAIT** and **MOVEWT** will allow waiting for more than 1 cycle if necessary.

The master cycle position specified is in terms of user units and is scaled by the **UNIT MASTER** parameter. Numeric Q variables may be used to specify master cycle positions. Note that a trigger input number does not need to match the axis number to which the **FOL MOVEWT** is assigned. Also, keep in mind that an axis does not need to be in following mode (**FOL ENABLE YES**) to utilize the **FOL MOVEWT** and master cycle concept. However, a master must have been previously assigned using the **FOL MASTER** statement.

If cam profiling is enabled (**FOL CAM YES**), master cycle positions are determined by execution of the profile. For this reason, master cycle positions *may not* be used as a pararmeter with **FOL MOVEWT** when cam profiling is enabled. For a complete discussion of master cycle parameters with cam profiling, please refer to the section titled *Profiles and Master Cycles*.

The section titled *Following Wait Statements* provides general information about **FOL WAIT** and **FOL MOVEWT**, as well as the differences in their use. The *Continuous Cut to Length* example in that section uses **FOL MOVEWT Ø** to ensure precise cut lengths.

See Also: **FOL MAS_CYC**, **FOL NEWCYC**, **FOL WAIT**, **FOL CYC_OFF**, **IN FOL TRIG**, **ON WT_ERR**

FOL_NEWCYC

Name	**FOL_NEWCYC**
Descriptor	Define Master Cycle
Type	Programming
Initial value	NO
Range	TRIG 1-4
Default	FOL NEWCYC * * * *
Syntax	FOL NEWCYC TRIG1 TRIG3 IMMED *
Options	TAB TRIG NULL IMMED NO
	F1 F2 F3 F4 F5 F6

Description

The **FOL NEWCYC** statement defines the beginning of a master cycle by setting the master cycle position to the value most recently specified with **FOL CYC_OFF**. If **IMMED** is specified, the master cycle position is set immediately. If a trigger is specified as the parameter, the 4000 will record the instruction to set the master cycle position when the specified trigger occurs. In either case, program flow continues normally. In the latter case, the master cycle is pending definition on the specified trigger, even though statements continue to execute. If an application requires suspending program flow until the trigger or a subsequent master cycle position occurs, the **FOL WAIT** statement may be used. **FOL NEWCYC NO** and **FOL NEWCYC IMMED** will remove the pending status of the master cycle definition. In this case, the former master cycle definition becomes effective again, and the specified trigger will not cause a new cycle definition. This also clears or undoes any **FOL WAIT** or **FOL MOVEWT** for master cycle position which was issued while the new cycle definition was pending a trigger input.

A new cycle automatically occurs, i.e., the master cycle position is set to 0, when the master cycle length (**FOL MAS_CYC**) is reached, even if no **FOL NEWCYC** statement has been executed.

If cam profiling is enabled (**FOL CAM YES**), the beginning of a master cycle is determined by execution of the profile. When cam profiling is enabled, The **FOL NEWCYC** statement is used to mark the repetitive portion of a cycle, and may only use the **IMMED** parameter. For a complete discussion of master cycle parameters with cam profiling, please refer to the section titled *Profiles and Master Cycles*.

See Also: **FOL MAS_CYC**, **FOL WAIT**, **FOL MOVEWT**, **FOL CYC_OFF**, **IN FOL TRIG**

FOL MAS_CYC

Name	**FOL MAS_CYC**
Descriptor	Master Cycle Length
Type	Set-Up
Initial value	Ø
Range	Ø - 99999999 master steps
Default	FOL MAS_CYC * * * *
Syntax	FOL MAS_CYC * 100 4.737 Q8
Options	TAB Q NULL
	F1 F2 F3 F4 F5 F6

Description

The **FOL MAS_CYC** statement defines the length of the master cycle in user units. This value is scaled by the **UNIT MASTER** parameter. Numeric Q variables can be used with this statement. The initial value for **FOL MAS_CYC** is 0, which means that the default master cycle length is infinite.

The concept of a master cycle may be useful when moves or other events must be initiated at certain master positions. By specifying a master cycle length, periodic actions may be programmed in a loop or with subroutines which refer to cycle positions, even if the master runs continuously. It is possible to program the 4000 to suspend program operation or wait for moves until certain master cycle positions or trigger inputs. The master cycle length, **FOL MAS_CYC**, should be defined before the functions which wait for periodic master cycle positions are used. An axis need not be in following

mode (i.e., **FOL ENABLE YES**) to utilize the concept of a master cycle. However, a master must have been previously assigned with the **FOL MASTER** statement.

If cam profiling is enabled (**FOL CAM YES**), the master cycle length is determined by the sum of the master distances in the repetitive portion of the profile. For this reason, the **FOL MAS_CYC** statement is ignored when cam profiling is enabled. For a complete discussion of master cycle parameters with cam profiling, please refer to the section titled *Profiles and Master Cycles*.

See Also: **FOL NEWCYC**, **FOL WAIT**, **FOL MOVEWT**, **FOL CYC_OFF**

FOL WAIT

Name	**FOL WAIT**
Descriptor	Wait for Trigger or Master Cycle Position
Type	Programming
Initial value	NO
Range	TRIG 1-4, ±99999999 master steps
Default	FOL WAIT * * * *
Syntax	FOL WAIT TRIG2 NO 14.53 Q8
Options	TAB Q NULL TRIG NO
	F1 F2 F3 F4 F5 F6

Description

The **FOL WAIT** statement causes the 4000 to suspend program execution until the specified trigger or master cycle position has occurred. This function is useful in delaying subsequent I/O operation until the master has achieved the required position or an object has been sensed. When a master cycle position is specified, it is always an *absolute* position relative to the start of the last master cycle.

If the **FOL WAIT** condition is a master cycle position, and that position has already been exceeded when the **FOL WAIT** statement is executed, then program flow proceeds immediately to the next statement. No error condition results from this.

If a master cycle position is used in a **FOL WAIT** statement, the wait may occur while a master cycle is pending definition or a trigger. In that case the wait will include the trigger, then the cycle position in the master cycle defined by that trigger. If the pending status is cleared by either a **FOL NEWCYC NO** or a **FOL NEWCYC IMMED**, the wait is cleared also.

If the master cycle position specified with **FOL WAIT** is larger than the master cycle length, (**FOL MAS_CYC**) the 4000 simply waits for the specified position relative to the start of the current cycle. For example, if the master cycle length is 25 inches, current master cycle position is 5, and an **FOL WAIT** is issued for master cycle position 67, the 4000 will wait for 62 more inches of master travel before continuing program execution. The value 67 is not referenced from where the master is currently, but rather total master inches from the start of the current cycle, even if it is necessary to wait for more than one complete cycle length. Here, our wait was for about 2.5 master cycles. The actual master cycle position counter is still being reset to 0 when every master cycle is complete, but the **FOL WAIT** and **MOVEWT** will allow waiting for more than 1 cycle if necessary.

The master cycle position specified is in terms of user units and is scaled by the **UNIT MASTER** parameter. Numeric Q variables may be used to specify master cycle positions. Note that a trigger input number need not match the axis number to which the **FOL WAIT** is assigned. For example, axis #2 could be performing a **FOL WAIT**, waiting for Trigger 4. Also, keep in mind that an axis does not need to be in following mode (i.e., **FOL ENABLE YES**) to utilize the **FOL WAIT** and master cycle concept. However, a master must have been previously assigned using the **FOL MASTER** statement.

FOL WAIT is similar to other **WAIT**s in that an **ON** condition may clear the **FOL WAIT** and allow subsequent statements to execute. A **FOL WAIT NO**

will also clear the wait. If one program is waiting on a **FOL WAIT**, another program may clear the wait by executing a **FOL WAIT NO**, assuming both programs are running under Multi Tasking

See Also: **FOL MAS_CYC**, **FOL NEWCYC**, **FOL MOVEWT**, **FOL CYC_OFF**, **IN FOL TRIG**

Example

In the example below, the master is an encoder mounted to gearing on a conveyor line. The gearing results in 16,000 encoder steps per conveyor inch. The slave on axis one is a 25,000 step/rev microstepper on a 36" long, 4 pitch leadscrew. The slave waits for the product to be sensed on the conveyor, accelerates to a 1 to 1 ratio, waits for a safe location to actuate the stamping equipment, then applies an inked stamp to the product at the correct location. After the stamp is placed, the slave quickly moves back to the starting position and waits for the next product. The example illustrates how the **FOL WAIT** command can be used to wait for master cycle positions in order to coordinate motion.

```
Statement                          Description
UNIT VEL 100000 * * *              'Set slave velocity scale factor to
                                   '100000
UNIT ACCEL 100000 * * *            'Set slave accel scale factor to 100000
UNIT POS 100000 * * *              'Set slave position scale factor to
                                   '100000 to program in inches
ACCEL 10 * * *                     'Acceleration = 10 inches/sec/sec
VEL 5 * * *                        'Velocity = 5 inches/sec (non-following
                                   'moves)
MODE M_ABS * * *
UNIT MASTER 16000 * * *            'Set master scale factor to program in
                                   'inches
FOL MASTER ENC1 * * *              'Assign encoder input #1 as master
FOL RATIO 1 * * *                  'Following ratio 1 to 1 slave to master
                                   'inch.
FOL MDIST 1 * * *                  'Accelerate the slave over 1 master inch
                                   'for following moves
FOL MAS_CYC 40 * * *               'Master cycle length is 40 inches
LABEL INK_ON                       'Label to repeat inking process
FOL ENABLE YES * * *               'Enable following on axis #1
FOL NEWCYC TRIG2 * * *             'Begin new master cycle on trigger #2
                                   '(product sensed on conveyor)
FOL MOVEWT TRIG2 * * *             'Start next move when trigger #2 is
                                   'active
MOVE SLEWCW * * *                  'Start continuous slave move on trigger
                                   '#2
FOL WAIT 10.5 * * *                'Wait until master position is 10.5
                                   'inches, this is when the stamping
                                   'device can be actuated without
                                   'mechanical damage to the leadscrew
                                   'assembly
OUT BIT7 = 1                       'Turn on actuator to place ink stamp on
                                   'product
FOL WAIT 12.0 * * *                'Wait until master position is 12
                                   'inches.  The ink stamp is pressed in
                                   'place by a stationary roller 1.5" in
                                   'length
OUT BIT7 = 0                       'Turn actuator off
MOVE STOP * * *                    'Stop slave move
FOL ENABLE NO * * *                'Disable following on axis #1
MOVE 0 * * *                       'Move back to home position
GOTO INK_ON                        'Begin cycle again on trigger #2
DONE                               'End of program
```

FOL SMOOTH

Name	**FOL SMOOTH**
Descriptor	Velocity Measurement Smoothing
Type	Set-Up
Initial value	1
Range	1-4
Default	FOL SMOOTH * * * *
Syntax	FOL SMOOTH 4 2 * 1
Options	TAB Q NULL
	F1 F2 F3 F4 F5 F6

Description

The **FOL SMOOTH** statement specifies the degree to which the Model 4000 filters and smoothes the measurements of master position and velocity. Valid choices of 1, 2, 3, and 4 for the **FOL SMOOTH** statement correspond to velocity averaging periods of 4, 8, 16, and 32 milliseconds, respectively.

The 4000 samples the master position every 2 msec, and by measuring the change in master position over a fixed number of samples, the master velocity is calculated. The current master position, the current ratio, and the estimation of current master velocity are all used to calculate the setpoint slave position for the next sample. If left unspecified, the following algorithm defaults to **FOL SMOOTH 1**.

For applications in which the master is vibrating or moving very slowly, smoother slave velocity may result with an **FOL SMOOTH** parameter greater than 1. When the master is changing velocity very quickly, or if the number of master pulses per sample period is large, an **FOL SMOOTH** value of 1 is recommended. Setting this value will best be done on a trial and error basis, since each application has widely varying components and requirements.

See Also: **FOL MAXVEL, FOL MAXACC, FOL VELFF**

FOL MAXACC

Name	**FOL MAXACC**
Descriptor	Slave Maximum Acceleration
Type	Set-Up
Initial value	Maximum for motor resolution
Default	FOL MAXACC * * * *
Syntax	FOL MAXACC 200 Q3 * *
Options	TAB Q NULL
	F1 F2 F3 F4 F5 F6

Description

The **FOL MAXACC** statement sets the maximum acceleration for slave axes. The **FOL MAXACC** statement accepts numeric Q variables as an argument and is scaled by the **UNIT ACCEL** parameter.

The profile of the acceleration for a slave axis can be defined with one of two statements: **ACCEL** or **FOL MDIST**. If the **ACCEL** command is used to determine slave acceleration, the acceleration ramp will be a time-based acceleration at the value specified, as long as the **ACCEL** value is not larger than **FOL MAXACC**. If the **FOL MDIST** command is used to define a move profile, the resulting acceleration will be determined in part by the speed of the master, however, the acceleration ramp the slave takes will never exceed the **FOL MAXACC** value.

For both cases above, if the required acceleration is larger than **FOL MAXACC**, the slave will begin falling behind its commanded position. The 4000 will attempt to make up this position error as soon as the commanded accel falls below **FOL MAXACC**. An error correction velocity is added to that implied by the ratio setpoint. The velocity used to make up the error is limited to that specified with **POSM MAXVEL**.

As with **FOL MAXVEL**, **FOL MAXACC** should be determined and defined early in the development stage of an application to prevent any damage to the load on the slave axis when unexpectedly high accelerations are

commanded. The torque available from the slave motor will also be a determining factor in this parameter in order to prevent motor stalls.

See Also: **FOL SMOOTH, FOL MAXVEL, FOL VELFF**

FOL MAXVEL

Name	**FOL MAXVEL**
Descriptor	Slave Maximum Velocity
Type	Set-Up
Initial value	Maximum for motor resolution
Range	Limited by slave motor resolution maximum velocity
Default	FOL MAXVEL * * * *
Syntax	FOL MAXVEL 100 Q2 * *
Options	TAB Q NULL
	F1 F2 F3 F4 F5 F6

Description

The **FOL MAXVEL** statement sets the maximum velocity at which slave axes may travel. The **FOL MAXVEL** statement accepts numeric Q variables as an argument and is scaled by the **UNIT VEL** parameter.

Normally in a following application, the slave velocities will be known based on the normal speed of the master and the commanded following ratios. In some cases, however, the master speed may be higher than normal, the slave may be commanded to perform a shift move, or some other event may occur which will cause the slave to travel at a velocity higher than expected. In these cases, the 4000 will raise the speed of the slave as necessary to perform the required move, but only up to the **FOL MAXVEL**. If the commanded speed is higher than **FOL MAXVEL**, the slave axis will start falling behind its commanded position. The 4000 will attempt to make up this position error as soon as the commanded speed falls below **FOL MAXVEL**. An error correction velocity is automatically added to that implied by the ratio setpoint. The velocity used to make up the error is limited to that specified with **POSM MAXVEL**.

The **FOL MAXVEL** should be determined and defined early in the development stage of an application to prevent any damage to the load on the slave axis when unexpectedly high velocities are commanded.

See Also: **FOL SMOOTH, FOL MAXACC, FOL VELFF**

FOL VELFF

Name	**FOL VELFF**
Descriptor	Enable or Disable Velocity Feed Forward
Type	Set-Up
Initial value	YES
Default	FOL VELFF YES YES YES YES
Syntax	FOL VELFF YES NO * *
Options	TAB YES NULL NO
	F1 F2 F3 F4 F5 F6

Description

The **FOL VELFF** statement allows the user to enable or disable velocity feed forward in the 4000 Following algorithm. Velocity feed forward is activated by default in the Following algorithm, but can be turned off as desired with the **FOL VELFF** statement.

The 4000 measures master position every two milliseconds, and calculates a corresponding slave setpoint. This calculation and achieving the subsequent slave setpoint position require 4 milliseconds. Enabling velocity feed-forward (**FOL VELFF YES**) eliminates any lag in slave position which would be dependent on master speed. It may be desirable to deactivate velocity feed forward when maximum slave smoothness is important and minor phase delays can be accommodated. A detailed discussion of velocity feed-forward is given in the section *Technical Considerations for Following*.

See Also: **FOL SMOOTH, FOL MAXACC, FOL MAXVEL**

FOL CYC_OFF

Name	**FOL CYC_OFF**
Descriptor	Master Cycle Offset Position
Type	Set-Up
Initial value	Ø
Range	±99999999 motor steps
Default	FOL CYC_OFF * * * *
Syntax	FOL CYC_OFF * -10.2 Q8 *
Options	TAB Q NULL
	F1 F2 F3 F4 F5 F6

Description

The **FOL CYC_OFF** statement defines the initial master cycle position in user units. The initial master cycle position is assigned with a new master cycle defined via the **FOL NEWCYC** statement. This value is scaled by the **UNIT MASTER** parameter. Numeric Q variables can be used with this statement. The default value for **FOL CYC_OFF** is 0, which means that the master cycle position will be 0 upon definition of a new master cycle (see **FOL NEWCYC**).

The concept of an initial master cycle offset may be useful if new master cycle definition must take place at a master position which is different from what needs to be considered the beginning of a periodic cycle. The initial position defined with **FOL CYC_OFF** applies to the first cycle only. When a master cycle is complete, the master cycle position rolls over to zero. A negative offset would be used if some master travel were desired before master cycle position was zero. A positive offset would be used if it was necessary to enter the first master cycle at a position other than 0. For example, suppose **FOL MAS_CYC** was set to 20 and **FOL CYC_OFF** was set to 7. When the **FOL NEWCYC** definition takes place, either via **FOL NEWCYC IMMED** or the specified trigger, the initial master cycle position will be 7. Rollover will occur after the master travels 13 more units, and the master cycle position would go to zero.

If cam profiling is enabled (**FOL CAM YES**), the master cycle offset position is determined by the sum of the master distances in the lead in portion of the profile. For this reason, the **FOL CYC_OFF** statement is ignored when cam profiling is enabled. For a complete discussion of master cycle parameters with cam profiling, please refer to the section titled *Profiles and Master Cycles*.

See Also: **FOL NEWCYC, FOL MAS_CYC, FOL_WAIT, FOL_MOVEWT**

FOL M_SYNC

Name	**FOL M_SYNC**
Descriptor	Define Master Synchronization Mark
Type	Set-Up
Initial value	NO
Range	Trig 1-4, 0 - 99999999 master steps
Default	FOL M_SYNC * * * *
Syntax	FOL M_SYNC TRIG1 Q1 * 50
Options	TAB Q NULL TRIG NO
	F1 F2 F3 F4 F5 F6

Description

The **FOL M_SYNC** statement defines an event which will cause the slave position to be read and saved for future periodic synchronization calculations. This external event may be any of the four trigger inputs TRIG1 through TRIG4, or a master cycle position. If a master cycle position is used, the value is scaled by the **UNIT MASTER** parameter. Numeric Q variables may be used with this statement. The default value for **FOL M_SYNC** is that the master sync mark is not defined, which means that the periodic synchronization features may not be used.

There are many applications in which periodic operations must occur in intervals which are not perfectly repeatable. The 4000 allows the user to define two external events, or "marks", which capture the slave position.

These are called *Master Sync Mark* and *Slave Sync Mark*, and are defined with the **FOL M_SYNC** and **FOL S_SYNC** statements respectively. If the sync mark is defined as a trigger, the slave position is captured on each occurrence of that trigger. If the sync mark is defined a master cycle position, the slave position is captured *on only the first occurrence of that cycle position* each time a new cycle is defined, or a master cycle has completed and rolled over to start a new cycle. For a complete understanding of a sync mark definitions and their use in periodic synchronization, please refer to the section titled *Periodic Master/Slave Synchronization* earlier in this chapter.

See Also: **FOL M_SYNC**, **FOL SYNC_OFF**, **IN Qn = FOL AXISn SYN_ERR**, **IN Qn = FOL AXISn M_SYNC**

FOL S_SYNC

Name	**FOL S_SYNC**
Descriptor	Define Slave Synchronization Mark
Type	Set-Up
Initial value	NO
Range	Trig 1-4, ±99999999 slave steps
Default	FOL S_SYNC * * * *
Syntax	FOL S_SYNC TRIG1 Q1 * 50

Options	TAB	TRIG	Q	NO	NULL	
	F1	**F2**	**F3**	**F4**	**F5**	**F6**

Description

The **FOL S_SYNC** statement defines an event which will cause the slave position to be read and saved for future periodic synchronization calculations. This external event may be any of the four trigger inputs TRIG1 through TRIG4, or a master cycle position. If a master cycle position is used, the value is scaled by the **UNIT MASTER** parameter. Numeric Q variables may be used with this statement. The default value for **FOL S_SYNC** is that the slave sync mark is not defined, which means that the periodic synchronization features may not be used.

There are many applications in which periodic operations must occur in intervals which are not perfectly repeatable. The 4000 allows the user to define two external events, or marks, which capture the slave position. These are called *Master Sync Mark* and *Slave Sync Mark*, and are defined with the **FOL M_SYNC** and **FOL S_SYNC** statements respectively. If the sync mark is defined as a trigger, the slave position is captured on each occurrence of that trigger. If the sync mark is defined as a master cycle position, the slave position is captured *on only the first occurrence of that cycle position* each time a new cycle is defined, or a master cycle has completed and rolled over to start a new cycle. For a complete understanding of a sync mark definitions and their use in periodic synchronization, please refer to the section titled *Periodic Master/Slave Synchronization* earlier in this chapter.

See Also: **FOL M_SYNC**, **FOL SYNC_OFF**, **IN Qn = FOL AXISn SYN_ERR**, **IN Qn = FOL AXISn S_SYNC**

FOL SYNC_OFF

Name	**FOL SYNC_OFF**
Descriptor	Define Expected Synchronization Position Difference
Type	Set-Up
Initial value	Ø
Range	±99999999 slave steps
Default	FOL SYNC_OFF * * * *
Syntax	FOL SYNC_OFF Q2 10 -5 *
Options	TAB Q NULL
	F1 F2 F3 F4 F5 F6

Description

The **FOL SYNC_OFF** statement defines the expected synchronization offset in the user's slave position units. This value is scaled by the **UNIT POS** parameter. Numeric Q variables may be used with this statement. The default value for **FOL SYNC_OFF** is 0, which means that a master and slave sync marks are expected to occur simultaneously.

Each time either a master or slave sync mark occurs, the corresponding slave position is captured and saved internally. The **FOL SYNC_OFF** statement defines the expected difference between these captured slave positions. This expected difference is called the *Slave Synchronization Offset*. By defining the offset expected between two positions instead of defining the position expected at a single synchronization mark, continuous motion in one direction is allowed without requiring a continuous re-calculation of the expected slave position. The difference between the actual offset and the expected offset is called the Sync Error. This error may be read into a Q variable using the **IN Qn = FOL AXISn SYN_ERR** statement. To understand exactly how to use this, please refer to the section titled *Periodic Master/Slave Synchronization* earlier in this chapter.

See Also: **FOL M_SYNC**, **FOL S_SYNC**, **IN Qn = FOL AXISn SYN_ERR**

FOL WIN_P

Name	**FOL WIN_P**
Descriptor	Define Master Window Position
Type	Set-Up
Initial value	Ø
Range	0 - 99999999 master steps
Default	FOL WIN_P * * * *
Syntax	FOL WIN_P Q1 10 * *
Options	TAB Q NULL
	F1 F2 F3 F4 F5 F6

Description

The **FOL WIN_P** statement defines the position of a following error detection window in the user's master position units. This value is scaled by the **UNIT MASTER** parameter. Numeric Q variables may be used with this statement. The default value for **FOL WIN_P** is 0, which means that the start of a following error detection window would coincide with the start of a master cycle.

The concept of an error detection window is useful in applications in which a periodic repetitive operation takes place, and precise synchronization is only important during a portion of the cycle. For such an application, the window defines the portion of a cycle in which excess following error is detected, while being ignored in the remainder of the cycle. For a complete discussion of the conditions which may result in following error, please refer to the section titled *Following Performance* earlier in this chapter. The error detection window start position must be less the master cycle length to be valid, i.e., meaningful. If its value is greater than master cycle length, the error detection window will not be valid, and *error detection will occur continuously*. Programmed response to detection of following error is

enabled through use of the **ON FOL_ERR** statement. The **FOL WIN_P** statement may be issued at any time, even while motion is in progress, and takes effect immediately. For a complete discussion of error detection windows and related topics, please refer to the section titled *Monitoring Following Error* earlier in this chapter.

See Also: **FOL PTOL**, **FOL WIN_W**, **ON FOL_ERR**, **IN Qn = FOL AXISn FOL_ERR**

FOL WIN_W

Name	**FOL WIN_W**
Descriptor	Define Master Window Width
Type	Set-Up
Initial value	Ø
Range	0 - 99999999 master steps
Default	FOL WIN_W * * * *
Syntax	FOL WIN_W Q3 2 * *
Options	TAB Q NULL
	F1 F2 F3 F4 F5 F6

Description

The **FOL WIN_W** statement defines the width of a following error detection window in the user's master position units. This value is scaled by the **UNIT MASTER** parameter. Numeric Q variables may be used with this statement. The default value for **FOL WIN_W** is 0, which means that the following error detection window is not valid.

The concept of an error detection window is useful in applications in which a periodic repetitive operation takes place, and precise synchronization is only important during a portion of the cycle. For such an application, the window defines the portion of a cycle in which excess following error is detected, while being ignored in the remainder of the cycle. For a complete discussion of the conditions which may result in following error, please refer to the section titled *Following Performance* earlier in this chapter. The error detection window width must be less the master cycle length and greater than zero to be valid, i.e., meaningful. If its value is less than master cycle length or zero, the error detection window will not be valid, and *error detection will occur continuously*. Programmed response to detection of following error is enabled through use of the **ON FOL_ERR** statement. The **FOL WIN_W** statement may be issued at any time, even while motion is in progress, and takes effect immediately. For a complete discussion of error detection windows and related topics, please refer to the section titled *Monitoring Following Error* earlier in this chapter.

See Also: **FOL PTOL**, **FOL WIN_P**, **ON FOL_ERR**, **IN Qn = FOL AXISn FOL_ERR**

FOL PTOL

Name	**FOL PTOL**
Descriptor	Define Following Error Tolerance
Type	Set-Up
Initial value	Ø
Range	0 - 99999999 slave steps
Default	FOL PTOL * * * *
Syntax	FOL PTOL Q4 * .01 *
Options	TAB Q NULL
	F1 F2 F3 F4 F5 F6

Description

The **FOL PTOL** statement defines the following position error tolerance in the user's slave position units. This value is scaled by the **UNIT POS** parameter. Numeric Q variables may be used with this statement. The default value for **FOL PTOL** is 0, which means that a following position error of any magnitude may be detected.

The concept of an allowable position following error is useful when there is a limit to the acceptable following error in the course of following operation.

Mechanical constraints as well as those imposed by user programming may cause the actual position of the slave to differ from the commanded position. For a complete discussion of the conditions which may result in following error, please refer to the section titled *Following Performance* earlier in this chapter. Programmed response to the detection of following error is enabled through use of the **ON FOL_ERR** statement, and may be limited to specific portions of a master cycle. If the error value ever exceeds the tolerance given with **FOL PTOL** when error detection is enabled, the 4000 will branch to the location specified in the **ON FOL_ERR** statement. The **FOL PTOL** statement may be issued at any time, even while motion is in progress. The new value takes effect immediately, and also clears any previously detected error. This avoids unwanted detection of previous errors. For a complete discussion of related topics, please refer to the section titled *Monitoring Following Error* earlier in this chapter.

See Also: **FOL WIN_P**, **FOL WIN_W**, **ON FOL_ERR**, **IN Qn = FOL AXISn FOL_ERR**

FOL LEAD

Name	**FOL LEAD**
Descriptor	Define Setpoint Lead Time
Type	Set-Up
Initial value	Ø milliseconds
Range	0-250 milliseconds
Default	FOL LEAD * * * *
Syntax	FOL LEAD 0.82 * Q1 *
Options	TAB Q NULL
	F1 F2 F3 F4 F5 F6

Description

The **FOL LEAD** statement defines the lead time for the specified axes. Some drives, such as the Dynaserv step and direction versions, have up to 6 milliseconds of lag, i.e., delayed response to step and direction input. This creates a velocity dependent position lag, which causes a synchronization error. In following, however, every position along the profile is important, because it must correspond to a master position. The setpoint lead feature allows the 4000 to dynamically advance the slave setpoint beyond what would be normally resulting from the profile. It is advanced by the product of the instantaneous velocity and the lead value specified with the **FOL LEAD** statement. Because of possible instability and profile skew, it is best to avoid this feature when possible, i.e., use the default value of zero. If it is necessary to compensate for drive lag, the proper value will probably need to be determined empirically.

See Also: **FOL DIRSET**, **FOL ENCCHK**

FOL DIRSET

Name	**FOL DIRSET**
Descriptor	Enable direction change setup time
Type	Setup
Initial value	YES
Range	FOL DIRSET * * * *
Default	FOL DIRSET YES NO * YES
Syntax	FOL LEAD 0.82 * Q1 *
Options	TAB YES NULL NO
	F1 F2 F3 F4 F5 F6

Description

The **FOL DIRSET** statement enables or disables the default 2 millisecond direction input change setup time of the 4000. Most Compumotor steppers require some minimum time for the drive to respond to a change on the direction input before any steps are given in the new direction. In normal positioning applications, motion always stops before a move is commanded in the opposite direction, so this requirement is of no consequence. In following however, the master may change direction, resulting in a direction

change on the slave without a new move command. When the setup time is enabled, the 4000 temporarily saves the steps which would have been sent with the direction change. These steps are sent during the next 2 millesecond update. Some drives, such as the Dynaserv and Z drives, do not have a direction change setup requirement. In order to facilitate smooth following on these drives, the **FOL DIRSET** statement may be used to disable the default 2 millisecond direction change setup.

See Also: **FOL LEAD**, **FOL ENCCHK**

FOL ENCCHK

Name	**FOL ENCCHK**
Descriptor	Enable encoder and motor step comparison
Type	Setup
Initial value	NO
Range	FOL ENCCHK * * * *
Default	FOL ENCCHK YES NO * YES
Syntax	FOL LEAD 0.82 * Q1 *
Options	TAB YES NULL NO
	F1 F2 F3 F4 F5 F6

Description

The **FOL ENCCHK** statement enables or disables the comparison between the commanded position in motor steps, i.e. controlled in motor step mode, and the actual position in encoder steps. This is especially useful when the slave is a Compumotor servo drive such as the Z drive or the Dynaserv. Its first major purpose is to assist in the tuning of the drive for minimum following error. It also facilitates rapid electronic response to excess following error. For a complete discussion of the purposes and uses of this feature, please refer to the section titled *Motor and Encoder Comparison*.

See Also: **FOL LEAD**, **FOL DIRSET**

FOL CAM

Name	**FOL CAM**
Descriptor	Enable cam profiling
Type	Programming
Initial value	NO
Range	FOL CAM * * * *
Default	FOL CAM YES NO * YES
Syntax	FOL LEAD 0.82 * Q1 *
Options	TAB YES NULL NO
	F1 F2 F3 F4 F5 F6

Description

The **FOL CAM** statement enables or disables the cam profiling mode of following motion. Cam profiling is used to delete and create pre-defined motion profiles. These profiles may then be executed at very high cycle rates. When cam profiling is enabled, motion definition and intitiation statements have changed meaning. Master cycle parameters also change. For a complete discussion of the purposes and uses of this feature, please refer to the section titled *Cam Profiling*.

See Also: **FOL MAS_CYC**, **FOL CYC_OFF**, **FOL NEWCYC**

FOLM

Name	**FOLM**
Descriptor	Moving Positioning System Parameters
Type	Set-Up
Default	N/A
Syntax	FOLM
Options	TAB DEF RATIO PDEF ENABLE
	F1 F2 F3 F4 F5 F6

Description

FOLM statements define the moving positioning system (MPS). The MPS allows point-to-point, contouring, ratio following, and many other types of moves to be super-imposed on the slave following the master. All of these moves are programmed as if the slave was standing still, making implementation of the MPS very easy. Below is a summary of the moving positioning system statements.

FOLM PDEF	Define slave's initial position when MPS is defined.
FOLM RATIO	Establish the ratio of slave to master required to keep the slave stationary with respect to the master.
FOLM DEF	Define the MPS. This involves setting the master cycle count and position to 0 and the slave position to that specified with the FOLM PDEF statement.
FOLM ENABLE	Enable the moving positioning system. Subsequent slave moves are now with respect to the MPS coordinate system.

See Also: **FOL**, **UNIT MASTER**

FOLM PDEF

Name	**FOLM PDEF**
Descriptor	Moving Positioning Offset
Type	Set-Up
Initial value	Ø
Range	±99999999 slave steps
Default	FOLM PDEF * * * *
Syntax	FOLM PDEF 0 Q7 12.35 *
Options	TAB Q NULL
	F1 F2 F3 F4 F5 F6

Description

The **FOLM PDEF** statement is used to define the slave's initial position when the moving positioning system is defined. This parameter is entered in user units, and is scaled by the **UNIT POS** parameter. This number will be used each time the MPS is defined, so it must be defined before the MPS is defined for the first time. Numeric Q variables can be entered for this statement, and the default value for **FOLM PDEF** is 0.

The Moving Positioning System in the 4000 allows programming of slave moves with respect to moving reference. Point-to-point, contouring, even ratio following moves can be performed by the slave while it's following the master. For more information on the moving positioning system, refer to the example under the **FOLM ENABLE** statement.

See Also: **FOLM RATIO**, **FOLM DEF**, **FOLM ENABLE**, **UNIT POS**

FOLM RATIO

Name	**FOLM RATIO**
Descriptor	Moving Positioning Ratio
Type	Set-Up
Initial value	Ø
Range	±127 slave steps per master step
Default	FOLM RATIO * * * *
Syntax	FOLM RATIO 1:1 -.5:1 Q2:Q9 *
Options	TAB Q NULL :
	F1 F2 F3 F4 F5 F6

Description

The **FOLM RATIO** statement establishes the ratio of slave to master required to keep the slave *stationary* in the moving positioning system. Assume **FOLM RATIO** is set to .5:.3 for an axis. The first parameter is

scaled by the **UNIT POS** value to give slave steps. The second parameter is scaled by the **UNIT MASTER** value to give master steps. For a **UNIT POS** of 25000 and a **UNIT MASTER** of 4000, the slave to master step ratio would be .5*25000 to .3*4000, or 125 slave steps for every 12 master steps.

If no second parameter is specified, it is assumed to be 1. Numeric Q variables can be used with this statement for slave and/or master parameters. The first, or slave, parameter is a signed number, which will be negative if the slave counts in the opposite direction from the master. This statement must be executed before the moving positioning system is defined by the **FOLM DEF** statement. For more information on the moving positioning system, refer to the *Moving Positioning System* section earlier in this chapter, and the example under the **FOLM ENABLE** statement.

See Also: **FOLM PDEF**, **FOLM DEF**, **FOLM ENABLE**, **UNIT MASTER**, **UNIT POS**

FOLM DEF

Name	**FOLM DEF**
Descriptor	Define Moving Positioning System
Type	Programming
Initial value	NO
Range	TRIG 1-4
Default	FOLM DEF * * * *
Syntax	FOLM DEF TRIG1 TRIG3 IMMED *
Options	TAB TRIG NULL IMMED NO
	F1 **F2** **F3** **F4** **F5** **F6**

Description

The **FOLM DEF** statement defines the moving positioning system (MPS). When the MPS is defined the 4000 establishes the position reference for master and slave with respect to the MPS. At the time of MPS definition, the slave position within the MPS is set to that defined with the **FOLM PDEF** statement. If the application requires that the master cycle position also be set to zero at the time of the MPS definition, the **FOL NEWCYC** statement should be used. The same trigger could be used to initiate both MPS definition and a new master cycle.

If **IMMED** is specified, the MPS is defined immediately. If a trigger is specified, the 4000 defines the moving positioning system when the trigger occurs. Program flow is not affected in either case. If the application requires that the program be halted until a specific master cycle position or the trigger occurs, the **FOL WAIT** and **FOL MOVEWT** statements may be used.

Just because the MPS is defined, it does not mean that subsequent slave moves will be with reference to the MPS. The **FOLM ENABLE** statement allows reference to the stationary or moving positioning systems. For more information on the moving positioning system, refer to the *Moving Positioning System* section earlier in this chapter, and the example under the **FOLM ENABLE** statement.

See Also: **FOLM PDEF**, **FOLM RATIO**, **FOLM ENABLE**, **IN FOL TRIG**

FOLM ENABLE

Name	**FOLM ENABLE**
Descriptor	Enter or Exit Moving Positioning System
Type	Programming
Initial value	NO
Default	FOLM ENABLE * * * *
Syntax	FOLM ENABLE YES NO * YES
Options	TAB YES NULL NO
	F1 F2 F3 F4 F5 F6

Description

The **FOLM ENABLE** statement is used to enter and exit the moving positioning system (MPS). The MPS must have been previously defined when this command is executed, and executing this command does not affect the master and slave position coordinates already established at the time the **FOLM DEF** command was executed. Even if the moving positioning system is not entered right after definition, the master and slave positions are constantly kept track of in ***both*** positioning systems. This ensures that no position information is lost when transferring from the stationary positioning system to the moving positioning system and back.

Entering and exiting the MPS simply means taking the point of view of the moving object or stationary reference, respectively. It does not affect the shaft speed of the slave. If an application requires that the slave start tracking the master immediately upon entry into the MPS (i.e., remain stationary in the MPS), a **MOVE STOP** or **MOVE** to a position should be commanded right after the MPS is defined and entered.

If the slave is in absolute mode, move commands issued while in the MPS will be performed with respect to the moving absolute position, and those executed while not in the MPS will be performed with respect to the stationary absolute position. The 4000 keeps track of both absolute coordinate systems. The moving positioning system can be illustrated more clearly with the example below.

See Also: **FOLM PDEF**, **FOLM RATIO**, **FOLM DEF**

Example

In the example below, a pattern of coordinates needs to be traced on a grid tray which is located on a moving conveyor belt. The part is moved with an XY stage, where the X axis will be following the speed of the conveyor. The third axis of the 4000 is controlling the conveyor, so the X axis will be following the step output of axis #3. Assume that the third motor is mounted to a pulley of 3" diameter driving the conveyor. The X and Y motors, 4000 axes 1 and 2, respectively, are coupled to 4 pitch leadscrews and are driven with drives of 25,000 step/rev resolution.

The XY stage will initially be homed and the absolute position set to 0. The operator enters the required pattern number, and a subroutine is called from another program location (not shown here) which determines the required values for Q variables used in the MOVE commands. The conveyor is set in motion and the trays start moving down the line.

When a tray is sensed, the moving positioning system is defined and entered immediately. When the system is defined, the X position is defined to be at 23.4". The trays are 20" in length and the home position of the X axis is 3.4" from the sensor which detects the leading edge of the tray. Setting our initial X coordinate to 23.4" means that the far edge of the tray is at position 0. This assumption was used in the subroutine **SET_VARS** where the coordinate location variables are set. After the MPS is entered by the X axis, the XY stage makes three linearly interpolated moves to the proper tray locations, turning on an actuator at each location. The X axis then exits the MPS and the XY stage returns to the home position to await the next tray.

Statement	Description
`UNIT POS 100000 100000 25000 *`	`'X and Y positioning scale 'factors for inches, third axis 'motor resolution`
`UNIT VEL 100000 100000 1326.291 *`	`'Velocity scale factors for 'inches/sec`
`UNIT ACCEL 100000 100000 1326.291 *`	`'Acceleration scale factors for 'inches/sec/sec`
`VEL 5 5 12.25 *`	`'Velocity`
`ACCEL 10 10 35 *`	`'Acceleration`
`UNIT PATH VEL 100000`	`'Unit path velocity for LINT 'mode move`
`UNIT PATH ACCEL 100000`	`'Unit path acceleration for 'LINT mode move`
`PATH VEL 5`	`'Path velocity`
`PATH ACCEL 10`	`'Path acceleration`
`LINT MODE YES YES * *`	`'Enable LINT mode on axes #1 'and 2`
`UNIT MASTER 1326.2912 * * *`	`'Master scale factor for inches`
`FOL MASTER MOT3 * * *`	`'Master for axis #1 is step 'output of axis #3`
`FOLM RATIO 1:1 * * *`	`'MPS slave to master ratio is 1 'to 1`
`FOLM PDEF 23.4 * * *`	`'Define initial slave position 'to 23.4"`
`FOLM MAS_CYC 35 * * *`	`'Master cycle length is 35 'inches (well larger than the 'slave offset + tray length)`
`MOVE HOMECW HOMECCW * *`	`'Move XY stage to home`
`MODE M_ABS M_ABS * *`	`'Absolute mode`
`PDEF 0 0 * *`	`'Define home position as 0`
`IN Q1 = LCD1,1 ^ENTER PATTERN ^`	`'Operator enters required grid 'pattern`
`GOSUB SET_VARS`	`'Subroutine that contains the 'grid coordinates (i.e., values 'of Q2 - Q7) for the possible 'patterns`
`MOVE * * SLEWCW *`	`'Start conveyor moving`
`LABEL PLACE_PT`	`'Label to repeat part placement 'on tray`
`FOL NEWCYC TRIG3 * * *`	`'Set master position to 0 on 'trigger #3`
`FOLM DEF TRIG3 * * *`	`'Define MPS on trigger #3`
`FOL WAIT TRIG3 * * *`	`'Suspend program operation 'until 1st tray is sensed`
`FOLM ENABLE YES * * *`	`'Enter MPS on axis #1, does not 'start motor`
`MOVE STOP * * *`	`'Must stop the X axis within 'the MPS, since when the MPS 'was entered, we are at rest in 'the stationary reference 'frame, but actually in motion 'in the MPS reference frame - a 'LINT mode move can not be 'started while an axis is in 'motion.`
`MOVE Q2 Q3 * *`	`'Do LINT mode move to 1st grid 'coordinates`
`OUT POB3 = 1`	`'Actuate part placement device`
`WAIT FOR .2 SECONDS`	`'Wait for placement`
`OUT POB3 = 0`	`'Turn actuator back off`
`MOVE Q4 Q5 * *`	`'Do LINT mode move to 2nd grid 'coordinates`
`OUT POB3 = 1`	`'Actuate part placement device`
`WAIT FOR .2 SECONDS`	`'Wait for placement`
`OUT POB3 = 0`	`'Turn actuator back off`
`MOVE Q6 Q7 * *`	`'Do LINT mode move to last grid 'coordinates`
`OUT POB3 = 1`	`'Actuate part placement device`

```
WAIT FOR .2 SECONDS                    'Wait for placement
OUT POB3 = 0                           'Turn actuator back off
FOLM ENABLE NO * * *                   'Exit the MPS, does not stop
                                       'motor
MOVE 0 0 * *                           'Move XY stage back to home
                                       'position, motor now at rest
GOTO PLACE_PT                          'Repeat the cycle
DONE                                   'End of program
```

UNIT MASTER

Name	**UNIT MASTER**
Descriptor	Set Master Unit Scale Factor
Type	Set-Up
Initial value	25000
Default	UNIT MASTER * * * *
Syntax	UNIT MASTER 4000 Q23 * 25000
Options	TAB Q NULL

F1	F2	F3	F4	F5	F6

Description

The **UNIT MASTER** statement, used with the Following option, allows master positions to be programmed in user units. In most cases, the user units for master and slave will be the same since they refer to the same physical system, (i.e., inches or revolutions) but this is not required.

The number in the first column of the **UNIT MASTER** statement does not necessarily represent the steps/unit for the encoder input for axis #1. Instead, it represents the steps/unit for whichever encoder input or step output will be the master for the slave on axis 1. This is shown in the example below.

Fractional values are allowed, but truncation errors may occur if the product of the scale factor and the value scaled is not a whole number. Numeric Q variables can also be used as the scale factor, and the default value of **UNIT MASTER** is 1. The **UNIT MASTER** parameter scales the following statements: master parameter of **FOL RATIO**, **FOL MDIST**, **FOL MAS_CYC**, **FOL WAIT**, **FOL MOVEWT**, **FOL WIN_P**, **FOL WIN_W**, **FOL M_SYNC**, **FOL S_SYNC**, **IN Qn = FOL AXISn MAS_P** and master parameter of **FOLM RATIO**. After scaling, the result must not be larger than 99,999,999 or an execution error will result.

See Also: **FOL**, **FOLM**

Example

In the example below, axis 1 is following the encoder input on axis #3. Unit scale factors, master and slaves axes, and the following ratio are set.

Statement	Description
`UNIT POS 25000 * * *`	`'Sets slave scale factor to 25000 for axis 1`
`UNIT MASTER 4000 * * *`	`'Sets master scale factor to 4000 for axis 1`
`FOL MASTER ENC3 * * *`	`'Axis 1 using encoder input #3 as master`

`FOL RATIO 1:1 * * *`	`'Set the slave to master ratio at 1 to 1`
	`'based on user units. Actual ratio in steps`
	`'= 1*25000 to 1*4000 or 6.25 slave steps for`
	`'each master step.`

IN FOL

Name	**IN FOL**
Descriptor	Get Following Axis Information
Type	Programming
Initial value	N/A
Range	N/A
Default	IN Qn = FOL AXISn MAS_P
Syntax	IN Q10 = FOL AXIS4 MAS_P IN Q1 = FOL AXIS2 TRIG1

Options						
	TAB	Q	MAS_P	MAS_C	SLV_P	ETC
	TAB	SHIFT	SYN_ERR	FOL_ERR	TRIG	ETC
	TAB	S_SYNC	M_SYNC	MAS_V		ETC
	F1	**F2**	**F3**	**F4**	**F5**	**F6**

Description

MAS_P The **IN Qn = FOL AXIS MAS_P** statement assigns to the Q variable the current master cycle position for the following axis specified. It is important to remember that this is not a position of that slave, but instead it is the position of that slave's master. The position returned is the position of the master within its current cycle. For a complete discussion of master cycles, please refer to the section titled *The Master Cycle Concept* earlier in this chapter. The master cycle position in master steps is inversely scaled by **UNIT MASTER** for the axis, so the resulting value in the variable is the cycle position expressed in the user's units. This value may be used for subsequent decision making, or simply recording the cycle position corresponding to some other event.

See Also: **FOL MAS_CYC, FOL NEWCYC, FOL CYC_OFF, IN Qn=FOL AXISn MAS_C**

MAS_C The **IN Qn = FOL AXIS MAS_C** statement assigns to the Q variable the current master cycle number for the following axis specified. It is important to remember that this is not a position of the master (or the slave), but instead it is the current cycle number. The master cycle number is set to zero when a new cycle is defined, and is incremented each time a master cycle finishes, i.e., rollover occurs. For a complete discussion of master cycles, please refer to the section titled *The Master Cycle Concept* earlier in this chapter. The master cycle number is not a unit of position, and has no scaling factor. The reported master cycle number is the number of completed master cycles since the last new cycle definition. It will often correspond to the number of complete parts in a production run. This value may be used for subsequent decision making, or simply recording the cycle number corresponding to some other event.

See Also: **FOL MAS_CYC, FOL NEWCYC, FOL CYC_OFF, IN Qn=FOL AXISn MAS_P**

SLV_P The **IN Qn = FOL AXIS SLV_P** statement assigns to the Q variable the current slave absolute position for the following axis specified. The position returned will depend on the positioning mode (motor or encoder) of the slave and whether or not the slave is currently in the Moving Positioning System (**FOLM ENABLE YES**). Please refer to the section titled *Moving Positioning System* earlier in this chapter. If the slave is not currently in the Moving Positioning System, the reported position will be identical to that reported with the **IN POS MABS** or **EABS** statement, for motor and encoder positioning respectively. If the slave *is* currently in the Moving Positioning System, the reported position will be the absolute motor or encoder position with respect to the moving positioning system zero reference, for motor and encoder positioning respectively. In other words, this statement may be used to obtain the position of the slave with respect to a moving object or a stationary reference. The position in slave steps is inversely scaled by **UNIT POS** for the axis, so the resulting value in the variable is the slave position expressed in the user's units. This value may be used for subsequent decision making, or simply recording the slave position corresponding to some other event.

See Also: **FOLM DEF, FOLM ENABLE, FOLM PDEF**

SHIFT The **IN Qn = FOL AXIS SHIFT** statement assigns to the Q variable the current value of the net, or absolute slave shift which has occurred at the current ratio for the following axis specified. This ratio may zero, i.e., the slave is at rest, or its current constant ratio reached as a result of a **MOVE SLEWCW** or **MOVE SLEWCCW**. The position returned will be the sum of all shifts performed on that axis since that axis reached constant ratio. Each time a new commanded constant ratio is reached, and at the end of preset distance move, the shift value is set to zero. The reported position will be the net motor or encoder shift for motor and encoder positioning respectively. The shift value in slave steps is inversely scaled by **UNIT POS** for the axis, so the resulting value in the variable is the slave shift expressed in the user's units. This value may be used for subsequent decision making, or simply recording the slave's net shift corresponding to some other event.
See Also: **FOL SHIFT**

FOL_ERR The **IN Qn = FOL AXIS FOL_ERR** statement assigns to the Q variable the current slave following error for the following axis specified. The following error is defined as the difference between the setpoint position and the actual position. For a complete discussion of the conditions which may result in following error, please refer to the section titled *Following Performance* earlier in this chapter. The error value returned will be the motor or encoder position error for motor and encoder positioning respectively. The error in slave steps is inversely scaled by **UNIT POS** for the axis, so the resulting value in the variable is the slave following error expressed in the user's units. This value may be used for subsequent decision making, or simply recording the slave following error corresponding to some other event.
See Also: **FOL MAXACC, FOL MAXVEL**

SYNC_ERR The **IN Qn = FOL AXIS SYNC_ERR** statement assigns to the Q variable the current slave synchronization error for the following axis specified. The 4000 allows the user to define two external events, or "marks", which capture the slave position. These are called *Master Sync Mark* and *Slave Sync Mark*, and are defined with the **FOL M_SYNC** and **FOL S_SYNC** statements respectively. Each time either a master or slave sync mark occurs, the corresponding slave position is captured and saved internally. The **FOL SYNC_OFF** statement defines the expected difference between these captured slave positions. This expected difference is called the *Slave Synchronization Offset.* The difference between the actual offset and the expected offset is called the Sync Error. This error may be read into a Q variable using the **IN Qn = FOL AXISn SYNC_ERR** statement. To understand exactly how to use this, please refer to the section titled *Periodic Master/Slave Synchronization* earlier in this chapter.
See Also: **FOL MSYNC, FOL SSYNC, FOL SYNC_OFF**

TRIGn This is useful in determining whether or not the trigger input has occurred, the first 4 bits in this table will be set when the trigger is defined for a function, then cleared when the trigger occurs. The latter two will be set if the trigger defines the corresponding sync mark, and will remain set until that sync mark is re-defined or defined with a NO parameter. Suppose for example that the trigger had been defined to start a new master cycle and define a master sync mark. Before the trigger occurs, the value assigned to the Q variable will be 17. After the trigger occurs, the value will be 16.

Defines Moving Positioning System	1
Defines new master cycle start	2
FOL WAIT on this trigger	4
FOL MOVEWT on this trigger	8
Defined as master sync mark	16
Defined as slave sync mark	32

This is useful in determining whether or not the trigger input has occurred, the first four bits in this table will be set when the trigger is defined for a function, then cleared when the trigger occurs. The latter two will be set if the trigger defines the corresponding sync mark, and will remain set until that sync mark is redefined or defined with a NO parameter. Suppose for example that the trigger had been defined to start a new master cycle and define a master sync mark. Before the trigger occurs, the value assigned to the Q variable will be 17. After the trigger occurs the value will be 16.
This trigger status request may also be useful by allowing one program to determine if another is waiting on a **FOL WAIT TRIGn** statement when both are running under multi-tasking. The example below shows how to check for only this condition.

```
IN Q1 = FOL AXIS2 TRIG1       'check axis 2's TRIG1 status
MATH Q1 = Q1 AND 4            'check only the FOL WAIT bit
IF Q1 = 4 GOTO IT_WAITS       'if the value is 4, the other
                              'task is waiting
```

S_SYNC The **IN Qn = FOL AXISn S_SYNC** statement assigns the Q variable the slave position most recently latched by the slave synchronization mark defined with the **FOL S_SYNC** statement. Each time the mark occurs, the slave position is captured and may be read with this statement. The position in slave steps is inversely scaled by **UNIT POS** for the axis, so the resulting value in the variable is the captured slave position expressed in the user's units. This value may then be used for whatever purpose is required, even if no master synchronization mark has been defined. These uses may include functions related to Master/Slave Synchronization, or simply general purpose data aquisition. For a complete discussion of Master/Slave Synchronization, refer to *Master/Slave Synchronization.*

M_SYNC The **IN Qn = FOL AXISn M_SYNC** statement assigns the Q variable the slave position most recently latched by the master synchronization mark defined with the **FOL M_SYNC** statement. Each time the mark occurs, the slave position is captured and may be read with this statement. The position in slave steps is inversely scaled by **UNIT POS** for the axis, so the resulting value in the variable is the captured slave position expressed in the user's units. This value may then be used for whatever purpose is required, even if no slave synchronization mark has been defined. These uses may include functions related to Master/Slave Synchronization, or simply general purpose data aquisition. For a complete discussion of Master/Slave Synchronization, refer to *Master/Slave Synchronization.*

MAS_V The **IN Qn = FOL AXISn MAS_V** statement assigns the Q variable the current master velocity. The velocity in master steps per second is inversely scaled by **UNIT MASTER** for the axis, so the resulting value in the variable is the master velocity expressed in the user's units. This value may then be used for whatever purpose is required, such as estimating a process cycle rate. The value returned is only approximate, and its accuracy it dependent on the value chosen for **FOL SMOOTH**. For a complete discussion of the effect of velocity smoothing on velocity measurement accuracy , refer to *Velocity Smoothing.*

ON FOL_ERR

Name	**ON FOL_ERR**
Descriptor	Interrupt on Excess Slave Following Error
Type	Programming
Initial value	N/A
Range	N/A
Default	ON FOL_ERR ANY_AXIS GOTO LABELØ
Syntax	ON FOL_ERR AXIS2 GOTO F_ERR2

Options	TAB TAB	DISABLE BEG	ALPHA END	GOSUB GOTO	FND_LBL	ETC ETC
	F1	**F2**	**F3**	**F4**	**F5**	**F6**

Description

The **ON FOL_ERR** statement enables the 4000 to continuously check for excess following error on the specified axis or all axes. For a complete discussion of the conditions which may result in following error, please refer to the section titled *Following Performance* earlier in this chapter. Whenever the slave's setpoint position is not equal to the actual slave position, a following error exists. The user may specify a following error tolerance using the **FOL PTOL** statement. If the magnitude of the actual following error ever exceeds the specified tolerance, the 4000 latches the condition of *following error tolerance exceeded*, and the 4000 will branch to the location specified in the **ON FOL_ERR** statement. If the user's program requires that the 4000 respond to a new occurrence of excess following error, the **FOL PTOL** statement should first be executed to clear the old error, and then the **ON FOL_ERR** statement executed to allow detection of the condition. For a complete discussion of related topics, please refer to the section titled *Monitoring Following Error* earlier in this chapter.

See Also: **FOL PTOL**, **FOL WIN_P**, **FOL WIN_W**, **IN Qn = FOL AXISn FOL_ERR**

ON WT_ERR

Name	**ON WT_ERR**
Descriptor	Interrupt on FOL MOVEWT Position Missed
Type	Programming
Initial value	N/A
Range	N/A
Default	ON WT_ERR ANY_AXIS GOTO LABELØ
Syntax	ON WT_ERR AXIS2 GOTO WT_QUIT

Options	TAB TAB	DISABLE BEG	ALPHA END	GOSUB GOTO	FND_LBL	ETC ETC
	F1	**F2**	**F3**	**F4**	**F5**	**F6**

Description

The **ON WT_ERR** statement enables the 4000 to continuously check for a **WT_ERR** condition (wait error) on the specified axis or all axes. If a **FOL MOVEWT** has specified a master cycle position as the wait condition for a subsequent move, and the master position has already exceeded that specified cycle position by the time the subsequent move is commanded, the **WT_ERR** condition is flagged and latched. For a complete understanding of wait errors, it is important to understand master cycles and waiting for cycle positions. Please refer to the sections titled *The Master Cycle Concept* and *Following Wait Statements* earlier in this chapter. When the **WT_ERR** condition is detected, the 4000 will branch to the location specified in the **ON WT_ERR** statement. If the user's program requires that the 4000 respond to a new occurrence of wait error, the **FOL MOVEWT** statement should first be executed to clear the old error, and then the **ON FOL_ERR** statement executed to allow detection of the condition. Although it is valid, it is not necessary to specify another wait condition in this case. A **FOL MOVEWT NO** statement may be used to clear the error.

See Also: **FOL PTOL**, **FOL WIN_P**, **ON FOL_ERR**, **IN Qn = FOL AXISn FOL_ERR**

DEFINE TRIGDB

Name	**DEFINE TRIGDB**
Descriptor	Define trigger debounce time
Type	Set-Up
Initial value	80 milliseconds
Range	4 - 1000 milliseconds
Default	DEFINE TRIGDB * * * *
Syntax	DEFINE TRIGDB 80 * Q1

Options	TAB	Q	NULL			
	F1	**F2**	**F3**	**F4**	**F5**	**F6**

Description

The **DEFINE TRIGDB** statement defines the total debounce time for the four trigger inputs. The debounce prevents noise or the mechanical switch bounce from causing a false interrupt on the trigger input. A trigger is initially recognized on the rising edge of the input. That trigger will not be recognized again until it has gone low and the debounce time, measured from the rising edge, has been exceeded. This debounce time affects registration and all of the **FOL** and **FOLM** statements that include triggers as a parameter. The initial value of 80 ms. will usually be long enough to debounce most mechanical and electronic switches, but this time may be lengthened if needed. In some applications, registration marks or master/slave synchronization marks may occur more frequently than 80 ms. In these cases, the debounce time may be shortened, provided the signal *bounce* is short enough. The debounce times are only accurate to ± 2 ms of the specified value, and the actual values used will always be between 4 and 1000 ms. The debounce times are specified for triggers 1, 2, 3, and 4 (left to right on the statement line).

See Also: **FOL MOVEWT**, **FOL NEWCYC**, **FOL WAIT**, **FOL M_SYNC**, **FOL S_SYNC**, **FOLM DEF**

Error Codes

The following is a list of error codes unique to the Model 4000.

Error Messages	Description
Big RAM not installed in U14, U15	This occurs if any **FOL** or **FOLM** statement or any **IN Qn** request for a following parameter is executed without the RAM required for the following option installed.
Invalid **FOL MASTER** specified	This indicates that an illegal master was specified in **FOL MASTER**. A slave may never use its own motor step count as its master. A slave in encoder step mode or with stall detect enabled may not use its own encoder step count as master.
FOL MASTER invalid if moving or **FOL ENABLE** invalid if moving	This indicates the statement is not allowed while the slave is moving. *Moving* means moving with respect to the current positioning system. A slave may be stationary with respect to a stationary reference, yet be *moving* in the moving positioning system. **FOL MASTER** or **FOL ENABLE** respectively
FOL MASTER not executed	This indicates that no **FOL MASTER** for the axis is currently specified. It will occur if any **FOL** or **FOLM** statement defining or enabling parameters or any **IN Qn** request for a following parameter is executed and no **FOL MASTER** statement was executed, or **FOL MASTER NO** was executed.
FOLM DEF not completed	This indicates that the statement is not allowed if no moving positioning system is defined. It could occur if **FOLM DEF** was never executed, or if the trigger which defines the moving positioning system has not occurred. **FOLM ENABLE YES**
FOL parameter too large	This indicates that the numeric parameter supplied with the statement is too large. **FOL MAS_CYC**—Error if: master steps>99999999 **FOL WIN_P**—Error if: master steps>99999999 **FOL WIN_W**—Error if: master steps>99999999 **FOL CYC_OFF**—Error if: master steps>99999999 or <-99999999 **FOL PDEF**—Error if: slave steps>99999999 or <-99999999 **FOL SYNC_OFF**—Error if: slave steps>99999999 or <-99999999 **FOL PTOL**—Error if: slave steps>99999999 **FOL MOVEWT**—Error if: master steps >99999999 or <-99999999 **FOL WAIT**—Error if: master steps >99999999 or <-99999999 **FOL WAIT**—Error if: master steps >99999999 or <-99999999 **FOL M_SYNC**—Error if: master steps>99999999 **FOL S_SYNC**—Error if: master steps>99999999
FOL parameter not valid	This indicates that the parameter supplied with the statement is not valid. **FOL MAS_CYC** —Error if: master steps are negative **FOL RATIO** —Error if: ratio denominator is negative **FOLM RATIO**—Error if: ratio denominator is negative **FOL MAXACC**—Error if: Error if value is Ø **FOL SMOOTH** —Error if: smooth number is not 1-4 **FOL WIN_P**—Error if: master steps are negative **FOL WIN_W** —Error if: master steps are negative **FOL PTOL** —Error if: slave steps are negative **FOL M_SYNC** —Error if: slave steps are negative **FOL S_SYNC** —Error if: slave steps are negative
Master cycle definition pending	This indicates a master cycle definition is pending a trigger, the master cycle position is unknown. **IN Qn AXISn MAS_P** **IN Qn AXISn MAS_C**
FOL SHIFT cannot start move	This indicates that a command phase shift cannot be performed. **FOL SHIFT #** - Error is already shifting or performing other time based move or **VEL** or **ACCEL** is zero or distance is >99999999 or <-99999999 **FOL SHIFT CW,CCW** - Error if **ACCEL** is zero
Master sync mark undefined	This indicates that no master sync mark definition exists. This may be because the **FOL M_SYNC** statement was never executed, or was executed with **NO** as the parameter. **IN Qn AXISn SYN_ERR**
Slave sync mark undefined	Indicates that no slave sync mark definition exists. This may be because the **FOL S_SYNC** statement was never executed, or was executed with **NO** as the parameter. **IN Qn AXISn SYN_ERR**
FOL RATIO value invalid	This indicates that the ratio given after scaling by **UNIT POS** and **UNIT MASTER** was outside the range ±127, or that the ratio denominator was zero **FOL RATIO** **FOLM RATIO**

I N D E X

T

V

W