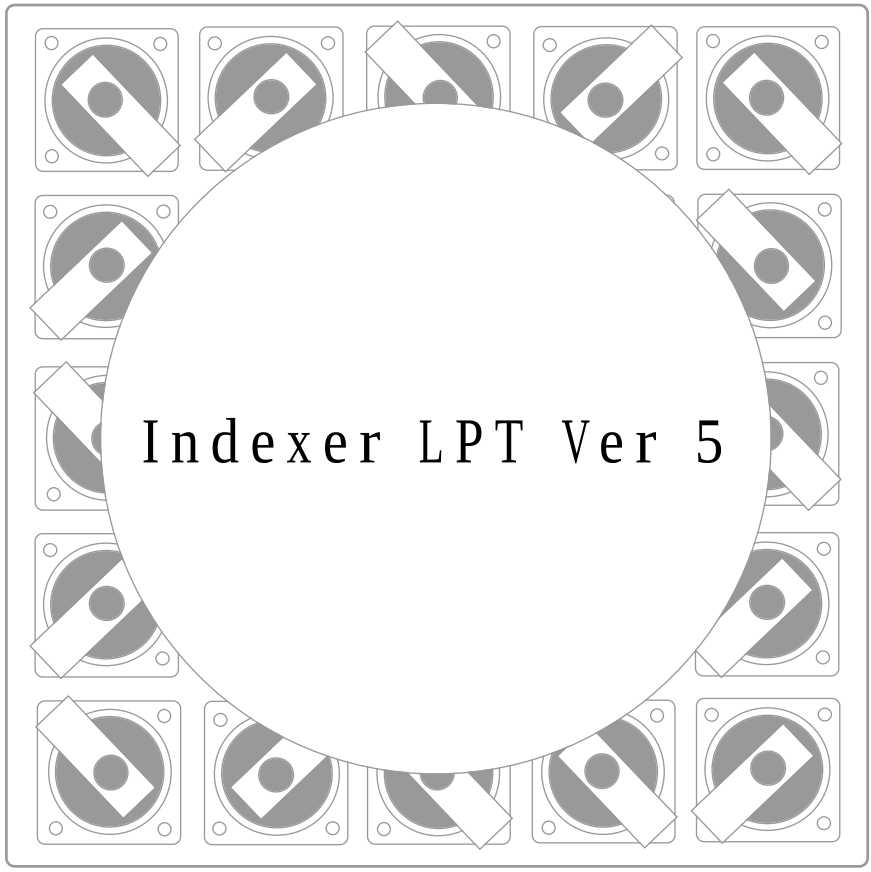
Indexer LPT Ver 5



A B I L I T Y S Y S T E M S

1422 ARNOLD AVE ROSLYN PA 19001 (215) 657-4338 FAX (215) 657-7815

http://www.abilitysystems.com motion@abilitysystems.com

128-51 R2

Indexer LPT ™ Manual (Version 5)

Ability Systems Corporation makes no warranty, except as specifically provided in the **Program License Agreement**, either expressed or implied, including but not limited to any implied warranties of merchantability or fitness for a particular purpose, regarding these materials and makes such materials available solely on an "as is" basis.

WARNING

DO NOT USE IN LIFE SUPPORT SYSTEMS

READ AND UNDERSTAND THE TERMS OF THE PROGRAM LICENSE AGREEMENT LOCATED ON THE DISKETTE PACKAGE. SEND THE MATERIALS BACK TO THE PLACE OF PURCHASE FOR A REFUND IF YOU DO NOT AGREE.

UNDERSTAND THE HAZARDS OF MOTION CONTROL SYSTEMS BEFORE USING THIS SOFTWARE!

By breaking the seal of the disk package or by using these materials you agree to be bound by the conditions of the Program License Agreement.

As per the program License Agreement, Ability Systems Corporation disclaims liability to anyone for special, collateral, incidental, or consequential damages in connection with or arising out of use of these materials. The sole and exclusive liability to Ability Systems Corporation, regardless of the form of action, shall be limited to the purchase price of the materials giving rise to claim.

The entire risk as to the performance of these materials is with the purchaser. Ability Systems Corporation assumes no responsibility of any kind for errors in the package or for the consequences of such errors. This allocation of risk is reflected in the purchase price of the product.

Ability Systems Corporation reserves the right to make changes to and improve its products as it sees fit

Chapter 11

COMMANDS

Chapter 1

INTRODUCTION

What Indexer LPT is

Thank you for choosing the **Indexer LPT** motion control software sub-system.

Indexer LPT software allows the hardware contained in an ordinary IBM compatible printer port to assume the role of a multi-axis stepper motor indexer. When used with commercially available translator drivers, **Indexer LPT** provides an easy to use and inexpensive means of motion control.

A module that is provided with the software affixes to a printer port and can be wired to implement a sophisticated set of controls, including joystick jog, feed rate override and feed hold. Each additional printer port provides sufficient input and output to control two axes of motion. **Indexer LPT** supports up to four printer ports, and can control up to seven axes of motion.

Signals for each axis comprise TTL level outputs controlling “step”, “direction”, “reduced current”, and “all windings off”. Two limit switches per axis may be wired directly to the printer connector. One auxiliary TTL level input per axis is provided to allow for additional system sensing. Limit switch closures automatically arrest motion.

The **Indexer LPT** software loads under Windows as a character type kernel mode device driver. Commands comprise plain English type strings of ASCII characters which are written to the “motor” device in the same manner as writing to a text file. Consequently,

Indexer LPT may be used with a variety of different application languages. **Indexer LPT** also communicates back to the user program. Messages can be read in the same manner as a file. Simple motion systems can be implemented even from batch files executed from a DOS box using the DOS *copy* command. Since **Indexer LPT** is as accessible as a text file, sophisticated machine controllers can be easily implemented by using application programs written by Ability Systems or others. Each axis may be controlled with respect to such things as position, home position, maximum and minimum velocity, acceleration, and automatic return to home. Query commands allow the user program to read the status of limit switches, read the auxiliary input lines, read position relative to home, and read back setup parameters. Advanced commands include a variety of powerful features including linear interpolation in up to seven simultaneous axes, circular interpolation, helical interpolation, vector velocity control, smooth look-ahead contouring, feed rate override, and feed hold..

DANGER

PERSONAL INJURY OR DEATH AND/OR EQUIPMENT DAMAGE MAY OCCUR FROM ELECTRICAL SHOCK OR FROM PHYSICALLY MOVING OBJECTS.

MOTOR DRIVE SYSTEMS SHOULD BE INSTALLED BY QUALIFIED PERSONNEL FAMILIAR WITH THE CONSTRUCTION AND OPERATION OF ALL EQUIPMENT IN THE SYSTEM AS WELL AS ASSOCIATED HAZARDS.

Before You Begin

Read the program license agreement provided. Use of this software comprises your agreement to the terms and conditions of the license agreement.

If you do not agree to the terms and conditions of the license agreement, do not remove the distribution media from its sealed container. If you return all materials provided to the place of purchase in their original condition, you are entitled to a refund for the purchase price less shipping and handling.

Fill in and return the registration card. This entitles you to thirty days of free customer support and entitles you to user privileges such as application notes and discounts on subsequent versions.

Chapter 2

MOTION BASICS

Step-Direction Control

The stepper motor itself may rotate in either direction in a sequence of discrete steps. It is the job of the “translator” or “driver” electronics to excite the windings of the stepper motor in a manner such that this motion is accomplished.

The unit which commands the translator when to “step” and in which direction is called the *indexer*. The indexer communicates this information to the translator by means of two logic signal lines typically called *step* and *direction*.

The Translator

The logic level of the *direction* line dictates whether the next step to be taken will be in the clockwise or counterclockwise direction. When a single pulse is issued from the indexer on the “step” line, the translator responds by manipulating the power to the stepper motor windings such that the motor moves a single increment in the direction specified by the *direction* line.

It can be seen, therefore, that the translator is basically a low level state machine and power driver. It responds immediately to the commands of the indexer to change the state of the power on the windings of the stepper motor such that the prescribed motion is accomplished. The angular resolution of each step is determined by the type of translator and stepper motor which is being used.

The Indexer

The indexer, on the other hand, is a high level controller which issues pulses at rates determined by parameters such as acceleration, starting velocity, maximum velocity, and extent of motion. The indexer receives commands from the user written program which prescribes a given end result. For example, the indexer may receive a command which indicates that the stepper motor is to be moved 2000 steps in the clockwise direction. The indexer responds by setting the voltage on the direction signal line to a level which effects clockwise rotation. Then the indexer issues pulses in such a manner that the motor accelerates from rest and decelerates to a controlled stop at a position corresponding to exactly 2000 pulses.

Additional Translator Features

Some translators allow additional control over the power to the stepper motor. Due to the nature of its design, the stepper motor draws the most current when it is not in motion. In an effort to conserve power, some translators provide a *reduced current* logical input line. As the name implies, this signal is used to limit the current to the motor windings, thereby allowing the user program to run the motor with the most amount of power only when it is most needed. (It should be noted that reducing current also reduces holding torque, an unsuitable condition in some applications).

In some applications it is desirable to remove the power to the stepper motor entirely. Some translators accommodate this application by providing a logic input, usually entitled *all windings off*. When this signal is driven to the prescribed state, all power is removed from the stepper motor windings.

Indexer LPT software uses the TTL level outputs of the parallel printer port to provide signals for *step, direction, reduced current*, and *all windings off*. Consequently, the computer itself assumes the role of the indexer in that the computer applies these signals from the printer adapter to the translator directly.

As was mentioned, only the *step* and *direction* signals are required for basic motion control. Since control is offered over the *reduced current* and *all windings off* signals in a manner which is independent of the motion commands, the designer may choose to use these signals as general purpose outputs.

Limit Switches

Indexers often contain the capability to arrest motion by means of limit switch inputs.

Two basic purposes for limit switches

The first purpose of limit switches is to define the operational boundaries of the object which is under position control. When actuated, the “high” limit switch arrests motion in the positive (+) direction (you may, for example, define clockwise shaft rotation as the “positive” direction), and the “low” limit switch arrests motion in the negative (-) direction. The user program may intentionally move the controlled object into the limit switches to determine its operational range. Since each limit switch only arrests motion in one direction, the user program can sense a limit switch closure and subsequently move the controlled object in the other direction. Used in this manner, the application program can use the limit switches as a means of determining physical boundaries.

Indexer LPT does not accommodate the second basic purpose of limit switches, which is to limit the over-travel of the object under position control as a safety measure to prevent potential damage or injury. In this case, limit switches are used to abruptly arrest motion, or even disconnect the power source, depending upon the design of the equipment.

WARNING

The limit switch functions provided by Indexer LPT are ONLY to be used for the first purpose of establishing operational boundaries and NOT to be used for the second purpose of safety or over-travel switches.

The exact implementation of safety over-travel limit switches is highly dependent on the design of the equipment.

In some designs four limit switches are used for a single axis. In an example, two “inner” limit switches, such as the ones used with **Indexer LPT**, are used to delimit the boundaries of normal operation. Two “outer” limit switches are used as safety switches to arrest motion in the event that the axis, for whatever reason, exceeds its normal boundaries and presents a potentially damaging or dangerous condition if allowed to proceed.

In some applications the “outer” limit safety switches are wired such that when actuated they remove the power from the motor, either by means of the translator *all windings off* input, or by means of a relay which removes all power.

In other applications, releasing the hold of a motor by removing power may cause an axis to break free because of gravity or spring tension and cause other damage. In such a case it may be wiser to use the safety switches in a manner which forces the motor to hold its present position.

In still other applications limit safety switches alone provide insuf-

ficient protection, and other measures of mechanical design and configuration such as protective enclosures, slip clutch couplings, lockouts, etc., must be used to assure the safety of the operation.

It is important to note that IN TIME ALL SWITCHES WILL FAIL. When working with dangerous or potentially damaging forces, the entire system must be designed with safety in mind. Do not rely on switches alone to avert a disaster.

Moving the Motor

Every motion control application has a unique set of dynamic characteristics. For example, the position of an object may be controlled by means of a stepper motor and a lead screw. In this example some major forces which the motor must overcome are friction, stiction, and inertia.

Friction comprises the forces introduced on adjacent physical members due to the drag produced along the contacting surface. Stiction is an initial value of friction which must be overcome to begin motion from rest. Finally, inertia is proportional to the mass being moved and is the force which resists change in velocity (resists acceleration).

Stepper motor systems may require a certain amount of resistive forces in order to function properly or to function at all. A stepper motor may appear not to be operational if motion control is attempted with a free running (unloaded) shaft. Resistance may be applied by attaching the stepper motor shaft to an appropriately sized and loaded pulley, flywheel, or lead screw.

Most stepper motor applications involve moving an object from one position to another by accelerating from rest and decelerating to a controlled stop at the predetermined position. Depending upon the extent of motion, the motion may begin with a period of acceleration, then proceed at a constant velocity before decelerating to a controlled stop.

Stepper motor velocity is expressed in units of steps per second. Change in stepper motor velocity, which is acceleration, is expressed in units of steps per second per second.

Stepper motors must begin motion at an initial starting velocity. The appropriate initial velocity is governed by the physics of the particular application and is often determined experimentally. **Indexer LPT** stores the value of the initial velocity for each axis in the *lowspeed* registers. The user may change the contents of the

lowspeed registers by means of the *set_lowspeed* command.

The value of the acceleration which is applied to each axis is stored in the *acceleration* registers. The user may change the contents of the *acceleration* registers by means of the *set_accel* command.

In an extended motion, the maximum speed at which the motor will step is determined by the value of the *highspeed* registers. The contents of the *highspeed* registers may be changed using the *set_highspeed* command. The highest value which may be placed in the *highspeed* registers is governed by the speed of the computer and may be determined using the *max_speed?* command. Refer to the chapter, **Special Considerations**, for some additional details on using the *max_speed?* command.

Indexer LPT accomplishes acceleration controlled motion by means of the *move*, *feed*, and *arc* commands. Constant velocity motion is also performed by means of the *jog* command.

Position Tracking

When **Indexer LPT** is first invoked, position tracking on every axis is disabled. Once the *set_home* command is issued on an axis, **Indexer LPT** will automatically track the accumulated steps which are sent to the axis. The number of pulses which effect positive rotation are added to the accumulated value and the number of pulses which effect negative rotation are subtracted from the accumulated value. This accumulated value represents the position of the axis relative to a reference "position" established by the *set_home* command. **Indexer LPT** tracks position to a magnitude of 2,147,483,647 steps in either direction.

After each motion command, or after the *position?* command, **Indexer LPT** automatically reports the position of the associated axis. If the axis has not been initialized by the *set_home* command, **Indexer LPT** reports "**unknown position**".

Controlled and Uncontrolled Limit Stopping

When motion is terminated by means of a limit switch closure, position tracking is lost. This comprises an "uncontrolled limit stop". **Indexer LPT** automatically reports the limit switch interruption, and all subsequent position queries for the affected axes receive an "`unknown position`" response. If subsequent position tracking is desired, it must be re-initialized by means of the *set_home* command.

Indexer LPT preserves position tracking when a constant velocity motion effected by the *jog* command is interrupted by a limit switch closure. This comprises a "controlled limit stop". It is important to note that **Indexer LPT** position tracking only relates to the control

signals which are sent to the translator. Consequently, if the holding force of the stepper motor is overcome by excessive system inertia, the position tracking information returned by **Indexer LPT** will not correspond to the actual position of the motor. This can be avoided by correctly matching the motor and translator to the requirements of the application. Acceptable inertial forces should be maintained by adjusting the *jog* velocity using the *set_jogspeed* command. This velocity is often determined experimentally.

Unless interrupted by limit switch closure, or a abort from a feed hold procedure, motion commands normally terminate by reporting the new position of the associated axis (axes). If interruption to motion occurs, it is also reported. Position tracking information is always available by means of the *position?* query command.

A useful application of controlled limit stopping is to use the *jog* command in automatically determining the useful range of motion between limit switches. In this case, the *jog* command may be used to move the axis into the limit switches in both directions. The user program can determine the distance using the *position?* query.

Simultaneous Motion

Up to six axes can be moved simultaneously using either the *move* command or the *feed* command. Both of these commands use a "best fit" straight line strategy to accomplish linear interpolation.

In multiple axis moves, the axis which is to be moved the greatest amount of steps is called the "dominant" axis.

The *move* command affords the type of control necessary to accomplish the most rapid traversal. Using the *move* command, the motion parameters set up in the *lowspeed*, *highspeed*, and *acceleration* registers associated with the dominant axis govern the velocity profile of the dominant axis. The other axes are controlled as necessary to traverse the best fit linear path to the destination.

The *feed* command is used in cases where the velocity along the path of traversal, known as the "vector" velocity, must be held constant. Under the *feed* command, the motion parameters set up in the *feed_lowspeed*, *feed_highspeed*, and *feed_accel* registers govern the velocity profile of the dominant axis. The dominant axis will begin motion at the velocity set up in the *feed_lowspeed* register and accelerate according to the rate set up in the *feed_accel* register. The final velocity (which would be governed by the *highspeed* register for the move command), however, is scaled down such that the vector velocity specified in the *feed_highspeed* register is attained. **Indexer LPT** appropriately scales the final velocity for one, two, and three dimensional feeds. When more than three axes are moved simultaneously, three dimensional scaling is used. Similar to the *move* command, the axes are controlled as necessary

to traverse the best fit linear path to the destination.

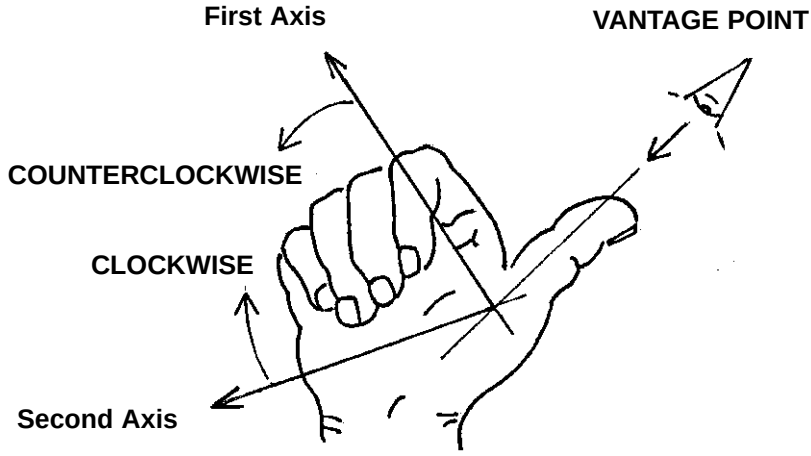
Circles and Arcs

Basic Theory

Circular interpolation is accomplished by means of the *circle*, *arc_to_angle*, and *arc_to_point* commands. Circles can be traversed in two manners: clockwise, and counterclockwise. The manner of traversal is determined by the first command argument, "`CW`" or "`CCW`", and the order in which the associated axes are called in the command line.

In three dimensional space, clockwise rotation from one perspective appears as counterclockwise rotation from another perspective. For example, from the perspective of the driver of an automobile, the tires on the left side of the car are turning clockwise and the tires on the right are turning counterclockwise. All of the tires, however, are turning in the same direction. To specify rotation without ambiguity, **Indexer LPT** uses a "right hand rule" to distinguish circular motion. As such, **Indexer LPT** uses the order in which the axes are called in the command argument to distinguish the perspective in which the circular motion is viewed. Assuming perpendicular axes, when the first axis called is "wrapped" into the second axis using fingers of the the right hand, the direction to which the thumb points determines the direction from which the circular path is viewed.

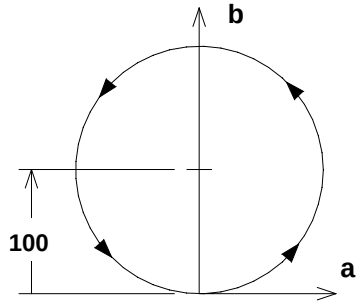
The manner of circular traversal between any two perpendicular axes can be visualized as follows: Place the knuckle of the little finger of your right hand on a point on the first axis in the positive direction such that you are able to place the tip of that finger on a



point on the second axis in the positive direction. Your little finger is now pointing in the counterclockwise direction.

As an example, assume a linear translation stage is superimposed upon this page of text such that movement from left to right extends the “a” axis in the positive direction and movement from bottom to top extends the “b” axis in the positive direction. The following command is issued:

```
circle:ccw,a,0,b,100
```

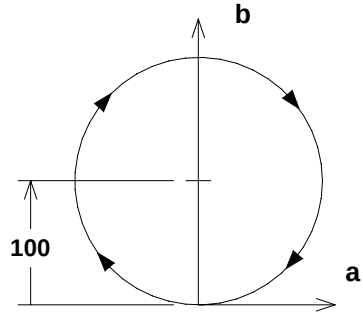


Vantage point is front of page

This command draws a circle whose center point is zero (0) steps from the present position in the “a” direction and one hundred (100) steps from the present position in the “b” direction. To determine the manner in which the circle is traversed, assume the lower left corner of the page is the origin. The bottom edge of the page, therefore, represents points on the “a” axis in the positive direction. The left edge of the page represents points on the “b” axis in the positive direction. Place the knuckle of the little finger of your right hand on the bottom of the page such that the tip of that finger touches the left edge. The direction to which your little finger points defines the counterclockwise direction.

Now consider the command:

```
circle:ccw,b,100,a,0
```



Vantage point is back of page

This command also draws a circle whose center point is zero (0) steps from the present position in the “a” direction and one hundred (100) steps from the present position in the the “b” direction. (The same center point as in the previous example). Since the “b” axis is the first axis called in the command, you must place the knuckle of the little finger of your right hand on the left edge of the paper, and your finger tip on the bottom edge. You must reach your hand around to the back of the page. The direction which your little finger points defines the counterclockwise direction. (From the perspective of the front of the page, however, the motion appears to be clockwise).

Two commands are available for drawing arcs: *arc_to_angle*, and

arc_to_point.

The *arc_to_angle* command requires that the user specify the distance from the present position to the center point of the arc, the direction of traversal (clockwise or counterclockwise), and the angle which is to be subtended. **Indexer LPT** calculates the destination position and traverses the arc, stopping at that position.

In applications sensitive to accumulated error due to numerical round off, it is desirable to specify the destination point exactly. Numerical control machines using “g codes” use this method. **Indexer LPT** accommodates this type of control using the *arc_to_point* command. The *arc_to_point* command requires the direction of traversal (clockwise or counterclockwise), the center point of the arc, and the end point of the arc. **Indexer LPT** first calculates the radius of the arc based upon the distance from the present position to the center point. The extent of the arc is governed by a radial projection from the destination point to the center point. **Indexer LPT** subtends the arc to the angle delimited by this projection. If there is a difference between the destination point on the arc at the completed angle and the destination point specified in the command line, **Indexer LPT** makes the best straight line path from the point on the arc to the destination point specified in the command line.

Circles and arcs are traversed at vector rates governed by the rules of motion defined under the *feed* command.

After executing a *circle*, *arc_to_angle*, or *arc_to_point* command **Indexer LPT** reports the destination position of each axis in the mailbox in the order in which the axes are called in the command line.

Segmentation

Indexer LPT constructs circles and arcs by means of a succession of linear interpolations, similar in function to *feed* commands. In order to accelerate into, and traverse the arc in a smooth, continuous motion, these feed commands are automatically executed from the look-ahead *queue* buffer (for more information about the look-ahead buffer, refer to the chapter entitle “Queue Processing”).

The method of approximating an arc by means of line segments is called “segmentation”. Each linear segment represents the chord of the angle over the arc which is subtended. The angle which the segment is laid is called the “chordal angle”. The maximum dimensional error which is introduced, called the “chordal”, or “segment” error can be calculated using the following formula:

```
Segment Error = Radius X (1 - cos(chordal-angle/2) )
```

The default chordal angle is five degrees In the default configuration, therefore, a complete circle would be approximated using sev-

enty two (72) segments.

In order to control segment error, you may use the *set_arcseg_degrees* command to set the chordal angle.

In an example, using the *circle* command, a twenty inch circle is subtended. Using the default chordal angle of five degrees, the maximum dimensional error is:

```
Segment Error = 10 x ( 1-cos(5/2) ) = .0095″
```

Using the following command, the chordal approximation angle is changed to one degree:

```
set_arcseg_degrees:1
```

The maximum error due to segmentation is reduced, shown by the following calculation:

```
Segment Error = 10 x (1-cos(1/2) ) = .00033″
```

With a chordal angle of one degree, a complete circle would be approximated with 360 linear segments. These segments are automatically loaded and executed from the look-ahead buffer. Once the motors have completed their cycle, the memory occupied in the buffer is cleared, and free for use by subsequent commands.

Since all circular interpolation utilizes the look-ahead buffer, if you attempt to execute a command which uses more memory than you have available for this purpose (memory is reserved using the *set_q_mem* command), then the command will fail and the following message will appear in the mailbox:

```
error,queue full
```

Each linear segment of a circular interpolation uses the amount of queue memory as a single *feed* command. The amount of memory that any interpolation requires is determined by the number of segments that it uses. For example, assume that the value of the *set_arcseg_degrees* register is 5, and the following command is issued:

```
arc_to_angle:ccw,a,0,b,1000,17
```

This command subtends an arc angle of seventeen degrees. Four segments are used to approximate the arc. Please note that the accuracy of the angle is NOT affected by the *set_arcseg_degrees* register. A seventeen degree arc is subtended using four linear segments, three which are included in five degree chords, and one in the remaining two degree chord. The amount of memory required in the queue buffer to execute this command is equivalent to four *feed* commands.

The amount of segments used in a circular interpolation can be very

large, especially considering the large angles which may be required in helical interpolations. You can determine the amount of queue memory a command will use by means of the *command_mem?* query. You can determine how much queue memory is available using the *q_mem?* query.

Calculation Dwell

In operations where continuous motion is required between linear and circular interpolations, **Indexer LPT** allows the circular interpolation commands to be used in the look-ahead buffer. For this type of operation, please refer to the chapter entitled "Queue Processing".

In cases where circular interpolation commands are not loaded into the queue buffer, but executed immediately, the amount of time is that is required for mathematical calculation before the motors begin motion is called "calculation dwell".The amount of calculation dwell is proportional to the number of segments that are being used to subtend the arc.

For most arcs, which subtend angles 360^{o} or less, calculation dwell is very small. However, consider the following command:

```
arc_to_angle:ccw,a,0,b,1000,36000
```

This command subtends an arc of 36000^{o} (one hundred complete circles). If a value of 1 is used in the *set_arcseg_degrees* register, this motion will be accomplished using the equivalent of 36000 *feed* commands.

More importantly, this command entails substantial processing, and consequently incurs substantial calculation dwell. This is to say, the machine may appear dormant for a enough time so that when motion commences it may catch a machine operator unaware. This comprises a safety consideration. Please, then, consider the following warning notice carefully:

WARNING

When designing a machine that can cause damage and/or injury due to potentially unexpected motion after long periods of calculation dwell, it is the responsibility of the machine designer to implement appropriate lock-outs, warning annotations, or other mechanisms to alert the operator that machine motion is impending.

One method may be to use a digital output to light a warning display, indicating to the operator that the machine is in operation, and that motion is impending. Other methods may be necessary depending on the nature of the machine.

Helical Interpolation

Helical interpolation is an operation by which two axes traverse a circular path, while one or more additional axes advance proportionately to the angle subtended in the path of the circle or arc. In a machine where a stylus is controlled along three orthogonal axes, a helical interpolation would cause the stylus to follow the path of a helix, similar in shape to a coil spring, or screw thread.

Indexer LPT has the capability to perform helical interpolation in up to four simultaneous axes. By definition, two of these axes are used to subtend the circular path. The other two axes advance proportionately to the subtended angle. In some machine designs the fourth axis can be used, for example, to keep a rotary axis tangent to the path of the helix.

The linear components of the commands that accomplish helical interpolation are added as optional arguments to the circular interpolation commands. For example, to subtend a complete circle with a radius of 1000 in the a and b axis, while advancing the c axis 300 steps, the following command is issued:

```
circle:ccw,a,0,b,1000,c,300
```

All motion starts and finishes together. The position of the *c* axis will track a path proportional to the angle subtended by the combined motion of the “a” and “b” axes, and come to a controlled stop when the circle is complete.

In a similar example, the “d” axis is also moved:

```
circle:ccw,a,0,b,1000,c,300,d,1450
```

In this example, both the c and the d axis track a path proportional to the angle subtended by the a and b axes.

Similar to the circle command, the *arc_to_angle* and the *arc_to_point* commands accommodate optional linear axes for helical interpolation.

The Feed Hold Feature

The *feed hold* feature allows a motion process to be suspended when the *feed hold* input is activated.

As long as the *feed hold* input is activated, the motion process remains suspended. When the *feed hold* input is de-activated, the motion resumes to completion. If, however, the *abort* input is activated while a motion process is suspended, the motion process will not be completed and will be terminated immediately. The *abort* input has no effect if the *feed hold* input is not activated. If a motion is aborted this message will appear in the mailbox:

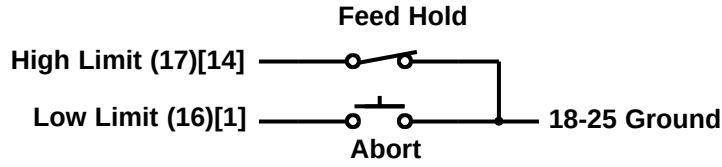
`abort`

Motor movements which involve acceleration, such as are effected by the *move* and *feed* commands, will decelerate to a controlled stop when the *feed hold* input is activated. Instantaneous movements, such as are effected by the *jog* command, will stop immediately.

Limit switch inputs for a designated axis are used for the *feed hold* and *abort* feature. By designating an axis to be used for *feed hold* inputs, the *high limit switch* and *low limit switch* inputs assume the role of *feed hold* input and *abort* input, respectively.

The *high limit switch* input on the designated axis controls the *feed hold* feature as follows: If the high limit switch input is TTL low, (such as when it is connected to ground), motion commands are permitted to proceed normally. Opening the connection causes the input to be pulled up internally to TTL high, activating the *feed hold* feature.

X-Group Pin Number - [ ]
Y-Group Pin Number - ( )



Typical Feed Hold and Abort Switch Wiring

The low limit switch input on the designated axis controls the *abort* feature as follows. If the low limit switch input is open (TTL high), the *abort* feature is disabled. Closing the connection to ground (TTL low) activates the *abort* feature.

In a simplified set up, a *feed hold* switch may be a normally closed switch circuit to ground and an *abort* switch may be a normally open switch circuit to ground.

In order for the *feed hold* feature to function, the “mode” of the axis that is being used must be in either “mode 3” or “mode 1”. The **Y Group** axis of the port where the Hardware Assist Module (**HAM**) resides is automatically assigned by **Indexer LPT** as “mode 3”. The default mode of all other axes is “mode 0”. The “mode 3” axis is recommended.

If the axis that you intend to use for the *feed hold* feature is not

"mode 3", then the axis must be switched to "mode 1" using the *axis* command.

Once the axis is either "mode 3" or set to "mode 1", the axis can be designated to provide *feed hold* inputs using the *feedhold_input* command. Several axes may be converted to "mode 1" using the *axis* command. Only one axis can be designated to be used for *feed hold* inputs.

In an example, assume that the high limit switch input on the "d" axis is to be used to accommodate the *feed hold* input. In this case, the low limit switch input on the "d" axis accommodates the *abort* input. To set up the *feed hold* feature in this manner, first convert the axis from the "motor control" mode to the "digital output" mode using the following command:

```
axis:d,1
```

Now that the axis has been set to this mode (and cannot be used for motor control), it can be designated to be used for *feed hold* inputs by issuing the following command:

```
feedhold_input:d,1
```

WARNING

Observe the following precautions when using the feed hold feature.

1) Do not use the feed hold feature as a safety stop.

2) The feedhold feature must be initialized in order to function.

3) Personal injury or equipment damage subject to failure of the feed hold feature should be averted by appropriate safety features designed into the system hardware.

4) System hardware design should not allow for inadvertent release of the feed hold feature, and consequent dangerous unexpected motion.

Misuse As a Safety Switch

Safety and emergency stop features should not be software dependent. Safety and emergency stop features should be built into the system hardware design. **THERE IS NO SUBSTITUTE FOR A RAPIDLY ACCESSIBLE, HARD WIRED, EMERGENCY STOP SYSTEM.**

Failure to Initialize

It is the responsibility of your software to initialize the *feed hold* feature. It is possible to begin a sequence of motion operations without initializing the *feed hold* feature. In a poorly designed system, the operator may expect the *feed hold* feature to function, only

to discover, at the most inopportune time, that it has not been initialized.

A simple software safeguard is to check the initialization of the *feed hold* feature before each motion sequence using the *feedhold_input?* command. This, however, is not a complete solution.

Again it should be emphasized:

THERE IS NO SUBSTITUTE FOR A RAPIDLY ACCESSIBLE, HARD WIRED, EMERGENCY STOP SYSTEM.

Switch Failure

The *feed hold* feature will fail to work if not initialized properly. It will also fail to work if the *feed hold* circuit fails in a short circuit mode. In either case, such failure may not be apparent to the operator until the feature is actually used. **Use a properly designed emergency stop system and an appropriately trained operator to avoid catastrophe.**

Inadvertent Release

As was mentioned, a potentially dangerous condition exists when an operator expects motion to be interrupted by the *feed hold* feature, and, due to some malfunction, the motion continues. A potentially more dangerous condition may exist if the machine is at rest due to the *feed hold* feature, and the operator inadvertently releases the *feed hold* feature, causing unexpected motion. The solution to this problem depends on the specific application. The best solution may involve a hard wired indicator. In one example, a double pole switch (or relay) is used to activate the *feed hold* input. One pole is used for the *feed hold* input to the computer. The other pole is used to turn on a visible and/or audible warning indicator which alerts the operator that the machine is dormant because of the *feed hold* feature. In another example, a “drop out” relay is used to activate the *feed hold* feature. The operator is required to press two physically separated pushbutton switches, occupying both hands, before the machine is allowed to resume motion.

Chapter 3

DEVICE DRIVER BASICS

What a device driver is

A “device driver” is a program which is generally used for the purpose of providing a universal interface, that is, a standard means of communication between a user program and system hardware.

This particular device driver communicates with the user program in ASCII text by means of the operating system’s file control operations. Consequently, application languages which use Windows or DOS under Windows to communicate ASCII text to and from files can be used to control stepper motors using **Indexer LPT** without the need for low level hardware control routines or libraries.

How a device driver is loaded

The file interface portion of **Indexer LPT** is loaded under the direction of the System Registry when the Windows GUI (Graphical User Interface) loads after the bootstrap sequence.

Indexer LPT is a character type of device driver

Indexer LPT software operates as a character device. Character devices are used in applications which communicate by means of a sequence of characters. Character devices assume “device” names similar to file names. The device name which is used to communicate with the **Indexer LPT** software is **motor**.

Once a character device driver is loaded, it can be accessed by

name in a similar manner as a file would be accessed. Consequently, almost any software which has the capability to control communication with a file can access a character device driver.

Accessing character devices from DOS

Indexer LPT can be accessed from a DOS box running under Windows. Please note that since **Indexer LPT** loads with the Windows GUI, it cannot be accessed from "DOS Compatibility Mode", since to get into this mode Windows unloads the GUI portion of the system. If your application requires only DOS and cannot use Windows, then the 16 bit **Indexer LPT** version may be the most appropriate for your use. (Contact Ability Systems for the availability of 16 bit **Indexer LPT**).

As an example of DOS access to **Indexer LPT**, a stream of characters comprising a sequence of commands can be sent to the motor device using the DOS "copy" command. Simply use an ASCII text editor, such as NOTEPAD or DOS's EDIT, to make a file containing **Indexer LPT** commands, one command per line. (The file in this example must contain no extraneous non printing characters such as tabs or spaces. The last command must be followed by a carriage return). Suppose we call this file **MYFILE.CMD**. The "motor" device may be issued the commands written in this file by typing from the DOS prompt:

```
C>copy myfile.cmd motor<Enter>
```

Indexer LPT places an answer to the last command which it receives in a memory area which we call the "mailbox". The contents of the mailbox can be read just as a file is read.

For example, a user desires to see what is in the mailbox of the "motor" device just after the **Indexer LPT** system has loaded and before any other commands have been issued. The following is typed from the DOS prompt:

```
C>type motor<Enter>
```

The following message appears on the screen:

```
C>installation successful
```

Accessing character devices from software

Application software typically accesses files and devices by opening a file number or a "handle" to the file using the "file open" function provided with the language. Reads and writes to and from the opened file are accomplished in the manner specified in the associated language programming manual. Specific examples of reading and writing to the **Indexer LPT** "motor" device in several different languages are given in the examples in the chapter entitled **Getting Started**.

Chapter 4

HARDWARE REQUIREMENTS

The Parallel Port

Parallel Ports

Indexer LPT software converts the parallel printer adapter hardware, called the “parallel port”, from its normal use as a printer control to a special use as an indexer and industrial controller. IBM compatible computers can support multiple parallel ports. Each parallel port is generally presented to the outside world as a twenty five pin D Subminiature female connector. Each port is distinguished from other ports by its’ Input/Output (I/O) memory address. This is referred to in the **Resources** section of the Windows **Device Manager** as the **Input/Output Range**, and describes the I/O memory addresses that are accessible on the port. The lower number of the range is called the port’s “base address”.

Port Types

The original configuration of the IBM Parallel Port Adapter is commonly referred to in the industry as “Standard”. However, it is sometimes referred to “Normal”, “ISA” or “AT”. **Indexer LPT** supports this “Standard” configuration.

Since the original inception of the IBM parallel interface, variations of the parallel port have emerged using differing signal input and output configurations. One such variation is known as “Bidirectional”, or BPP. Another is known as the “Enhanced Parallel Port”, or EPP. BPP and EPP configurations are not supported by **Indexer LPT**.

As of this writing the latest parallel port standard is known as the Enhanced Capabilities Port, or "ECP". The ECP standard is rigidly defined by the Institute of Electronics Engineers under the IEEE 1284-1994 specification. Some adapter card manufacturers specify that their product conforms to the IEEE-1284 specification. Others may simply refer to it as ECP. **Indexer LPT** supports the IEEE-1284, or ECP, specification.

On-Board Parallel Port

Most computers include a single integrated parallel port. The port type and base address are generally configurable in the computer's BIOS Setup, which is usually accessible when the computer is powered up.

We do not recommend using this port for varying reasons. The most significant reason is that electrostatic damage to this port cannot be easily repaired. Also, it has been our experience that many computer manufacturers do not rigidly comply with industry specifications for the parallel port. Nevertheless, some installations use this port as a convenient location to install the **Hardware Assist Module** (HAM).

If you decide to use the on board port for the **HAM**, use the computer's BIOS Setup to configure that port to either "Standard" or ECP. If **Indexer LPT** does not recognize the **HAM** when it loads, the port is either damaged or incompatible. You will therefore need to locate the **HAM** on a connector to a parallel port adapter card.

ISA Parallel Port Adapter Cards

The original (and third party compatible) parallel port for IBM compatible computers is rugged, easy to configure and inexpensive. These adapter cards are generally designed for the ISA type bus connector, and can be purchased to accommodate one or two ports. Port base addresses are configured by means of jumper connectors located on the card. When assigning a base address using jumpers, it is important to make sure that you do not assign a base address that is being used elswhere in the system.

Some ISA type parallel port adapter cards allow you to select port types by means of jumpers. For these cards you should select "Standard" or ECP. Do not select BPP, EPP, or EPP/ECP.

PCi Parallel Port Adapter Cards

Parallel adapter cards designed for use in the PCi bus that support the ECP specification are available from numerous manufacturers. These cards are configured by means of manufacturer supplied software, and do not have jumper selections. Some special considerations will be given in this manual on using these cards, but not before we describe what an **Indexer LPT** "axis" is.

What an Axis Is

Symmetrical Set of Signals

Most of the **Indexer LPT** commands and queries reference a letter designator that we call an “axis”. In the industry, the word “axis” is used to refer to the control over a single motor, since a motor is often used to robotically effect motion along or around a geometric axis. In this manual we refer to the letter designation that we call an “axis” as a group of input and output signals generally associated with the control over a single motor. These signals consist of the output signals for *step*, *direction*, *all windings off*, and *reduced current*, as well as the input signals for high and low limit switches, and an *auxiliary input* line.

Although we call this group of signals an “axis”, some or all of the signals can be used for purposes other than motor control. For example, the output signals entitled *all windings off* and *reduced current* are most often used simply as general purpose digital outputs.

The “axis” letter designator is used specify the group of signals, the port on which they reside, and their location on that port. The “axis” letter designation correlates an **Indexer LPT** command to the physical locations of the connector pins to which it applies.

Each Printer Port Controls Two (2) Axes

Each printer port contains two symmetrical sets of axis signals. **Indexer LPT** commands access each axis by name. The axis names are “a”, “b”, “c”, “d”, “e”, “f”, “g” and “h”. The physical location of the axis is determined by the base address of the printer adapter card. The association between axis name and card base address is shown in **Figure 4-1**.

In an example, consider a computer is configured for motion control using a single printer port located at a base address of 378(hex). The following command:

```
move:c,5500
```

would generate 5500 pulses on pin 2 (the *step* signal) while holding pin 3 (the *direction* signal) at a high logic level.

Printer Adapter Addressing

Standard Addresses

IBM compatible computers accommodate up to three “standard” printer ports at address locations 3BC(hex), 378(hex), or 278(hex).

Non-Standard (PCi) Addresses

Parallel adapter boards that are designed for use on the PCi type bus are automatically automatically assigned Input/Output addresses by the operating system. In some cases the automatically assigned addresses are “Standard” addresses. In other cases they may not be.

The base address of a parallel port installed on a PCi type card can be determined from the Windows **Device Manager**. You can launch the **Device Manager** by double snapping over the **System** icon in the **Control Panel**, and by then snapping over the **Device Manager** tab. To find the base address of an installed parallel port, locate the port under the **Ports** section of the **Device Manager**, right snap over the particular LPT port icon, and select **Properties** from the fold down menu that appears. Under the **Resources** tab of the **Properties** dialog, locate a field entitled **Input/Output Range**. The lower of the associated numbers is the base address of the port.

If you are unable to install the PCi ports that you are using at standard addresses, or if your application requires more than six axes, then you must use of non-standard addressing. If you cannot locate instructions on accommodating non-standard addresses in the README.TXT file of your distribution disk, contact Ability Systems for detailed instructions.

Line Printers

Cautions Against Inadvertent Writes

Inadvertent writes to a printer port which is used for motor control may cause unexpected and erratic motor movement, and/or destroy Indexer LPT setups, requiring re-starting of the system.

It is therefor not advisable to use a printer, or software set up to use a printer, on computers dedicated to motion control.

However, on certain systems where the nature of motion control does not present a safety hazard, some simple precautions are all that is necessary. Printer output is normally directed to LPT1. If you must have a printer attached to your system, use LPT1 for the printer, and the remaining parallel ports for motion control. Make sure that your system (operating system printer setups) and software does not direct its output to parallel ports other than LPT1, and make sure that motor controls are powered down whenever you are printing.

Translator Wiring

It is important to realize that the output signals from the printer port are TTL technology and voltage level. Numerous translators are available which accept TTL type signals. Consequently, with

Indexer LPT it is possible to configure a motion control system with nothing more than simple point to point wiring between the parallel interface and the translators.

It is important to check the signal levels and polarity of the translator that you are using. For example, most translators accept negative pulses for the step input, and require that the direction input is set up and stable during the duration of the pulse. This is the manner in which **Indexer LPT** functions. However, if your translator accepts a positive pulses, it may appear as if it is functioning properly, but may lose step during a change in motor direction. Some translators have a jumper, or selector switch to specify positive or negative pulses. Make sure your translator is set up to accept negative pulses.

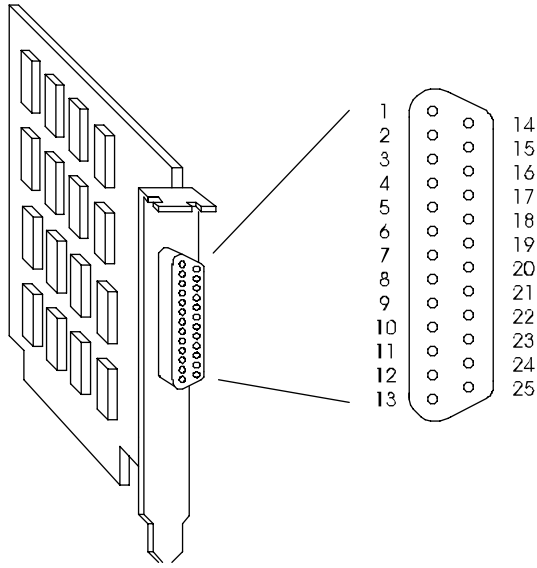
Some translators, such as the Gecko drive illustrated in Figure 4-3 and the Pacific Scientific 6410 in Figure 4-7, are optically coupled. Since TTL outputs reliably sink, but do not source current, these drives require a small 5 Volt power supply to provide drive current for the opto-couplers. When using a power supply in this manner, to assure maximum noise immunity make sure that the +5 Volts from the power supply does not exceed the open circuit voltage of the parallel port.

Limit Switch Wiring

In a standard IBM compatible printer adapter, the connections which **Indexer LPT** uses as limit switch inputs are pulled up internally to 5.0 volts through a resistance of 4.7K to 10K ohms. Grounding these connections by means of an external switch comprises a limit switch closure. Examples of how limit switches can be wired are given in Figures 4-2 through 4-7.

Aux Input Wiring

Each axis has an associated *auxiliary input* signal line. Unlike the limit switch inputs, the internal connection to this input has no pull up resistor, and therefore must be driven by a TTL level signal in order to function reliably. The *auxiliary input* is a convenient means of digital input.



		CARD BASE ADDRESS					
		278 (hex)		378 (hex)		3BC (hex)	
AXIS		a	b	c	d	e	f
Reference Group		x	y	x	y	x	y
FUNCTION	Step	2	6	2	6	2	6
	Direction	3	7	3	7	3	7
	Reduced Current	4	8	4	8	4	8
	All Windings Off	5	9	5	9	5	9
	Low (-) Limit Switch	1	16	1	16	1	16
	High (+) Limit Switch	14	17	14	17	14	17
	Auxiliary Input	13	12	13	12	13	12
	Signal Ground	18-25		18-25		18-25	
		PIN NUMBERS					

Figure 4-1

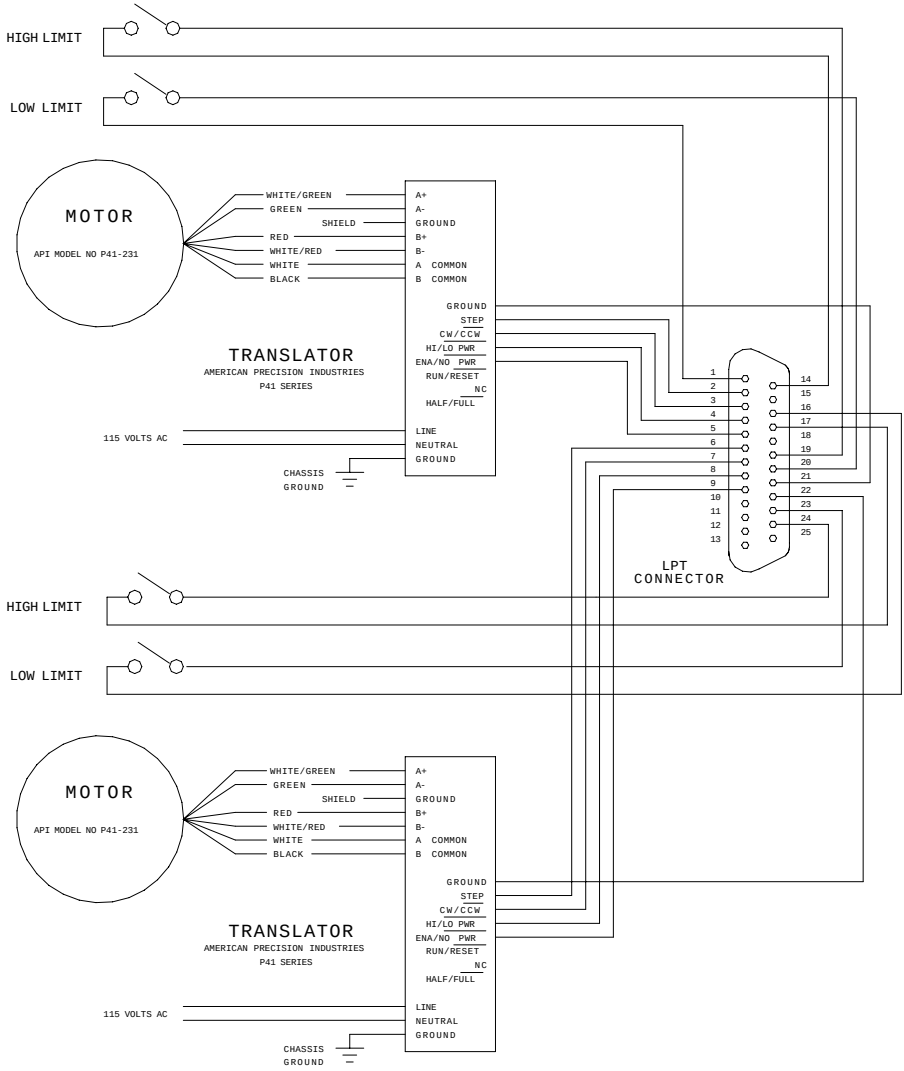


Figure 4-2

WIRING DIAGRAM - API Series Translator

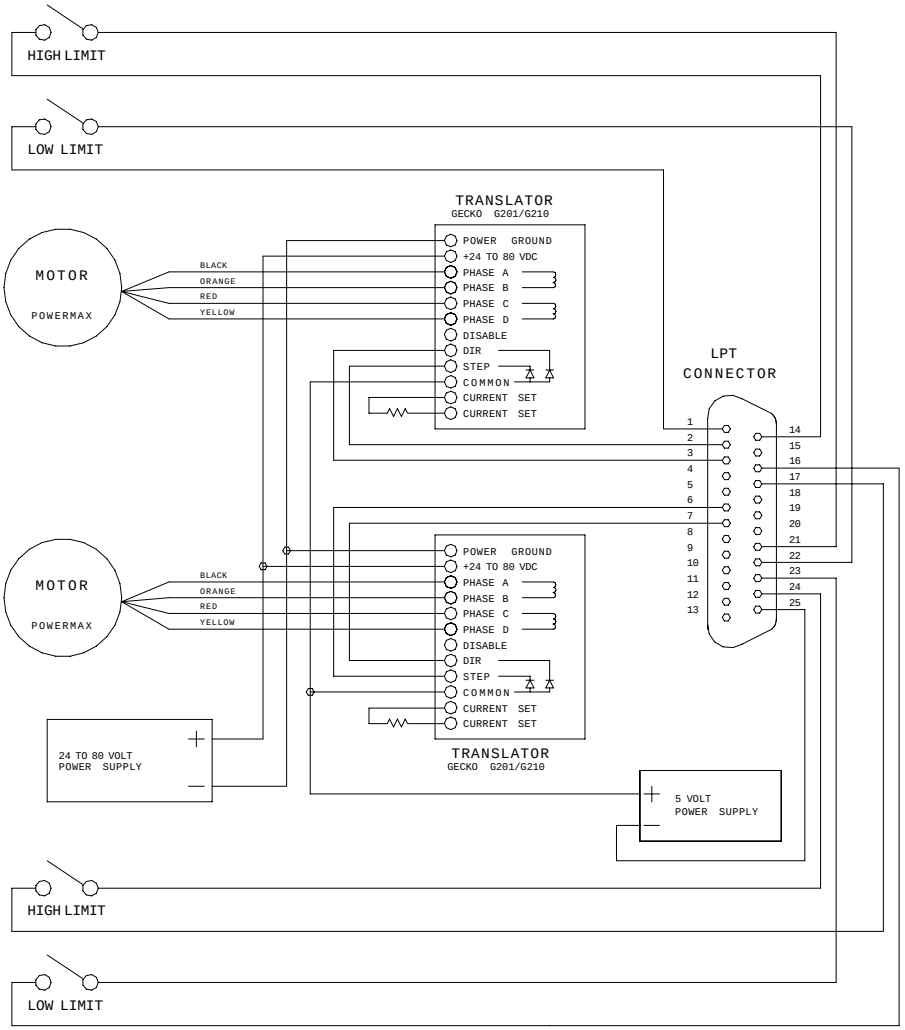


Figure 4-3

WIRING DIAGRAM - Gecko 201/210 Translator

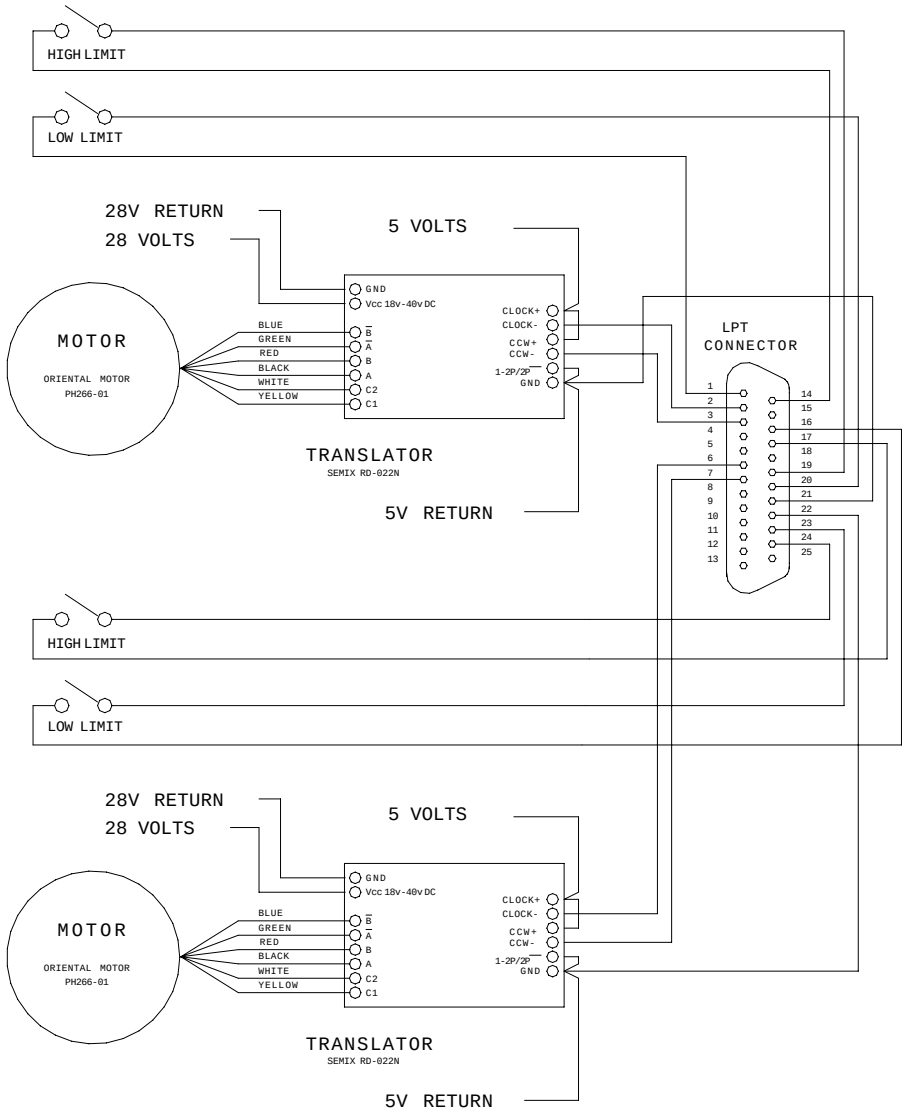
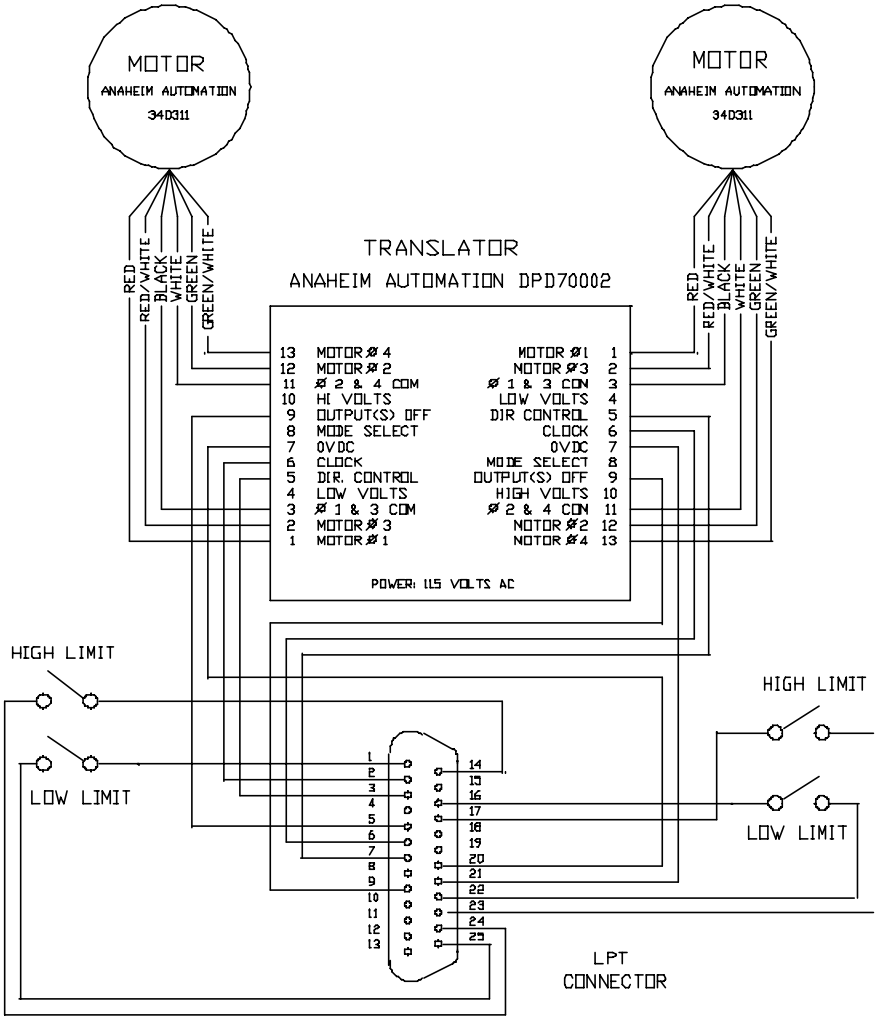


Figure 4-4

WIRING DIAGRAM - Semix RD-022N Translator



NOTE:
SET TRANSLATOR INTERNAL JUMPERS FOR CLOCK AND DIRECTION MODE AND NEGATIVE PULSE INPUT

Figure 4-5

WIRING DIAGRAM - Anaheim Automation DPD Series Translator

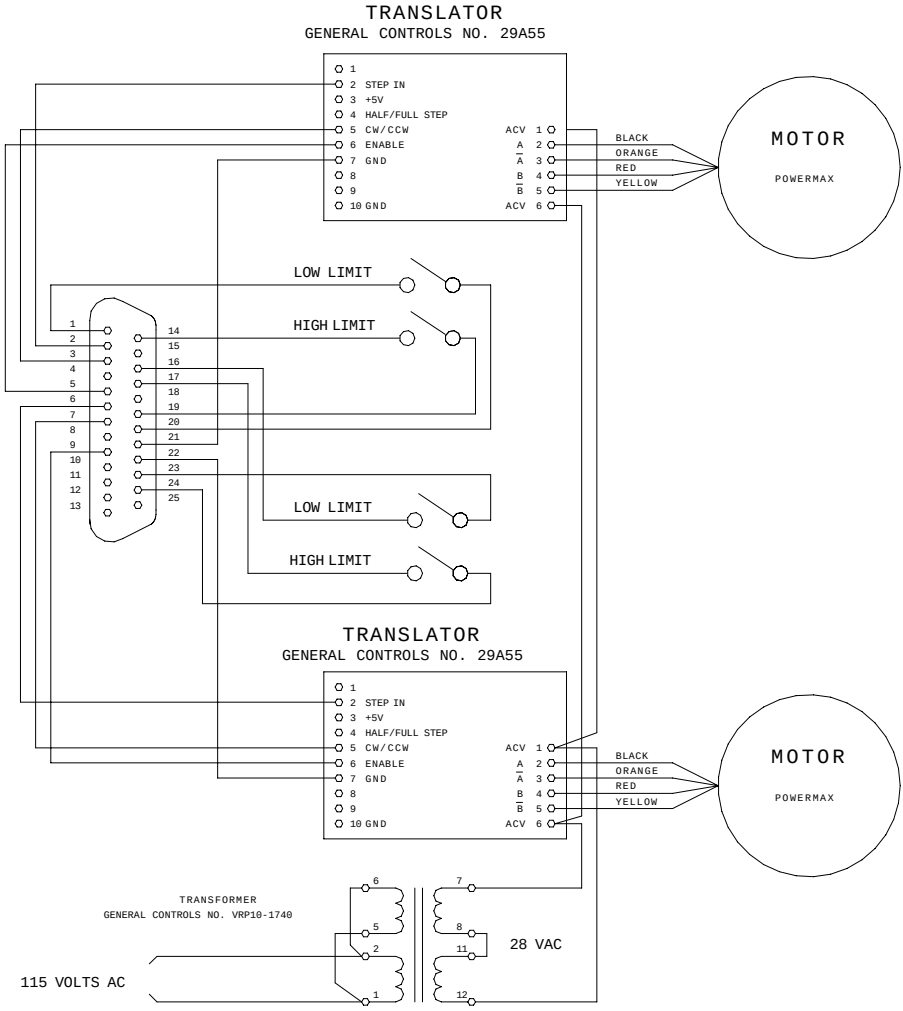


Figure 4-6

WIRING DIAGRAM - General Controls 29A55 Translator

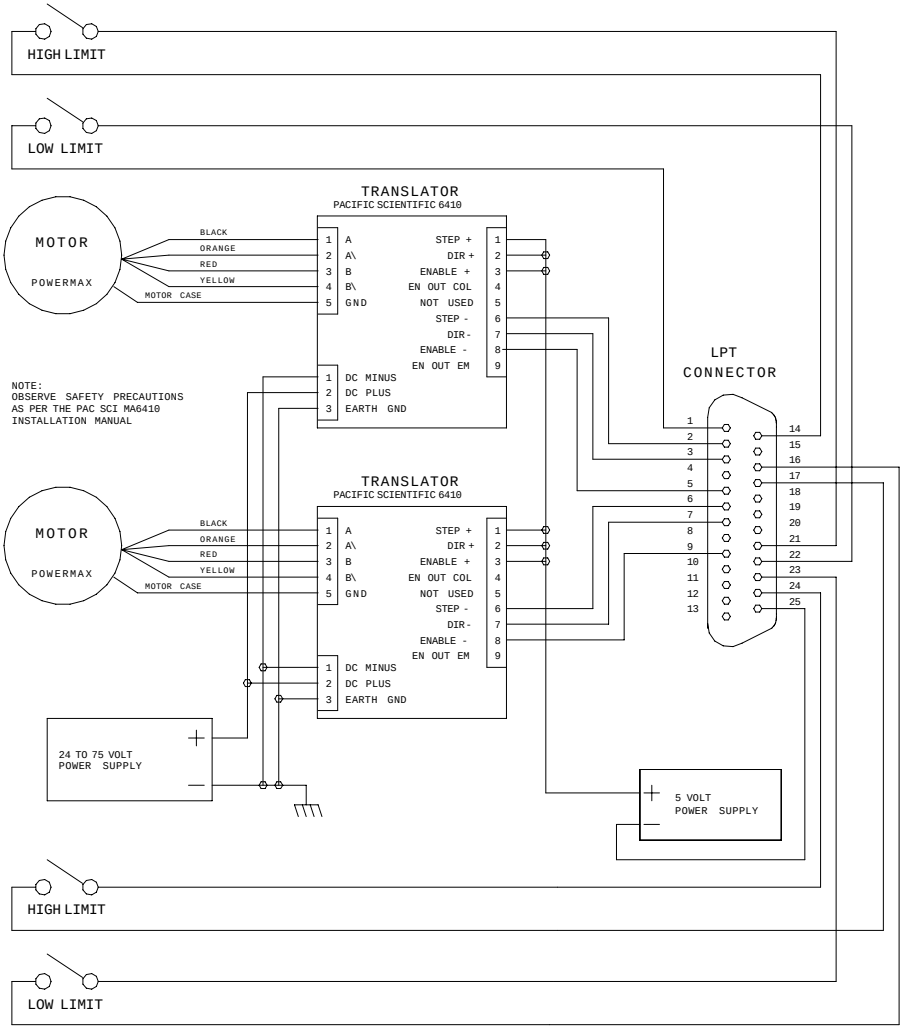


Figure 4-7

WIRING DIAGRAM - Pacific Scientific 6410 Translator

Chapter 5

SIGNAL DEFINITIONS

The purpose of this section is to provide a description of the signals which are present on the Printer Adapter card, and how these signals are used with the **Indexer LPT** system.

Information concerning the electrical characteristics of these signals has been obtained from the IBM Technical Reference Manual. Variations may exist, however it is our experience that these variations are slight. To determine the exact electrical specifications for any given printer adapter, refer to the manufacturer's specifications.

Two groups of signals per adapter

Indexer LPT divides the available connections on each printer adapter into two essentially identical groups, thereby allowing for control of two axes per adapter. We shall refer to these groups as the "X group" and the "Y group". Grouping associates each pin of the 25 pin connector on the printer adapter card with its function. The base address of the particular adapter card determines the axis which each group represents. The association between card addresses, pin functions, and pin numbers is depicted in Figure 4-1.

For example, if the base address of an adapter card is 278(hex), then the "X group" of signals present on that adapter comprises the "a" axis and the "Y group" of signals on that adapter comprises the "b" axis. Referring to the signal definitions below, (or Figure 4-1) the "step" signal for axis "a" is located on connector pin 2 and the "step" signal for axis "b" is located on connector pin 6.

Signal Ground

The “signal ground” connection referred to in this text is available at connections on pins 18 through 25.

Step

X group pin number 2

Y group pin number 6

Electrical Characteristics

TTL level output. Sourced from a 74LS374 buffer/latch with 30 ohms of series resistance and .0022uF of parallel capacitance.

Usage

Controls translator “step” or “pulse” input. This signal is normally high. Negative pulses are issued during commands which cause motion.

Direction

X group pin number 3

Y group pin number 7

Electrical Characteristics

TTL level output. Sourced from a 74LS374 buffer/latch with 30 ohms of series resistance and .0022uF of parallel capacitance.

Usage

Controls translator “direction” or “clockwise/counterclockwise” input. The level of this signal determines the direction the motor will turn when pulses are issued on the associated “step” signal. This signal is high during rotations in the positive direction, and is low during rotations in the negative direction.

Reduced Current

X group pin number 4

Y group pin number 8

Electrical Characteristics

TTL level output. Sourced from a 74LS374 buffer/latch with 30 ohms of series resistance and .0022uF of parallel capacitance.

Usage

This is an auxiliary output signal which may be used to control a circuit which commands the translator to reduce the current applied to the stepper motor windings.

This output responds only to the *reduced_current* command, and can therefore alternately be used as a general purpose output signal.

All Windings Off

X group pin number 5

Y group pin number 9

Electrical Characteristics

TTL level output. Sourced from a 74LS374 buffer/latch with 30 ohms of series resistance and .0022uF of parallel capacitance.

Usage

This is an auxiliary output signal which may be used to control the translator input to remove power from the stepper motor.

This output responds only to the *winding_power* command, and can therefore alternately be used as a general purpose output signal.

Low Limit Switch

X group pin number 1

Y group pin number 16

Electrical Characteristics

TTL level input internally pulled high through a 4.7K ohm resistor. Input characteristics correspond to that of a 74LS240 buffer.

Usage

This signal may be wired in series with a normally open limit switch which completes a circuit connection to signal ground when closed.

When this signal is not connected it is internally pulled high, indicating an “open low limit switch” condition. When this signal is connected to signal ground a “closed low limit switch” condition prevails.

A “closed low limit switch” condition arrests motion in the negative (-) direction.

DANGER

The limit switch detection features of this program are designed only to provide limit detection within the normal operating region of the device being controlled and NOT to provide over-travel protection in cases where equipment damage or personal injury may result.

In cases where equipment damage or personal injury is possible due to over-travel, other means of limit protection is imperative.

High Limit Switch

X group pin number 14

Y group pin number 17

Electrical Characteristics

TTL level input internally pulled high through a 4.7K ohm resistor. Input characteristics correspond to that of a 74LS240 buffer.

Usage

This signal may be wired in series with a normally open limit switch which completes a circuit connection to signal ground when closed.

When this signal is not connected it is internally pulled high, indicating an “open high limit switch” condition. When this signal is connected to signal ground a “closed high limit switch” condition prevails.

A “closed high limit switch” condition arrests motion in the positive (+) direction.

DANGER

The limit switch detection features of this program are designed only to provide limit detection within the normal operating region of the device being controlled and NOT to provide over-travel protection in cases where equipment damage or personal injury may result.

In cases where equipment damage or personal injury is possible due to over-travel, other means of limit protection is imperative.

Auxiliary Input

X group pin number 13

Y group pin number 12

Electrical Characteristics

TTL level similar in characteristics to an un-terminated LS-TTL input.

Usage:

This signal is used as a general purpose input. It must be driven to logic levels to provide a reliable indication.

Chapter 6

GETTING STARTED

Software installation

Make sure to read the installation notes in the README file located on the distribution diskette.

Place the distribution diskette in the A:> drive. Select **Run** from the **Start** menu, and type:

```
A:\Setup
```

Snap **OK** to start the Setup procedure. Follow the directions presented to you in the menus that follow. Make sure to read all the notes provided in the **Help** screens by snapping on the **Help** button corresponding to each step.

The Hardware Assist Module (HAM)

The *Hardware Assist Module*, which we refer to as the **HAM**, is a small electronic module which attaches to an available printer port connector. The **HAM** contains electronics to support the *feed rate override* feature (**FRO**). For wiring to the **HAM** and use of **FRO**, refer to the chapter entitled FEED RATE OVERRIDE.

Make sure that the **HAM** is attached to an available printer port connector. The **HAM** has two connectors, one male (pins), and one female (sockets). Make sure that the male connector on the **HAM** attaches to the female connector on a printer port. To avoid potential damage to the **HAM**, DO NOT plug the female connector of the **HAM** onto to any male connector on the computer.

Hardware Checkout Using IXDIAG

In the development of control systems it is greatly helpful to make sure that the hardware elements are functioning properly before attempting to configure an application program or to write and debug software. With this in mind, the program **IXDIAG.EXE** is supplied with your **Indexer LPT** package. **IXDIAG.EXE** can assist you in your first installation and use of **Indexer LPT**, and can also serve as a useful tool in troubleshooting problems with new designs and system hardware.

IXDIAG.EXE can be launched by snapping over its icon in the **Start->Programs** menu.

IXDIAG.EXE allows full access to the functions of **Indexer LPT** by means of an easy to use, menu driven interface. This program allows the user to exercise all of the **Indexer LPT** commands.

When **IXDIAG.EXE** is first invoked, it looks for the presence of a correctly installed **Indexer LPT** "motor" device. If **Indexer LPT** is installed correctly, **IXDIAG.EXE** will inform the you that the device driver is present. **IXDIAG.EXE** will then execute subsequent user control over **Indexer LPT** by means of calls to its device driver.

If **Indexer LPT** software is not installed, or if it is not installed correctly, **IXDIAG.EXE** reports the difficulties which it encountered.

We recommend that the new user first exercise motion control hardware by means of **IXDIAG.EXE**. This program is easy to use, and provides an excellent means for experimenting with **Indexer LPT** commands.

Using Indexer LPT with Application Programs

Application programs written to accommodate various types of automatic machinery are available from Ability Systems. Configuring these programs does not require computer programming, but does require a working knowledge of **Indexer LPT**.

The application programs' setup menus and dialogs help you associate **Indexer LPT** commands to the task at hand. Once the application program is configured, the user interacts with the menus and prompts of the application program, and communication between **Indexer LPT** and the application program occurs in the background.

For example, consider the design of a cutting tool that uses a motor to advance a linear stage. Typically, you may start implementing this design by simply checking the operation of the motor from IXDIAG.EXE using an **Indexer LPT** command such as:

```
move:a,4000
```

This command exercises the motor controlled by the “a” axis by applying 4000 pulses to the translator. The amount that the motor shaft rotates depends on the internal geometry of the motor and the configuration of the translator. The linear distance that the stage subtends depends on the mechanical ratio between the rotation of the motor shaft and the screw follower or pulley that it is coupled to. Nevertheless, using the *move* command you can experiment with the motor at the lowest and most basic level.

Once you are assured that the motor is moving properly, you may wish to calibrate the stage by measuring the distance that it moves for a given amount of step pulses. After you enter this information (number of steps and corresponding distance) into the stage setup dialog provided in the application program, you can use the features of the application program to control the stage using units of measure that relate to it’s actual function, e.g. “inches, or millimeters” instead of “steps”.

Setup menus and dialogs allow you to use **Indexer LPT** commands and queries to design and customize your machine to do such things as monitor external switches or to activate solenoids. When designing a machine, you would typically exercise an **Indexer LPT** command or query using IXDIAG.EXE to troubleshoot your wiring at the simplest level. You would then enter that command in an appropriate dialog of the application program.

For example, the Ability Systems **G Code Controller** product allows you to customize “M” code commands with sequences of **Indexer LPT** actions. Suppose you wanted to customize “M12” to activate an air clamp using the **Indexer LPT** digital output command:

```
winding_power:c,0
```

and you wanted to customize “M13” to release the clamp using:

```
winding_power:c,1
```

After wiring the solid state relay that engages the clamp to be activated by the output signal associated with to these commands, you can verify its functionality by exercising these commands from IXDIAG.EXE. Once you have verified that these commands are appropriate and functioning, you can type them into the list boxs in the dialog provided in the **G Code Controller** program for customizing M codes After that, the user interacts with the application program with M12 and M13 to control the clamp. The application program, in turn, communicates these commands to **Indexer LPT**.

In another example, suppose you wanted to use the low limit switch input on the **Indexer LPT** “f” axis to read a normally open pushbutton as a start switch. To do this, you would wire the pushbutton

so that it grounded the associated input pin when the button is pushed. You could verify the integrity of your wiring using the **View Signal Status** section of IXDIAG.EXE, watching the low limit switch status field for this axis change from “0” to “1” when you pressed the switch.

The **Indexer LPT** query command to read this switch is:

```
-limit?:f
```

The **View Signal Status** section of IXDIAG.EXE repeatedly issues this command, updating the display after each try. However, you can exercise this command from the **Command Motor** section by typing the command and pressing **Enter**. If the switch is open when you press **Enter** it will respond with “0”. If the switch is closed it will respond with “1”. Once you have verified that you have wired the switch correctly, and that you are using the correct syntax to query the input pin, you can enter this information into the dialog that the application program provides to configure an external start switch. Consequently, when the application program needs to wait for the condition of the start switch, it will transparently communicate with **Indexer LPT** using this command. The end user need only press the switch at the appropriate time.

These are some of the ways in which you may need to use **Indexer LPT** commands when configuring application programs available from Ability Systems. User written and software available from third parties may use different configuration techniques, such as setup scripts.

Programming with Indexer LPT

Indexer LPT receives commands comprised of ordinary ASCII character strings which are written to the “motor” device in much the same manner as writing to a file named “motor”. All command strings must end with a carriage return character.

Indexer LPT leaves an ASCII message in a one-line buffer, which we call the “mailbox”, after the completion of each command. The mailbox can be read in much the same manner as reading from a file. All messages in the mailbox are terminated with a carriage return character.

Since **Indexer LPT** is a character device driver, and behaves much the same as a file, it can be controlled by virtually any programming language which has the capability of communicating ASCII text to and from a file.

These languages include C, Pascal, and BASIC. Even higher level languages and programs can be used to communicate with **Indexer LPT**.

The name of the **Indexer LPT** device is “motor”. When using language file handling functions to communicate with **Indexer LPT**, use “motor” as the file name.

When communicating with disk files and devices, application languages usually obtain a “file pointer” by means of a “file open” command. Subsequent reads and writes are accomplished by referencing this file pointer with the appropriate language function.

Safety

Make sure to read the chapter entitled **Special Considerations**, especially the section involving safety.

Be aware that some motion control applications require interaction between the application program and the operating systesm, as well as with **Indexer LPT**, to provide for operator safety. These measures require a knowledge of both computer programming and machine design, and cannot be implemented using the simplified methods outlined in the next section.

Using Indexer LPT from a DOS Window

Sending Text Files to Indexer LPT

One simple means of writing to **Indexer LPT** is by means of the DOS “copy” command. The usage of the “copy” command is as follows:

```
copy <source file> < destination file>
```

A file containing **Indexer LPT** commands can be created using an ASCII text editor such as Window’s **Notepad** or DOS’s **EDIT**. The transfer of these commands to **Indexer LPT** is accomplished by copying this file to a “file” called “motor”.

In an example, motion hardware is wired to printer port hardware on the “c” axis. Using a text editor, a file named **TST1.CMD** is created. The content of this file is as follows:

```
move:c,2400
```

(Make sure you end this line with a carriage return).

This file is copied to the motor device by typing the following from the DOS prompt:

```
C>copy tst1.cmd motor<Enter>
```

The motor will then move 2400 steps in the positive (+) direction.

In another example using the same hardware setup, a file named **TST2.CMD** is formed having the following contents:

```
set_home:c
move:c,3500
move:c,-1000
move:c,800
home:c
```

(Make sure the last line is ended with a carriage return)

This file is copied to the motor device by typing the following from the DOS prompt:

C> copy tst2.cmd motor<Enter>

The motor will then follow the sequence of 3500 steps in the positive (+) direction, 1000 steps in the negative (-) direction, 800 steps in the positive (+) direction, and return to the starting position.

DOS Batch Files

A batch file is essentially a text file that contains a sequence of DOS commands. Batch files can be composed using a text editor such as DOS's **EDIT**, or Window's **Notepad**. The file name extension for a batch file must be ".BAT". A batch file can be run from within a DOS window by typing its name, similar to running an executable (.EXE) program. You can also construct a Windows **Shortcut** to a batch file in the same manner as you can construct a **Shortcut** to an executable program.

Using a batch file, you can implement sequences of **Indexer LPT** commands for cycling certain types of machinery that requires only a small amount of program or user interaction. Batch files are also handy to use in troubleshooting, since they can be quickly made to implement repetitive sequences.

One easy method is to simply use a batch file to automatically repeat the previously described method of copying a text file to **Indexer LPT**. For example, to do this you may construct a DOS batch file named **SEQUENCE.BAT,** which contains the following ASCII text:

```
:loop
copy tst2.cmd motor
goto loop
```

You can execute this batch file by typing the following from the DOS prompt:

C>sequence<Enter>

It may be desirable to send commands to **Indexer LPT** directly

from within the batch file by directing the output of the **ECHO** command to the "motor" device with the ">" operator. The following batch file behaves essentially the same as the previous example:

```
:loop
echo set_home:c>motor
echo move:c,3500>motor
echo move:c,-1000>motor
echo move:c,800>motor
echo home:c>motor
goto loop
```

Note that it is not necessary to reset the home position each pass through the loop. Consequently, the example code may be changed as follows:

```
echo set_home:c>motor
:loop
echo move:c,3500>motor
echo move:c,-1000>motor
echo move:c,800>motor
echo home:c>motor
goto loop
```

This example shows how a batch file may be used to implement a simple repetitive sequence. The use of batch files is more completely described in legacy DOS documentation, such as the book *Running MSDOS* by Van Wolverton (Microsoft Press).

Using DOS Commands to Read From Indexer LPT

After the execution of each command, **Indexer LPT** deposits a string of characters in a one line buffer which can be read similar to reading a line of text from a file. One simple means of seeing what is in the buffer is by using the DOS **type** command. The usage of the **type** command is as follows:

C> type <filename><Enter>

For example, immediately after **Indexer LPT** is installed the response buffer contains the string, "installation successful". To view the contents of the buffer, type the following from a DOS prompt:

C> type motor<Enter>

The contents of the response buffer will appear on the screen as

shown:

```
C>installation successful
```

Programming Indexer LPT from C

In the following example, the C programming language is used to perform simple reads and writes to the **Indexer LPT** "motor" device. When using this example there are some considerations the programmer should keep in mind.

Two file pointers are opened for the "motor" device. One file pointer, `fpr`, is opened for read only; the other file pointer, `fpw`, is opened for write only. Consequently, the file pointer `fpr` is used in all of read operations and `fpw` is used in all of the write operations.

Notice that the program opens the read pointer first, checks to see if it is valid, and exits the program with an error code if the pointer is not valid. This action checks for the presence of the "motor" device on the system and exits the program if it is not there.

Consider the situation of opening the write pointer first. If the "motor" device was not present on the system the program would create a "counterfeit" file called "motor". This would then create an elusive problem. The program would then write to and read from the "counterfeit" file as if it were writing to and reading from **Indexer LPT**, except of course, the **Indexer LPT** would not be functioning.

If this error is made it is important to delete the counterfeit "motor" file before correctly installing the real **Indexer LPT** "motor" device. If the counterfeit "motor" file is not removed before installing **Indexer LPT**, your operating system will not allow deletion of the counterfeit file until the real **Indexer LPT** "motor" device is removed from the system.

If the programmer wishes to convert **Indexer LPT** step position string information to a numeric value, the C language **atol** function is recommended if you are using a 16 bit compiler. If you are using a 32 bit compiler, you may use **atoi**. **Indexer LPT** tracks position to a magnitude of 2,147,483,647 steps in either direction. Consequently, the **signed long** data type is most appropriate for 16 bit compilers, while an integer will suffice for 32 bit code.

The following example code was tested under the Borland C compiler. We have found that Borland C flushes the output buffer stream during each execution of the `fputs()` function. Microsoft C, on the other hand, queues the output buffer. Consequently, when using Microsoft C, an `fflush()` function must be performed following each write to assure that the output is sent to the device when expected.

The distribution diskette contains two sample programs in C: SAMPLTC.C for (Borland) Turbo C, and SAMPLMSC.C for Microsoft C. SAMPLMSC.C contains an **fflush()** command after each device write. SAMPLMSC.C will run under both Microsoft C and Borland C.

```
/* Example C program demonstrating the use
of Indexer LPT using one file pointer for
reads and another file pointer for writes
*/
#include stdio.h
main()
{
   FILE *fpr, *fpw;
   char instring[80];
   long position;
   if( (fpr = fopen(“motor”, “r”)) == NULL)
   {
       printf(“cannot open motor
              device\n”);
       exit(1);
   }
   fpw = fopen(“motor”, “w”);

   /*  Don’t forget to terminate each
    command string with the \n character */

   /* Set up for position tracking */
   fputs( “set_home:a\n”, fpw );

   /* Move the motor */
   fputs( “move:a,2000\n”, fpw );

   /* Read position from the mailbox */
   fgets(instring, 80, fpr);
   printf(“%s”, instring);

   /* Convert ASCII position to numeric */
   position = atol(instring);
   printf(“%ld\n”, position);

   /* Send the motor home */
```

```
    fputs( "home:a\n", fpw );

    /* Read mailbox again and print */
    fgets(instring, 80, fpr);
    printf("%s", instring);

    fclose (fpw);
    fclose (fpr);
    exit(0);
}
```

Programming Indexer LPT from BASIC

In BASIC, the `OPEN` command is used to associate a file number with the **Indexer LPT** "motor" device, and the `PRINT #` command is generally used to write to **Indexer LPT**. For example, to associate the output file number 2 with **Indexer LPT**, use the following BASIC command:

```
OPEN "motor" FOR OUTPUT AS #2
```

After associating a file number with **Indexer LPT**, you may now use the file number as a handle to perform write operations. In this example, to send the character command string "`move:a,1000,b,2000`" to **Indexer LPT**, use the following line of BASIC code:

```
PRINT #2, "move:a,1000,b,2000"
```

Often it is necessary to format the value of variables into strings that you send to **Indexer LPT**. BASIC provides an easy means of doing this using semicolons as delimiters. For example, suppose you had a variable, X, whose value was 1000. Consider he following line of BASIC code:

```
PRINT #2, "move:a,";X
```

This code formats the value of X into the string that is "printed" to **Indexer LPT**. In this example, the string that is actually sent to **Indexer LPT** is

```
move:a,1000
```

Now consider a variable , `X`, having a value of 1000, and a variable, `Y`, having a value of 2000. Consider the following code:

```
PRINT #2, "move:a,";X;"b,";Y
```

The string that **Indexer LPT** receives in this example is:

```
move:a,1000,b,2000
```

To read from the **Indexer LPT** "motor" device a `LINE INPUT` command is recommended instead of the normal `INPUT` command. This is due to the fact that some of the **Indexer LPT** messages consist of two or more fields separated by commas. Normally the `INPUT` command will stop reading after the first comma. The `LINE INPUT` command will read up to the carriage return, line feed sequence, and will therefore capture the entirety of the message.

To use `LINE INPUT` to read from **Indexer LPT**, you must first obtain a file number to read. For example, you may use the following line of code

```
OPEN "motor" FOR INPUT AS #1
```

Now to read the mailbox into a string variable entitled `RESP$`, use the following code:

```
LINE INPUT #1, RESP$
```

If a numeric value is anticipated, and the programmer wishes to convert it to a numeric value, the BASIC language **VAL$** operator is recommended. For example, suppose the string variable, RESP$, contained the ASCII sequence of characters:

```
3571
```

To convert this string to a numeric value, and assign that value to a numeric variable, Z, use the following code:

```
Z = VAL(RESP$)
```

In the following program example, GWBASIC is used to perform simple reads and writes to the **Indexer LPT** "motor" device. Please note that GWBASIC may generate an error if a device is opened for OUTPUT after opening a device of the same name for INPUT. An error is not generated if the device is first opened for OUTPUT, then opened for INPUT.

```
10 OPEN "MOTOR" FOR OUTPUT AS #2
20 OPEN "MOTOR" FOR INPUT AS #1
30 REM Set up for position tracking
40 PRINT #2, "SET_HOME:A"
50 REM Move the motor
60 PRINT #2, "MOVE:A,2000"
70 REM Read position from the mailbox
80 LINE INPUT #1, RESP$
90 REM Convert ASCII position to numeric
100 POSITION = VAL(RESP$)
110 REM Print both ASCII and numeric
```

```
120 PRINT RESP$,POSITION
130 REM Send the motor home
140 PRINT #2,"HOME:A"
150 REM Read mailbox again and print
160 LINE INPUT #1,RESP$
170 REM Convert ASCII to numeric and print
180 POSITION = VAL(RESP$)
190 PRINT RESP$,POSITION
200 CLOSE #1
210 CLOSE #2
220 END
```

The manner in which BASIC requires you to use file numbers may vary, depending on the version of BASIC that you are using. For example, some versions of Visual BASIC do not allow a file or device to be assigned a number to read and another number to write at the same time. In this case, you must close the file number to write before opening one to read, and vise verse. This can conveniently be accommodated in your program with a subroutine, or in a function call, and it takes relatively little processing time.

Programming Indexer LPT from Pascal

In the following example, Borland Turbo Pascal is used to perform simple reads and writes to the **Indexer LPT** "motor" device. In using this example there are some considerations the programmer should keep in mind.

Unlike the examples in C and BASIC, only one file variable, **FileVar**, is opened for the "motor" device. This file variable is used for both read and write operations.

In this example a "dummy" read is attempted before the file variable is used for writing. Similar to BASIC, if the "motor" device does not exist on the system, a run time error is generated and the program immediately terminates.

The programmer should take the same precautions against creating a "counterfeit" file named "motor" when using Pascal as with C and BASIC.

The Pascal `Writeln` function is used instead of the ordinary `Write` function because `Writeln` includes the carriage return line feed sequence which **Indexer LPT** requires.

Similar to the example in BASIC, the Pascal `Readln` function grabs the entire message string from the mailbox including the carriage return and line feed.

In the example the file variable is initialized before each read operation with a `Reset` function, and before each write operation with a `Rewrite` function. This is necessary to keep the file pointers properly aligned. In order to conserve code the programmer may wish to compose a function which writes to the device and also deposits the response from the mailbox into a string variable. The purpose of the example, however, is to clearly illustrate writing to and reading from the **Indexer LPT** "motor" device.

If the programmer wishes to convert position messages to a numeric value, the Pascal language **Val** function is recommended, and the `Real` data type is appropriate.

```
:program PascalExample;
var
     FileVar : Text;
     instring : string[80];
     position : real;
     code : integer;

begin
{ Open file variable }
      Assign(FileVar, 'motor');

{ Force an error if the device is not
installed }
      Reset(FileVar);
      ReadLn(FileVar, instring);

{ Set up the axis for position tracking }
      Rewrite(FileVar);
      WriteLn(FileVar,'set_home:a');

{ Move the motor }
      Rewrite(FileVar);
      WriteLn(FileVar,'move:a,2000');

{ Read position from the mailbox }
      Reset(FileVar);
      ReadLn(FileVar, instring);

{ Print the position to the screen }
      WriteLn(instring);
```

```
{ Convert the position string to a numeric
value }
     Val(instring, position, code);

{ Print it to the screen }
     WriteLn(position:0);

{ Send the motor home }
     Rewrite(FileVar);
     WriteLn(FileVar,'home:a');

{ Read again and print to screen}
     Reset(FileVar);
     ReadLn(FileVar, instring);
     WriteLn(instring);

{ Close device before leaving }
     Close(FileVar);

end.
```

Chapter 7

FEED RATE OVERRIDE

What Feed Rate Override Is

The *feed rate override* (**FRO**) feature adds the ability to smoothly and continuously adjust motor speeds concurrently with motor motion by means of an externally applied control voltage. It applies to single and multiple axis motion, including contouring. **FRO** applies universally to all motion timing, including starting speeds, running speeds, acceleration and *vshift*.

One way to visualize the **FRO** feature is to consider an externally applied voltage to establish a "set point" defining a percentage faster or slower that motion shall occur with respect to the programmed motion rates. Whenever the set point changes, **FRO** technology tracks this percentage, making the actual motion align with the desired overide setting.

The tranfer characteristic between voltages applied and percentage override is configurable by means of the *fro_low*, ,*fro_high*, *fro_lowvolt* and *fro_highvolt* registers (although the default values are sufficient for most applications).

The **FRO** set point can also be offset under program control (using the *set_fro_offset* command to change the *fro_offset* register). This allows the application program to modify speed within a queued contour “as if” it was being adjusted by an externally applied signal. The total amount of **FRO** applied to the set point is the combination of the amount recorded in the *fro_offset* register and that applied by means of the external control voltage. The toal feed rate override will never exceed the limits established by the *fro_low* and

the *fro_high* registers.

Software control over the **FRO** set point is always effective. It is not disabled when the *fro_enable* register is "0". When the *fro_enable* register is “0”, the set point established by the *fro_offset* register is always relative to a **FRO** value of 100%.

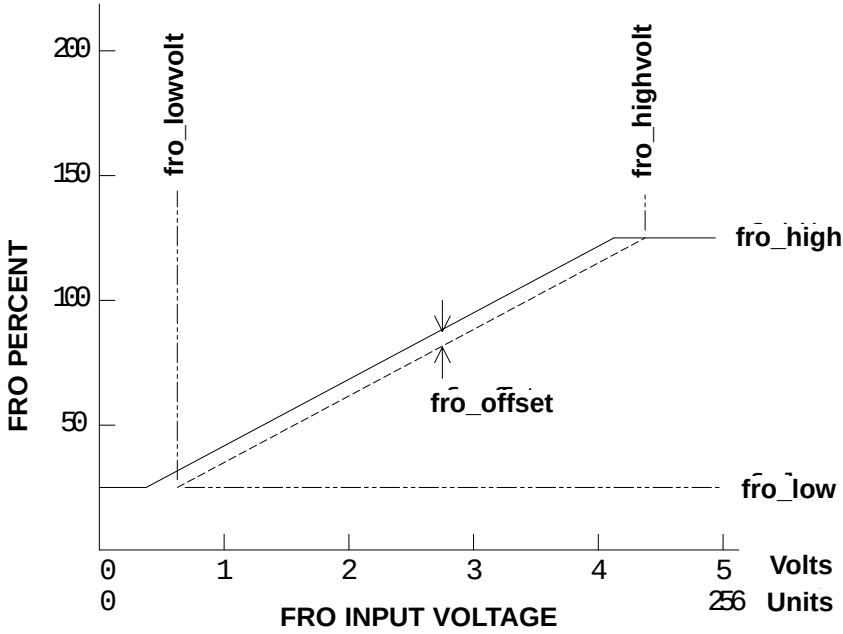


Figure 7-1

When the *fro_enable* register contains a value of “1”, an externally applied control voltage contributes to the **FRO** set point. The control voltage must be within the range of zero (0) to five (5) volts, and can be generated by means of an external circuit or a simple potentiometer.

Using an external circuit, feed rate can be dynamically changed to implement low cost adaptive control.

A simple potentiometer easily implements a means for an operator to vary the speed of motion with a control knob.

The Hardware Assist Module Supports FRO

The **Hardware Assist Module** (**HAM**) provides the necessary interface to communicate the FRO control voltage to the software. The **HAM** consists of a small module having a DB25 male con-

nector on one end, and a DB25 female connector on the other.

The DB25 male connector attaches to a parallel port dedicated for use with **Indexer LPT**. On computers with multiple parallel ports, **Indexer LPT** will automatically detect which port the **HAM** is attached to. The base axis (**X Group** axis) that the HAM is detected on can be read by means of the **Indexer LPT** *ham_axis?* query.

(**X Group** and **Y Group** signals are shown in the chart on page 25).

Indexer LPT and companion products are supported by the **HAM**. Reverse engineering the **HAM** violates your agreement to use the software and invalidates your license.

All of the pins of the parallel port connecting to the **HAM** that may be used for **Indexer LPT** functions are presented electrically to the DB25 female connector on the far side of the **HAM** with the exception of pins 6, 7, 8 and 9 (the output signals from the **Y Group** axis). All signals from the **X Group** axis can be used normally.

Limit switch inputs for the **Y Group** axis, consisting of pins 16 and 17, as well as the **Auxiliary Input** pin 13, feed straight through. You can use pins 16 and 17 for the *feed hold* feature if you wish, and use the **Auxiliary Input** as normal.

Pins 11 and 15 are dedicated to the FRO feature. Pin 11 accepts the 0-5 Volt FRO control voltage. Pin 15 provides a voltage source capable of applying power to a 1K potentiometer.

Wiring

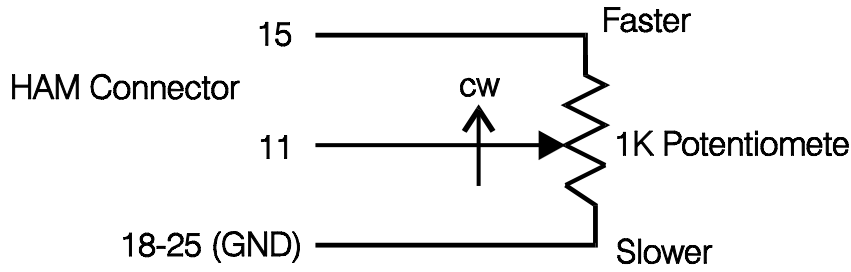


Figure 7-2

For potentiometer control, connect the wiper of the potentiometer to pin 11. Connect the end of the potentiometer on the side that the wiper moves towards when you want speed to increase to pin 15. Connect the other end of the potentiometer to ground, located on any of pins 18 to 25.

Activation

By default the **FRO** feature is disabled. You can enable **FRO** by means of the *set_fro_enable* command. The command:

```
set_fro_enable:1
```

enables FRO. The command:

```
set_fro_enable:0
```

disables FRO.

Be careful NOT to enable **FRO** if a control voltage is not being applied to the **HAM**, or if a potentiometer is not installed. An open circuit on pin 11 will cause erratic speed variation when **FRO** is enabled.

> **DO NOT ACTIVATE FEED RATE OVERRIDE IF A POTENTIOMETER IS NOT CONNECTED, OR IF A CONTROL VOLTAGE IS NOT APPLIED.**

If you wish to designate the status that FRO will assume when the software loads you may use the *save_fro_enable* command. This command will preserve the current state of the *fro_enable* register when the computer is shut down, and restore it the next time **Indexer LPT** loads.

Resolution

Indexer LPT is capable of resolving the control signal to up to 100 variations of speed, corresponding to the voltage span from *fro_lowvolt* to *fro_highvolt*. Resolution can be changed by means of the *set_fro_res* command. The default and recommended resolution is 50. You can save the current resolution to non-volatile memory by means of the *save_fro_res* command.

Voltage Span

Voltage span adjustment is useful for calibrating the position of a control knob to a calibrated dial. The unit of value used is 5/256 volts. In other words, a value of 256 corresponds to 5 volts. You may use the *set_fro_highvolt* command to set the upper (fastest) position of the dial. Use the *set_fro_lowvolt* command to set the lower (slowest) position of the dial. The default value for the *fro_highvolt* register is 256, representing 256 units of 5/256 Volts, or 5 Volts The default value for *fro_lowvolt* is 0, representing 0 Volts.

Voltage to Speed Transfer Ratio

The amount of **FRO** speed variation can be modified by the

set_fro_low and *set_fro_high* commands. The *fro_low* value corresponds to the percentage of nominal speed when the control voltage is at the *fro_lowvolt* limit. It also represents the lowest percentage of feed rate override that can be applied.

The *fro_high* value corresponds to the percentage of nominal speed when the control voltage is at the *fro_highvolt* limit. It also represents the highest percentage of feed rate override that can be applied.

Default values for the *fro_low* and the *fro_high* registers are 30 and 130 respectively.

For example, in the default configuration where the *fro_lowvolt* and *fro_highvolt* registers are 0 and 256 respectfully, when the control voltage is 5, the speed of motion will be increased to 130 percent of the programmed value. When the control voltage is 0, the speed of motion will be decreased to 30 percent of the programmed motion. With the (default) resolution of 50, 50 different speeds can be realized depending on the position of the potentiometer, or the level of the control voltage.

The **FRO** feature was designed to vary speeds within the normal operating range of most control applications. It is NOT meant to slow speeds to a stop. Use the **feed hold** feature for stopping motion.

Physical Range Limits

Caution should be exercised to avoid over stress of motion controlled components. When selecting allowable limits for speed, acceleration and *vshift*, account for changes that various **FRO** settings will introduce.

.

Chapter 8

QUEUE PROCESSING

What Queue Processing Is

It is often desirable, especially in machine tool control applications, for motion to proceed from one command into the next command without decelerating to a stop between each command. **Indexer LPT**'s *queue processing* feature accommodates this task. "Look-ahead" processing not only over-rides deceleration between commands, but also anticipates stresses at the transition points, and adjusts axis speeds accordingly.

The commands which are most used in queue processing are *q_begin*, *q_end*, and *q_go*. The following command sequence demonstrates loading and executing a command queue in a simple example:

```
q_begin
feed:a,4000
arc_to_angle:ccw,a,0,b,1000,90,c,1000
feed:a,1000,b,4500,c,3100
feed:a,200,b,1150,c,500
feed:a,100,c,300
q_end
q_go
```

The *q_begin* command instructs **Indexer LPT** to begin recording commands into the queue buffer. The *q_end* command terminates

the recording process. Motion begins when the *q_go* command is executed.

After the *q_end* command, but before the *q_go* command, you may execute **Indexer LPT** motion commands individually. This feature allows the application program to perform the relatively time consuming task of loading the queue in a separate operation. After the queue is loaded, a machine component can be moved into into position, and the queue can be executed without the calculation dwell associated with loading the queue. This feature is most used in cutting tools. If the tool were in contact with the work when the queue was being loaded a burn mark may result. Instead, the queue can be loaded first, then the tool can be moved into the work, and the queue executed (virtually) immediately with the *q_go* command.

Contouring

A smooth contour can be constructed by queuing commands. **Indexer LPT** accelerates into the contour through the initial commands in the queue, and decelerates to a controlled stop at the end of the queue.

When progressing from command to command, some amount of instantaneous shift in step rate is necessary in order for the motion to follow its pre-defined path. In an ideal mathematical model, an instantaneous change in velocity is impossible, since it would require an infinite force, or zero mass. However, when dealing with step motors, “instantaneous” actually means “within the time period of one step”. By design, step motors will withstand certain amounts of instantaneous shift in velocity. When **Indexer LPT** begins motion from rest, using for example the *feed* command, the step motor velocity “instantaneously” shifts from zero steps per second to the value in the *feed_lowspeed* register. When executing commands from the queue buffer, an instantaneous shift in step rate is generated while the motors are moving from one segment to another segment of a complex path.

When closely traversing a smooth contour by means of small changes in direction, relatively small instantaneous changes in step rate occur when control is passed from one *feed* command to the next. The greater the change in direction and the greater the speed, the greater the instantaneous shift in step rate. If the instantaneous shift is allowed to be sufficiently large, an axis may be over-stressed and fly out of step. When processing a queue of commands, **Indexer LPT** determines the velocity shift which can be withstood by each axis. **Indexer LPT** automatically adjusts actual velocities so as not to over-stress any axis during the traversal of a contour.

Velocity Shift

Consider the case of a direction reversal of a single axis, as demonstrated in the following command sequence:

```
q_begin
feed:a,1000
feed:a,-1000
q_end
q_go
```

In this example **Indexer LPT** decelerates the "a" axis to the *feed_lowspeed* setting at the completion of the "`feed:a,1000`" command, and immediately executes the "`feed:a,-1000`" command starting at *feed_lowspeed* velocity. Since the axis is reversing in direction, the instantaneous shift in velocity between commands is TWICE the value set up in the *feed_lowspeed* registers. Consequently, if queue processing is being used, and if the possibility of direction reversal exists, the value of the *feed_lowspeed* register must be equal to or less than ONE HALF the instantaneous velocity shift which can be withstood from an axis at rest. The particular value, of course, is governed by the mechanical dynamics of the system and the power available to the motor.

In another example, consider the situation which occurs when control passes from one *feed* command to another in multiple axes:

```
q_begin
feed:a,1000,b,1000
feed:a,8660,b,5000
q_end
q_go
```

In this example, the first *feed* command represents a motion vector which is directed at a 45 degree angle off the "a" axis. Assume that the feed rate (vector velocity) at the completion of the first *feed* command is 1000 steps per second. Each axis, therefore, is moving at a rate of 707 steps per second. The second *feed* command represents a linear motion which is directed 30 degrees off the "a" axis. In order to maintain this new path after leaving the first *feed* command, the "a" axis must instantaneously increase in velocity by 159 steps per second to 866 steps per second, and the "b" must decrease in velocity by 207 steps per second to 500 steps per second. This shift in velocity is necessary for the step motors to follow the path which is defined.

The instantaneous vector velocity at the transition point affects the magnitude of the velocity shift for each axis. It follows that if the magnitude of the vector velocity were ten times as great, the transitional shift in velocity for each axis would increase by a factor of ten. In this example, if the vector velocity at the completion of the first *feed* command were 10000 steps per second, a velocity shift of 1590 steps per second on the “a” axis and negative 2070 steps per second on the “b” axis would be necessary.

The angle between the one path of motion and the following path of motion also affects the magnitude of the velocity shift. The greater the angle, the greater the velocity shift must be.

By monitoring both velocity and vector angle, **Indexer LPT** regulates velocity throughout the contour to avoid excessive velocity shift between commands.

Adjusting Velocity Shift

Indexer LPT maintains a strategy for determining the acceptable velocity shift at the range of velocities from *feed_lowspeed* to *feed_highspeed*. **Indexer LPT** processes the queue of commands so that the acceptable velocity shift will not be exceeded during the execution of the queue.

As previously mentioned, at the starting velocity, each axis must be able to withstand a shift in frequency of two times the initial velocity (*feed_lowspeed*). However, since the torque available from a stepper motor generally decreases with velocity, the motors will typically not be able to withstand the same velocity shift at elevated velocities. The amount of velocity shift which **Indexer LPT** will permit at any given velocity is governed (in part) by the value stored in the *vshift* register.

When an axis is operating at *feed_highspeed* velocity, the amount of velocity shift which is allowed is at its minimum, and is two times the value in the *vshift* register. The amount of allowable velocity shift is at its maximum at or below *feed_lowspeed* velocity. At or below *feed_lowspeed* velocity, the amount of allowable velocity shift is two times the value of the *feed_lowspeed* register. The amount of allowable velocity shift at intermediate velocities is proportionally scaled between the maximum allowable (at *feed_lowspeed*) and the minimum allowable (at *feed_highspeed*).

In the default configuration, **Indexer LPT** selects a minimal value for *vshift* based upon the values of the *feed_highspeed*, *feed_lowspeed*, and *feed_accel* registers. The value of *vshift* may be changed by means of the *set_vshift* command. The *vshift* register will not accept values which exceed the value of the *feed_lowspeed* register.

It should also be noted that whenever the *feed_highspeed*, *feed_lowspeed*, or *feed_accel* registers are changed, a new default value for *vshift* is calculated and installed. Be careful not to set the *vshift* register before changing the values of these other registers. The value in the *vshift* register will be over-written.

The *vshift* register may be written to in whole units of steps per second. The value of the *vshift* register may be read by means of the *vshift?* command. Depending upon the values of the *feed_lowspeed*, *feed_highspeed*, and *feed_accel* registers, the default *vshift* value may be set to a fractional unit less than one. The *vshift?* command will read a fractional unit as zero. The following command may be used to set the *vshift* register to its default value:

```
set_vshift:default
```

Generally speaking, a larger value in the *vshift* register creates a reduced tendency for **Indexer LPT** to decelerate when approaching a transition point. The optimal value for *vshift* depends upon the torque which is available from the motors at operational speeds. If for example, the torque of the motors drops off substantially at the *feed_highspeed* velocity, then the value for *vshift* should be small to avoid over-stressing the axes during sharp transitions. If however, there still remains substantial torque at this velocity, a higher value in the *vshift* register may be of benefit. The contour will be more rapidly traversed and a more consistent vector velocity will be maintained.

Some systems operate at sufficiently slow speeds (or at sufficiently high torque) so that the same amount of velocity shift can be tolerated at all speeds. In such a case you may wish to set the *vshift* register to the same value as the *feed_lowspeed* register. You cannot set the *vshift* register to a value higher than the value of the *feed_lowspeed* register.

It must be emphasized the amount of “acceptable” velocity shift mentioned thus far is the amount of velocity shift which **Indexer LPT** determines acceptable according to its contouring strategy. The actual velocity shift which can be physically sustained depends on factors including friction, inertia, and motor/drive characteristics. The system designer must therefore set up the *feed_lowspeed*, *feed_highspeed*, *feed_accel*, and *vshift* registers so that velocity shifts determined “acceptable” by **Indexer LPT** fall within the physical limitations of the system.

Memory Management

When the control portion of **Indexer LPT** loads it sets aside a portion of system memory for use by the queue buffer. The amount of memory which it attempts to allocate is stored in the system Registry as either the default amount set up during installation, or

the result of the last *set_q_mem* command. The *set_q_mem* command writes to the system Registry. Consequently, **Indexer LPT** will attempt to take out the same amount of memory from Windows the next time **Indexer LPT** is run, even if the computer had powered down.

Queue buffer memory becomes occupied as commands are entered into the queue. When the queue is executed by means of a *q_go* command, or if it is emptied by means of a *q_reset* command, the entire amount of allocated memory is once more made available for more commands.

The *q_empty?* command is a convenient means of determining if the queue is empty at any particular time. The *q_mem?* command reports the amount of memory available for use in the queue buffer. If the queue is empty, the *q_mem?* command reports the size of the queue buffer.

The *command_mem?* command provides a means of determining how much memory is occupied by a specific command. Consider the following query:

```
command_mem?:feed:a,100,b,-300
```

As a result, **Indexer LPT** will fill the mailbox with an ASCII numeric string designating the amount of memory which this command would occupy if loaded into the queue buffer.

The amount of memory required by circular interpolation commands, such as *circle*, *arc_to_angle*, and *arc_to_point* depend upon how many segments **Indexer LPT** uses to approximate the geometry. Consequently, the memory demand for these commands depends upon the angle that is subtended and the value of the *arcseg_degrees* register.

Flow Control

The process of transferring commands to **Indexer LPT** in an orderly and appropriate manner is called “flow control”. Application programs, such as those available from Ability Systems, hide the complexity of flow control from the end user. However, if you are writing application programs you must understand the logic of flow control in order to obtain optimim performance from the advanced contouring features of **Indexer LPT**.

The simplest implementation of flow control is the use of the DOS “copy” command. Consider the use of the following DOS command:

```
C>copy <command filename> motor
```

In this case, DOS’s own **COMMAND.COM** routine reads the

command file and sends it to **Indexer LPT** a line at a time. Although very effective, this technique is obviously lacking in features. For one thing, it may be desirable for the operator to view the **Indexer LPT** commands on the screen as they are being passed to the device driver. In order to achieve this, a program can be written in BASIC, C, Pascal, or other language, which reads the input file a line at a time, prints the command to the screen, then sends the command to **Indexer LPT**. Such a program accomplishes a very simple form of flow control with an added display feature. Similar to DOS's "COPY", commands are sent to **Indexer LPT** exactly as they appear in the command file. Applications which require queue processing, however, may require the flow control routine to assume additional responsibilities.

Since the queue is not infinitely deep, it is possible to attempt to load the queue buffer with more commands than it can accommodate. If an attempt is made to enter a command into the queue, and insufficient memory remains to accept that command, **Indexer LPT** ignores the command and places the following message in the mailbox:

```
error,queue full
```

This is known as over-writing the queue buffer. Of course it is important not to lose commands in this manner!

One method of avoiding this condition is to provide your system with sufficient memory, and use *set_q_mem* to allocate a sufficient amount of memory to the queue buffer for the most complex contour that you would reasonably expect to service.

In some applications it may be necessary or desirable for the flow control program to monitor the amount of the memory remaining in the queue buffer as it queues each command. If the flow control routine detects that the buffer is full, it may generate an appropriate handling sequence.

The system designer must determine which method or combination of methods is most appropriate to the particular application. Examples of some flow control routines are included on the distribution diskette.

Indexer LPT provides a convenient means for the flow control routine to monitor the queue buffer. The memory requirement of each queuable command can be determined by means of the *command_mem?* query command. In response to the *q_begin* command, **Indexer LPT** reports the amount of memory remaining in the queue buffer. **Indexer LPT** similarly reports the remaining buffer memory as subsequent commands are queued. The flow control routine can thereby determine if room exists in the buffer before issuing each command.

What does the flow control routine do when it determines that the next command will not fit in the queue? This depends upon the particular application and/or what type of operation is involved.

One of the most elementary things which the flow control routine can do is to immediately execute the queue to free up the queue buffer. To accomplish this, the flow control routine issues a *q_end* command followed by a *q_go*. This sequence processes the current queue, executing instructions up to and including the last command. When execution of the queued commands is complete and the queue buffer is again ready to receive more commands, the flow control program issues a *q_begin* command and proceeds to read subsequent commands from the command file and write them into the queue. When the last command is read and queued, the flow control program issues a final *q_end* and *q_go*.

This strategy will avoid losing commands due to buffer over-write. However, recall that the principle reason queue processing exists is to uniformly control velocity across the path of a contour. To stop at an arbitrary point and delay for the additional time necessary to load the queue defeats the purpose of queue processing. The following example shows how this apparent anomaly is overcome by means of a flow control strategy.

Consider as an example an operation where a contour is being traversed using “a” and “b” axes, and a tool is lowered into the work with the “c” axis. Queue processing is being used to improve the quality of the process by providing a more uniform vector speed across the contour. However, if the tool is engaged in the work and not advancing it risks leaving a dwell mark. In this example it is acceptable to withdraw the tool from the work, process the queue, re-engage the tool into the work, then execute the queue. To accommodate this strategy, **Indexer LPT** allows non-queued commands to be executed between the *q_end* command and the ensuing queue execution (*q_go*). The listing below demonstrates a sequence of instructions as they may be received by **Indexer LPT** from a flow control routine. It is important to notice that the dwell time necessary to load the queue occurs when the tool is withdrawn.

```
# Begin queuing with cutter withdrawn
q_begin
.
# Queue-ed commands - define the contour
.
q_end
# Engage the cutter into the work
feed:c,-400
# Execute the queue - smooth contouring
```

```
#operation
q_go
# Withdraw the cutter from the work
move:c,400
# Begin queuing with cutter withdrawn
q_begin
```

In this application, the flow control routine queues commands with the cutter withdrawn and monitors the memory available as each command is read from the command file and queued. When the buffer is full, the flow control routine generates the *q_end* command to process the queue. The flow control routine then generates:

```
feed:c,-400
q_go
feed:c,400
q_begin.
```

Thus by monitoring the amount of memory in the queue buffer as commands are queued, a flow control program can avoid over-writing the queue while implementing a sequence of operations which virtually eliminates cutting tool dwell.

The best strategy which is adopted in flow control depends on the particular application. The strategy which you decide on for your system is likely to involve a combination of the principles which have been discussed. Command sequences and flow control routines are provided on the distribution diskette to demonstrate these principles and to assist you in constructing your own flow management software.

Safety Concerns

In motion control systems where human intervention can be hazardous, it is important to protect the operator against unexpected motion. When working with contours that load a great number of segments into the queue buffer, the time which it takes to load the queue may give the appearance that the machine is dormant. It is important in these cases to provide some means by which the operator is alerted that the computer is processing, and that machine motion is impending. The method that is used depends upon the particular machine. For example, for some machines a notification placed on the computer screen may be sufficient. For other machines, an annotation light and buzzer located in physical proximity to the hazardous area may be necessary. Other machines may require shield and lockout mechanisms, physically preventing oper-

ator intervention until an operation is complete. The method which is used is up to the machine designer. Safety should be your primary concern.

Speed Control

It is possible to change the speed of the contouring motion within the queue under software control by means of features provided in the feed rate override (FRO) control. (Refer to the chapter entitled **Feed Rate Override**).

Specifically, the *set_fro_offset* command is acceptible for use within the queue buffer. Its use in continuous control over contouring speed can be illustrated in the following example.

This example demonstrates a smooth contour into and out of an arc. Once entering into the arc, motion decelerates to 80% of the feed rate. After leaving the arc, motion accelerates to resume at 100%. (Please note that in this example a certain amount of additional slowing might “naturally” occur due to the effect of *vshift*).

```
q_begin
feed:a,10000
set_fro_offset:-20
arc_to_angle:ccw,a,0,b,4000,90
set_fro_offset:0
feed:b,10000
q_end
q_go
```

The default value of the *fro_offset* register is 0. Consequently, the first *feed* command executes at 100% of the setup parameters (*feed_lowspeed*, *feed_highspeed*, *feed_accl* and *vshift*). As soon as the first *set_fro_offset* command is executed within the queue, the FRO set point is changed to 100%-20%= 80%. Motion timing tracks the FRO set point, decelerating to 80% of the established feed rate as it enters the arc. The second and following *set_fro_offset* command restores the *fro_offset* register to its default value, causing subsequent motion to proceed at 100%.

Please also note that if the hardware portion of FRO has been enabled, the effects of FRO hardware combine as per the diagram in **Figure 7-1**.

“On the Fly” Digital Output

Digital output commands *winding_power*, *reduced_current* and *bit* can be executed from within the queue buffer while motors are in

motion.

This feature is especially useful in machines that must activate a particular opearation at a point where motion occurs at predictable rate, having completed the acceleration cycle.

Consider, for example, a machine that uses motion control to spray a coating along a path on a workpiece at a carefully controlled density. Since the speed of the spray nozzle affects the density of the coating, it is important to activate the nozzle at a particular point after the motors have accelerated to vector rate, and to deactivate the nozzle at a point before decelerating to a stop. To accomplish this, the nozzle must be activated while the motors are moving, which is to say, “on the fly”.

During a queue loading sequence, one of the digital output commands can be "attached" to the end of each motion command. For example, consider the sequence:

```
q_begin
feed:a:100
reduced_curent:b,0
feed:a,250
q_end
q_go
```

In this example the *reduced_current* command is attached to the end of the first *feed* command. When the queue is executed by means of the *q_go* commmand, the *reduced_current* command will be executed after the first *feed* command. Motion control is continuously processed from *feed* command to *feed* command and smoothness of motion is not affected by the introduction of the *reduced_current* command.

After the queue buffer is opened by means of *q_begin*, if a motion command does not exist in the queue buffer, the digital output command will be processed immediately. For example, in the sequence:

```
q_begin
reduced_current:b,0
feed:a,100
feed:b,200
q_end
q_go
```

... the *reduced_current* command will not be suspended until the

queue is executed via *q_go*. In this case the *reduced_curent* command will be executed immediately.

Only one digital output command can be attached to the end of a motion command. If consecutive digital output commands are attached to any motion command, only the last one will have effect. For example, in the sequence:

```
q_begin
feed:a,200,b,100
reduced_current:b,0
winding_power:a,1
feed:b,200
q_end
q_go
```

... only the *winding_power* command will be attached to the end of the first *feed* command. The *reduced_current* command will have no effect.

Chapter 9

SWITCH SCANNING & JOYSTICK

The Scanning Feature

Switch scanning is a powerful feature which enables a relatively large number of switches to be sensed using a limited number of inputs and outputs. The procedure comprises a few simple steps. The scanning input line is "pulled high", that is, it is connected to 5 Volts through a series resistor. (The limit switch inputs satisfy this criteria internally). When an open circuit condition exists on a scanning input line, the input senses a 5 volt condition.

The scanning output lines are normally high (5 Volts). **Indexer LPT** accomplishes scanning in the following manner:

1) A specified output line is momentarily brought low (0 Volts).

2) During the time period when the specified output line is low, the specified input line is sensed. The result is placed in the mailbox: `1` if the input is low (switch closed), `0` if the input is high (open circuit).

3) The output line which was momentarily brought low is returned to its normal high condition.

If the circuit connecting the input line is open, the input line will be high when sensed. However, if the switch which connects the specified output line to the input line is closed during the time period when the output line is low, the input line will be sensed to be low. When **Indexer LPT** reads a low signal on the input line during the scan cycle, it concludes that the connection has been made by means of a switch closure.

It is important to note that only one of the scanning outputs can be low at any given time. Note also that the outputs which are high "appear" as an open circuit to the input. The particular switch closure is determined by reading the input when a known output is in the low condition. The term "scanning" applies to the practice of testing for switch closures by performing this operation sequentially on each of a number of output lines.

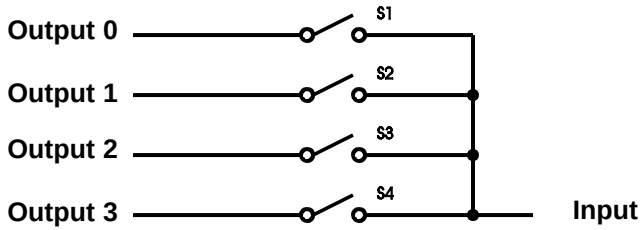


Figure 9-1

The diagram in **Figure 9-1** helps to show how four switches can be sensed by scanning four outputs into one input. All outputs are normally high. An open circuit on a *limit switch input* reads high, since it is internally pulled up.

(Caution: Since on most printer cards the *auxiliary input* is a floating TTL input, it cannot be relied upon to read "high" during a open circuit condition without a pull up resistor. To use the *auxiliary input* as a scanning input, connect the line to 5 volts through of a 4.7K ohm resistor).

The scanning procedure is summarized as follows:

The procedure begins by holding **Output 0** low and reading the **Input**. If the **Input** is low, then it is concluded that switch **S1** is closed. After the input is read, **Output 0** is returned to its normal high condition.

Next **Output 1** is held low and the **Input** is read. If the **Input** is low, then it is concluded that switch **S2** is closed. **Output 1** is then returned to its normal high condition.

This procedure is repeated to scan **S3** using **Output 2**, and **S4** using **Output 3**.

Notice that the circuit in Figure 9-1 can only be used for scanning if only one switch is closed at any given time. If more than one switch is closed, an unacceptable short circuit condition will exist between the scanning outputs which are connected to the simultaneously closed switches. In some applications, it is possible to

physically prevent two switches from simultaneously closing. In other applications, it cannot be prevented.

A diode in series with each switch can be used to isolate each of the outputs, thereby allowing the condition of simultaneous switch closures. This technique takes advantage of the fact that current flows from the input into the output when the output is low. The diode prevents current from passing from the high outputs into the output which is being scanned low, essentially making the high outputs behave as open circuits regardless of the condition of their respective switches.

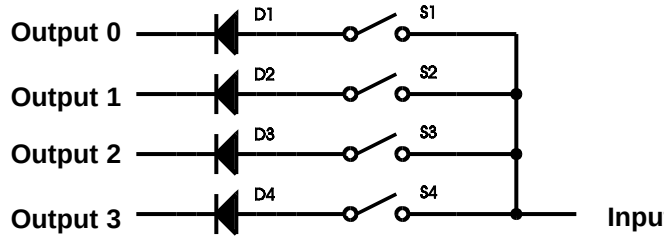


Figure 9-2

The diagram in **Figure 9-2** shows a circuit which uses diodes to isolate the outputs in a scanning circuit. **Figure 9-3** shows the schematic representation of the diode next to a picture of the physical part. Small germanium diodes such as the 1N34A are recommended. These are available through a number of electronic suppliers, including Radio Shack (Radio Shack part number for a pack of ten is 276-1123). They are small, cylindrical in shape and about 3/16" long. The leads are usually .022" in diameter and extend about 1". The cathode end is marked by a band which is painted or etched around the body of the part.

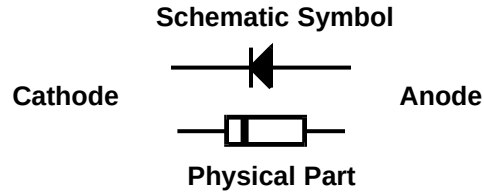


Figure 9-3

The diagram in **Figure 9-4** shows how eight switches can be wired for scanning using four outputs and two inputs. Additional switch-

es can be scanned by increasing the number of scanning outputs or by increasing the number of inputs. The number of switches which can be scanned is determined by multiplying the number of scanning outputs by the number of available inputs.

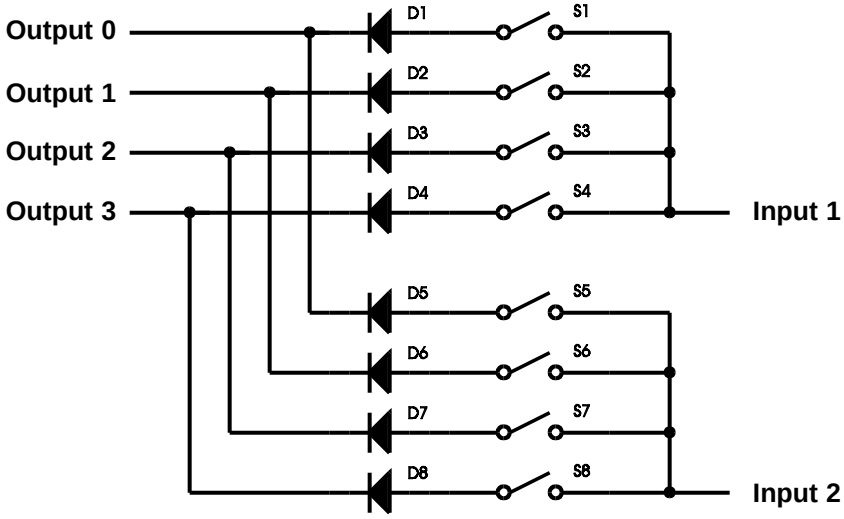


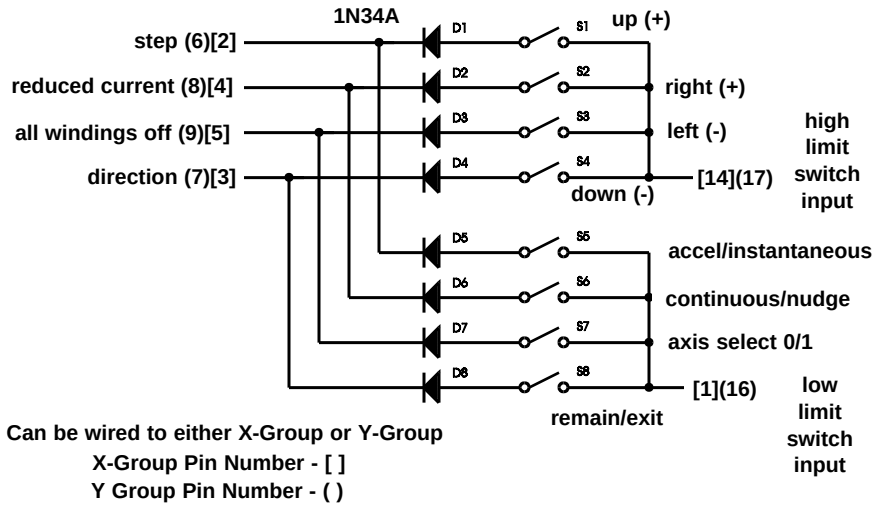
Figure 9-4

By setting the *mode* of an axis using the *axis* command, **Indexer LPT** allows the four outputs associated with an axis to be allocated for special use in scanning. When scanning joystick switches, the limit switch inputs associated with the scanning axis are used as inputs. However, when using the *scan* command to sense discrete switch closure, any of the available inputs may be used as the scanning input.

The Joystick Feature

The joystick feature allows an axis to be set up and used for joystick control. The wiring diagram in **Figure 9-5** shows a typical full joystick configuration. The wiring diagram in **Figure 9-6** shows a scaled down (minimum features) joystick implementation. The pin numbers used for making the actual connections to the printer connector can be determined by reference to the axis addressing chart in the chapter entitled HARDWARE REQUIREMENTS.

It should be noted that in configuration options where switches are momentary action, and more than one switch physically cannot be closed simultaneously, diodes are not required



Wiring for Joystick Switches and Options

Figure 9-5

Joystick Switch Assignments

S1 - S4 Joystick Switches

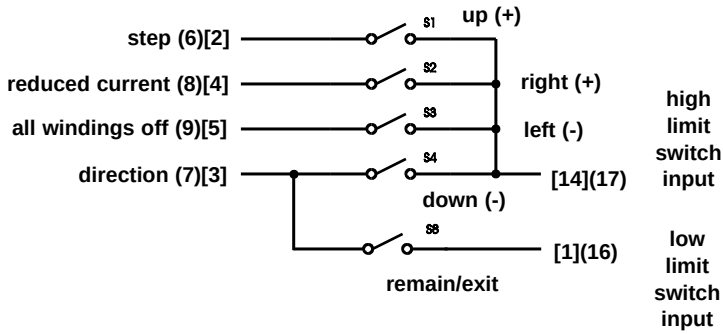
(momentary action normally open switches) These switches are used to actually control motor movement. Switches **S1** and **S4** control one axis. Switches **S2** and **S3** control another. When active, switches **S1** and **S2** always effect motion in the (+) direction. Switches **S3** and **S4** are used to activate motion in the (-) direction. The particular axes which these switches control is determined by the *joystick_input* setup command and the status of switch **S7**.

These switches may be part of a single joystick switch, with a common connection to the appropriate high limit switch input.

Another useful configuration is to locate four momentary push-button switches in a diamond shape pattern on a control pattern.

S5 Acceleration Select Switch

(toggle switch) When this switch is open, the axis which is being moved by the joystick will be governed by the values set up in its *lowspeed*, *highspeed* and *acceleration* registers. When this switch is closed, joystick controlled motion will occur at a constant (instantaneous) speed according to the value set up in its *jogspeed*



Wiring for Joystick Without Options

Figure 9-6

register.

Accelerated motion, obtained when this switch is open, is useful for rapid positioning. Constant speed motion, obtained when this switch is closed, is helpful in accurate positioning.

S6 Extent Select Switch

(toggle switch) The setting of this switch determines if the axis being controlled is to send out a continuous stream of pulses, or if it is to send out a burst of one or more pulses. (The actual number of "burst" pulses is user defined by means of the *joystick_input* set-up command). When this switch is open a joystick switch closure (**S1**-**S4**) will maintain motion as long as the joystick switch remains closed. When this switch is closed, a joystick switch closure will cause motion by means of a burst of pulses. If the joystick switch remains closed after the .5 second delay, another burst of pulses is issued. Motion is immediately terminated when the joystick switch contact opens. Control options obtained using this switch are useful in very accurate positioning applications where the user can only move the axis close to the desired position with continuous motion. Final positioning is accomplished by"nudging" the motor to its final position a step or a few steps at a time.

S7 Axes Select Switch

(toggle switch) Up to four axes can be controlled by means of a single joystick. This switch selects which axes (in sets of two) the joystick switches are to control. The second argument in the *joystick_input* command associates the axes which are to be controlled with the position of this switch. A value of `0` in the second argument of the *joystick_input* command defines the behavior of the joystick switches (**S1**-**S4**) when **S7** is open. A value of `1` in the sec-

ond argument of the *joystick_input* command defines the behavior of the joystick

Another way to understand the function of **S7** is to think of it as selecting one of two *joystick_input* setups. When the switch **S7** is open, the `0` setup is selected. When the switch S7 is closed, the `1` setup is selected. In one application **S7** can be used to select between two sets of two axes, giving the joystick switches control over four independent axes.

In another application, the same two axes can be defined by the `0` and `1` *joystick_input* setup, yet with different values for the number of burst pulses. In this application, S7 is essentially being used to select the resolution of the burst mode.

S8 Exit Switch

(momentary action normally open switch) This switch enables the user to exit the joystick control mode. When the *joystick_go* command is issued, **Indexer LPT** will continually scan switches **S1** through **S8**. A switch closure on **S8** will cause **Indexer LPT** to discontinue joystick scanning and return to the calling program.

It should be noted that the joystick control switches and switch **S8** are the only additional switches required for joystick operation. If switches **S5**, **S6**, and **S7** are eliminated, the joystick feature will function as if these switches were installed but left in the open position.

Software Setup

"Quick Start" Setup

Once the hardware is configured, setting up **Indexer LPT** for joystick control is a relatively simple matter. The easiest way to understand the software setup is by means of example.

Consider a motion system where the "e" axis is to be used for joystick switch scanning. Axes "a" and "b" are to be controlled by the joystick when switch **S7** is open. Axes "c" and "d" are to be controlled by the joystick when switch **S7** is closed. When the joystick is configured to "nudge" by closing **S6**, the desired number of burst steps on all axes is to be 10.

The complete software setup is as follows:

```
axis:e,2
joystick_input:e,0,a,10,b,10
joystick_input:e,1,c,10,d,10
```

To activate the joystick command mode, the following command is issued:

```
joystick_go
```

To leave the joystick mode, close the contacts of switch **S8**. (Note that the ONLY way which **Indexer LPT** uses to leave the joystick command mode is by means of a closed circuit through switch **S8**. Therefore, make sure that this switch wiring is installed before trying the *joystick_go* command).

Since the "e" axis is to be used for a scanning purpose, the mode of that axis must first be changed by issuing the following "axis" setup command:

```
axis:e,2
```

The argument `2` is used to designate the scanning mode.

Next, using the *joystick_input* command the joystick switch response is defined for the condition where selector switch **S7** is open.

```
joystick_input:e,0,a,10,b,10
```

The first argument, "e", specifies that the "e" axis is to be used as the joystick input.

The second argument, `0`, specifies that this *joystick_input* command is being used to define joystick control for the condition where **S7** is in the open position.

By calling out for the "a" axis in the next argument, this command establishes control over the "a" axis using switches **S1** and **S4**. (Closing **S1** will cause the "a" axis to move in the (+) direction. Closing **S2** will cause the "a" axis to move in the (-) direction).

The third argument ,`10`, specifies that a maximum of 10 pulses are to be dispensed at a time when the "a" axis is being controlled and **S6** is closed.

The fourth argument, `b`, establishes control over the "b" axis by switches **S2** and **S3**. (Closing **S2** will cause the "b" axis to move in the (+) direction. Closing **S3** will cause the "b" axis to move in the (-) direction).

The last argument, `10` specifies that a maximum of 10 pulses are to be dispensed at a time when the "b" axis is being controlled and **S6** is closed.

If it is desired that the "b" axis is to be controlled by switches **S1** and **S4**, and the "a" axis is to be controlled by switches **S2** and **S3**, then the following *joystick_input* command would be is used:

```
joystick_input:e,0,b,10,a,10
```

The first axis called out in the *joystick_input* command is always the axis which is controlled by scanning switches **S1** and **S4**. The second axis called for is always the axis which is controlled by scanning switches **S2** and **S3**.

All arguments in the *joystick_input* command must be present. If only one axis is to be controlled, "none" should appear in the appropriate fields. For example, suppose in this setup only the "b" axis is to be controlled by switches **S1** and **S4**. The *joystick_input* command is constructed as follows:

```
joystick_input:e,0,b,10,none,none
```

Finally, the following command is used to define joystick operation when switch **S7** is in the closed position:

```
joystick_input:e,1,c,10,d,10
```

Notice that the "e" axis is again specified as the axis which is used to scan the joystick switches. The second argument, `1`, specifies that the following arguments apply only when switch **S7** is in the closed position. The "c" axis is controlled by switches S1 and **S4**, and will move 10 steps at a time when switch **S6** is closed. The "d" axis is controlled by switches **S2** and **S3**, and will also move 10 steps at a time when switch **S6** is closed.

Chapter 10

SPECIAL CONSIDERATIONS

One of the advantages of **Indexer LPT** is that the computer itself assumes the role of the indexer as it takes control of the translator driver. As such, the user does not need to purchase separate indexer electronics. The electronic signals necessary for motion control are generated directly from the computer's printer adapter. Since **Indexer LPT** is a device driver, communication with the user program is extremely fast and easy. There are, however, some inherent limitations which require special considerations.

Computer Occupation

In order to maintain close control over rapidly applied signals, **Indexer LPT** assumes total control over the computer while the motor is in motion. In other words, when the user program instructs a motor to move, it will wait until the motion is complete before other processing can continue. Applications which must proceed concurrently with stepper motor motion cannot use **Indexer LPT**. For these applications, we recommend using separate indexer electronics which can operate independently from the computer's CPU (such as a second computer running **Indexer LPT**).

Computer Speed

The maximum stepping rate which can be obtained is determined by the execution speed of the host computer. When **Indexer LPT** first runs, it determines the speed of the computer. The maximum stepping rate for your computer may be obtained by means of the *max_speed?* command.

Use of the System Clock

In normal operation, the computer is interrupted every 55 milliseconds by the system's timer. This interrupt causes the system BIOS to execute a small procedure that updates the memory locations which track the date and time of day. In order to keep this interrupt from interfering with the tightly controlled step timing, **Indexer LPT** turns the interrupt off just before motion begins. After the motion is finished, the interrupt is restored, and time tracking once again commences.

Since the system timer is ineffective during motor movements, some time-tracking is temporarily lost. **Indexer LPT** adjusts the system time value after each motion command by reading the time from the battery maintained real time clock, and adjusting the system clock accordingly.

Unexpected Motion - Safety Considerations

If the motion controlled components of the machine you are designing can potentially cause damage or injury to the operator, you must build into your system appropriate safeguards. We feel that these safeguards should include design considerations as well as operator training.

If it is impossible or not feasible to safe-guard your system, do not use this product and seek an alternative solution.

Almost all machinery can potentially cause personal injury, either by means of airborne debris, hazardous cutting surfaces, or other objects in motion. It is the responsibility of the machine designer to incorporate features into the machine which avoid needlessly hazardous conditions. We feel that unexpected motion of computer controlled components comprises a needlessly hazardous condition.

As the reference implies, "unexpected motion" occurs when the machine operator is taken by surprise, and a machine component moves in a manner which is potentially hazardous. The machine designer should take into consideration the following potential sources of unexpected motion which are peculiar to PC controlled machine tools.

Accidental Start

The use of a single keyboard keystroke or mouse snap presents a safety hazard on certain types of equipment, since these actions can easily occur inadvertently. Applications software written by Ability Systems incorporate methods to reduce the chances of an accidental start, and are strongly suggested as guidelines for third party and user written applications. These methods include the following

very strong suggestions for application programmers:

1) The application program should never allows motor movement as a result of a single mouse snap. If a mouse snap must be used, the menu or dialog that could cause a potentially hazardous motion should time out in an appropriately short amount of time, so that the cursor could never be dangerously poised over a selection that could be inadvertently chosen by an accidental mouse snap.

2) If the keyboard is used to effect motion (keyboard "start"), always require at least two keys to be simultaneously depressed. Using only **Enter** to initiate a control sequence is especially dangerous.

3) When monitoring external start switches, the application program should always make sure that the switch is deactivated before acknowledging that the switch is activated. This will assure that the program does not "run away" if the start switch is accidentally activated when the program or menu is invoked. Applications that require two start switches for safety can use this technique to make sure that the second switch has not been defeated (permanently activated by means of a weight or rubber band) for operator "convenience".

Windows Multi-Tasking

Application programs that communicate to **Indexer LPT** run as a Windows task. Typically, the task runs when the window that it is operating from is in focus. Unless special provision is made, when the user snaps over another window, and the task communicating with **Indexer LPT** loses focus, communication to **Indexer LPT** is suspended.

In most cases, suspending communication to **Indexer LPT** doesn't present a safety problem. In the case of a cutting tool, the tool may be suspended in its present position and cause a burn mark. But this seldom presents a threat to safety. However, a seriously dangerous condition could present itself when the task is brought back into focus, and communication with **Indexer LPT** resumes.

Consider how easy it would be to accidentally bring the application into focus. If the screen cursor is accidentally positioned over the dormant application and the mouse button is snapped, unexpected motion may occur.

In some types of machines it may be acceptable to simply run the application program in full screen mode, thereby reducing the possibility of accidentally switching out of, and later into the program. In more hazardous machines, such as cutting tools, it is recommended that your application run "System Modal".

Application programs that run "System Modal" remain in focus

until an orderly procedure is used to switch focus. During this orderly procedure, your program can make sure that communication of motion commands to **Indexer LPT** is not pending.

Computational Delay for Complex Shapes

Even though the calculation dwell for each element of a complex shape is very small, with the ability to handle extremely large and complex geometric shapes (comprising the equivalent of thousands of *feed* commands), it is possible to generate extensive cumulative calculation delay times. A potentially hazardous condition occurs when the delay time exceeds the time it would take for an operator to inadvertently perceive that the machine is dormant, and interfere with the operation in a manner that would endanger himself or others, such as breaking down a setup fixture and placing his hand in the path of a cutting tool. In machines where this type of occurrence would compromise safety, appropriate safeguards must be designed into the system.

One type of safeguard may be a simple warning light and corresponding annotation (e.g. illuminated sign), which warns the operator that the machine is in operation, and that the machine may begin motion at any time. The application program can control the illumination of a light by means of the **Indexer LPT** digital output lines, such as the ones entitled *reduced current*, or *all windings off*.

An audible buzzer can also be controlled while the machine is in operation, which may be used in combination with a visible warning. Some machines may require a solenoid actuated latch, which would physically prevent operator intervention until the program is finished.

If you consider the operation very hazardous, you may design your software so that motion never occurs automatically after any calculation dwell, and always requires an operator to manually actuate a start condition. This procedure may be organized as follows: 1) Load the queue buffer, starting the loading process with *q_begin*, followed by the motion commands, in turn followed by *q_end*. This portion of the procedure can potentially occupy the most time. 2) Alert the operator that the machine is ready to commence motion. 3) Monitor a manually actuated start condition, such as one or two pushbutton activated signal inputs.

Chapter 11

COMMANDS

ABORT?

Synopsis:

Return the logical condition of the abort switch.

Syntax:

```
abort?
```

Returns:

The mailbox contains an ASCII numeric character:

`1` *abort* input is activated.

`0` *abort* input is not activated.

The abort input is “activated” when there exists a circuit path to ground on the *low limit switch* input pin (TTL low) of the axis designated to support the *feed hold* feature. The abort switch is “not activated” when an open circuit condition exists on this pin (TTL high).

If this command is issued and the *feed hold* feature has not been enabled, the following message appears in the mailbox:

```
error, disabled
```

Example:

Assume the “d” axis has been specified to support the *feed hold* feature by means of the following command sequence:

```
axis:d,1
feedhold_input:d,1
```

Also assume the *low limit switch input*, which now serves as the *abort* input, is connected to an open circuit, and is therefore at TTL high potential. In this example, this input is connected to a normally open switch circuit to ground. When the switch is open the *abort* input is “de-activated”. The following command is issued:

```
abort?
```

As a result, the following ASCII character appears in the mailbox:

```
0
```

If the normally open switch in this example is closed, making a circuit path to ground, and thereby applying a TTL low potential to the *abort* input pin, the following ASCII character appears in the mailbox following the *abort?* command:

```
1
```

ACCEL?

Synopsis:

Read the contents of the *accel* register.

Syntax:

```
accel?:<axis>
```

Returns:

An ASCII numeric string designating the acceleration setting of the selected axis in steps per second per second is placed in the mailbox. This value is either the default value, or the value which was set by the *set_accel* command.

Example:

Assume the value of the *accel* register for the “c” axis has been previously set to 495. To make this value available to be read from the mailbox the following command is issued:

```
accel?:c
```

As a result, the following string of ASCII characters is placed in the mailbox:

```
495
```

ARCSEG_DEGREES?

Synopsis:

Read the contents of the *arcseg_degrees* register.

Syntax:

```
arcseg_degrees?
```

Returns:

An ASCII numeric string designating the value of the *arcseg_degrees* register appears in the mailbox.

Example:

Assume you have just set the value of the *arcseg_degrees* register to 2 using the following command:

```
set_arcseg_degrees:2
```

After writing the following command to **Indexer LPT**:

```
arcseg_degrees?
```

the following message appears in the mailbox:

```
2
```

ARC_TO_ANGLE

Synopsis:

Traverse an arc to the designated angle around the specified center point. Optionally traverse up to two additional axes proportionately to the subtended angle (helical interpolation).

Syntax:

```
arc_to_angle:<direction>,<axis>,
      <steps to cp>,<axis>,<steps to cp>,
      <angle>
      [,<axis>,<steps>[,<axis>,<steps>]]
```

Only positive values for `<angle>` are accepted. Decimal angles are permissible.

The direction in which the arc is drawn, clockwise or counterclockwise, is determined by the first command argument, `<direction>`, and the order in which the axes appear on the command line. “Clockwise” and “counterclockwise” are perceived in three dimensional space by “wrapping” the first axis into the second axis with the fingers of the right hand and viewing the circular motion from the direction of the thumb.

`<direction>`	**Meaning**
`cw`	arc is subtended clockwise
`ccw`	arc is subtended counterclockwise
`<steps to cp>`	Signed magnitude of steps from the present position to the position of the center point
`<angle>`	Whole number or decimal value of angle to be subtended
`<axis><steps>`	Optional arguments for helical interpolation

Side Effects:

The arc or helix is approximated by linear segments. The segment angle is determined by the *arcseg_degrees* register. Linear segments approximating the geometry are loaded into and executed from the queue buffer. Motion along the paths of the interpolation are governed by setup parameters in the *feed_lowspeed*, *feed_highspeed*, and *feed_accel* registers.

If a queue load sequence has been initiated with the *q_begin* com-

mand, the segments comprising this command are appended to the end of the queue buffer.

If a queue load sequence has not been initiated, then this command automatically begins the queue load sequence, loads the interpolation segments into the queue buffer, closes the sequence, and executes the motion.

In certain applications this command must be used according to user safety considerations, as demonstrated in the fourth example below.

If a limit switch closure of a member axis is detected in an associated direction of movement, all motion is terminated and position tracking on all axes associated with the motion is lost.

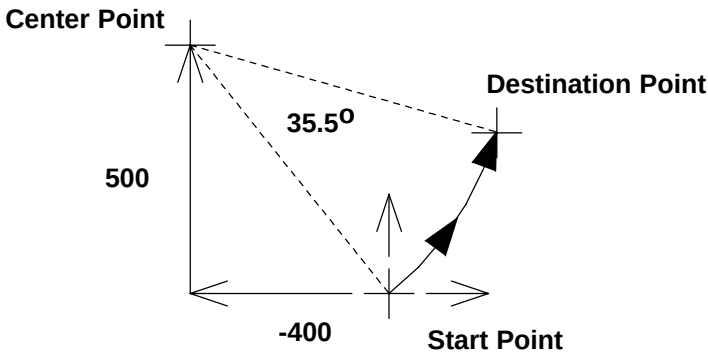
Returns:

An ASCII string representing the final position of each axis is placed in the mailbox. Position information concerning each axis is separated by a colon and appears in the order in which the axes are called out on the command line.

Example:

To draw an arc in the counterclockwise direction using a center point located minus 400 steps in the “a” direction and 500 steps in the “b” direction, and subtending an angle of 35.5 degrees the following command is issued:

```
arc_to_angle:ccw,a,-400,b,500,35.5
```



Example:

Assume the present position of both “a” and “b” axes is zero and the following command is issued.

```
arc_to_angle:ccw,a,0,b,1000,270
```

After traversing this arc to an angle of 270 degrees, the following message is available to be read in the mailbox:

```
-1000:1000
```

This message represents the final positions of the “a” and “b” axes respectively. If in this example **Indexer LPT** senses a closure of the high limit switch of the “b” axis while moving the “b” axis in the positive direction, the arc is immediately terminated and the following message would appear in the mailbox:

```
limit,b,+
```

Position tracking on all axes associated with the motion would be lost.

Example:

Consider the execution of the arc in the previous example with additional simultaneous motion of the “c” and the “d” axis, where the “c” axis travels 500 steps, and the “d” axis travels -300 steps. The command to execute this motion is as follows:

```
arc_to_angle:ccw,a,0,b,1000,270,c,500,d,-300
```

Example: - Safety Consideration

Consider the following command which follows the path of a complete circle 100 times in the “a” and “b” axes, while simultaneously advancing the “d” axis 10000 steps:

```
arc_to_angle:ccw,a,0,b,1000,36000,c,10000
```

Assuming the value of the register controlled by the *arcseg_degrees* command were the default value of 5 degrees, **Indexer LPT** would approximate the helix defined by this command with 36000/5 = 7,200 linear segments. Assuming your computer has enough available memory to handle this large an object, and assuming you allocated enough memory in the queue buffer with the *set_q_mem* command, **Indexer LPT** would execute this command immediately after loading the segments into the queue buffer.

This example demonstrates a command that may take considerable time to process before motion commences. If the time it takes presents a safety hazard, where you think an operator may be endangered by unexpected motion, your application program can split the operation of calculation from the operation of executing the command using the *q_begin*, *q_end*, and *q_go* commands, as demonstrated in the following sequence.

```
# Open the look-ahead buffer
q_begin

# Load without executing the motion
arc_to_angle:ccw,a,0,b,1000,36000,c,10000

# Close the look-ahead buffer
q_end

# Monitor a start switch before proceeding
[Application Program waits for an input]

# Execute the command
q_go
```

ARC_TO_POINT

Synopsis:

Traverse an arc defined by the present position, the specified center point, and the specified destination point. Optionally traverse up to two additional axes proportionately to the subtended angle (helical interpolation).

Syntax:

```
arc_to_point:<direction>,
      <axis>,<steps to cp>,
      <steps to destination>,
      <axis>,<steps to cp>,
      <steps to destination>
      [,<axis>,<steps>[,<axis>,<steps>]]
```

The direction in which the arc is drawn, clockwise or counter clockwise, is determined by the first command argument and the order in which the axes appear on the command line. “Clockwise” and “counterclockwise” are perceived by “wrapping” the first axis into the second axis with the fingers of the right hand and viewing the circular motion from the direction of the thumb.

<direction>	**Meaning**
cw	arc is subtended clockwise
ccw	arc is subtended counterclockwise
<steps to cp>	Signed magnitude of steps from the present position to the position of the center point
<steps to destination>	Signed magnitude of steps from the present position to the final destination point
<axis><steps>	Optional arguments for helical interpolation

Side Effects:

The arc is traversed along the specified axes according to motion parameters set up in the *feed_lowspeed*, *feed_highspeed*, and *feed_accel* registers. If a limit switch closure of a member axis is detected in an associated direction of movement, all motion is terminated and position tracking is lost.

The radius of the arc is determined by the distance between the present position and the center point. **Indexer LPT** traverses the arc to the angle which is determined by a radial projection extending from the center point to the destination point. If the destination point does not lie on the arc, **Indexer LPT** traverses the best straight line from the completed angle to the destination point.

The arc or helix is approximated by linear segments. The segment angle is determined by the *arcseg_degrees* register. Linear segments approximating the geometry are loaded into and executed from the queue buffer.

If a queue load sequence has been initiated with the *q_begin* command, the segments comprising this command are appended to the end of the queue buffer.

If a queue load sequence has not been initiated, then this command automatically begins the queue load sequence, loads the interpolation segments into the queue buffer, closes the sequence, and executes the motion.

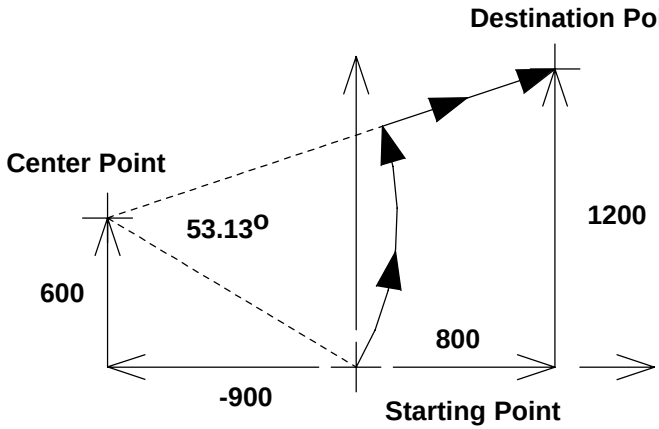
Returns:

An ASCII string representing the final position of each axis is placed in the mailbox. Position information concerning each axis is separated by a colon and appears in the order in which the axes are called out on the command line.

Example:

Assume the present position of both “a” and “b” axes is zero and the following command is issued:

```
arc_to_point:ccw,a,-900,800,b,600,1200
```



An arc with a radius of 1082 steps is traversed to an angle of 53.13 degrees. From the location on the arc of 53.13 degrees, **Indexer LPT** moves to the destination point as if instructed by a feed command.

After completion, the following message appears in the mailbox:

```
800:1200
```

This message represents the positions of the “a” and “b” axes respectively. If in this example **Indexer LPT** senses a closure of the high limit switch of the “b” axis while moving the “b” axis in the positive direction, the arc is immediately terminated and the following message would appear in the mailbox:

```
limit,b,+
```

Position tracking on all axes associated with the motion would be lost.

Example

This example traverses an arc in the “a” and “b” axes, while simultaneously moving the “c” and “d” axes to their destination locations along a trajectory that is proportional to the angle being subtended:

```
arc_to_point:ccw,a,0,100,b,100,100,c,50,d,75
```

If all axes started from the home position, the following message, reporting the positions of the axes that were moved, appears in the mailbox:

```
100:100:50:75
```

AUX_INPUT?

Synopsis:

Determine the logic level of the *auxiliary input* signal lines.

Syntax:

```
aux_input?:<axis>
```

Side Effects:

Each axis is provided with an auxiliary TTL level input line which may be read by means of this query command. Unlike the limit switch input lines which are internally pulled high, this input must be driven to its logic level by an external source.

Returns:

The mailbox contains an ASCII numeric character:

`0` (zero) if the input voltage level is TTL low.

`1` (one) if the input voltage level is TTL high.

Example:

An external circuit is driving the voltage of the *auxiliary_input* pin of the “c” axis high when the following command is issued:

```
aux_input?:c
```

As a result, the following character appears in the mailbox:

```
1
```

AXIS

Synopsis:

Select whether the outputs associated with the designated axis are to be used for motor control, for discrete digital output, or for switch scanning

Syntax:

```
axis:<axis>,<mode>
```

<mode>	Meaning
0	Axis is used for motor control
1	Axis is used for discrete digital output
2	Axis is used for switch scanning
(3	Axis is reserved for the *feed rate override* feature)*

Side Effects:

The default mode of each axis accommodates the use of its associated output signals for motor control. By changing the mode using the *axis* command, the output signals can be used for simplified discrete output control using the *bit* command, or for joystick and switch scanning operations using the *scan* and *joystick* commands.

Once the mode of an axis is changed, the limit switch inputs of that axis may be assigned for special use associated with the *feed hold* features using the *feedhold_input* command.

* “Mode 3” is assigned by **Indexer LPT** to the axis that is associated with the *feed rate override* hardware. Consequently, you cannot use this command to change an axis to or from “mode 3”.

Returns:

After changing the mode of an axis using this command, the axis may be restored to its original mode once at a later time in the same manner. If, however, the limit switches are currently being used for the *feed hold* feature, the *axis* command will not revert the axis back to the motor control mode and the following message will appear in the mailbox:

```
error,feedhold is enabled
```

(Otherwise, if the new value is accepted, an ASCII string representing the new mode for the axis will appear in the mailbox).

Consequently, the *feed hold* feature must be disabled using the following command:

```
feedhold_input:<axis>,0
```

before the axis command will permit the mode to be changed in this instance.

Example:

After the following command is issued, the “d” axis is converted to accommodate discrete digital output using the *bit* command.

```
axis:d,1
```

The following ASCII character appears in the mailbox:

```
1
```

The limit switches on the “d” axis may now be converted to support the *feed hold* feature by means of the *feedhold_input* command.

Example:

The following command may be used to convert the axis back to use as a motor controller:

```
axis:d,0
```

If the “d” axis is presently being used for the *feed hold* feature inputs, the axis will not be converted for use as a motor controller and the following error message will appear in the mailbox:

```
error,feedhold is enabled
```

Otherwise, the following ASCII character, representing the new mode for the axis, will appear in the mailbox.

```
0
```

If the “d” axis has been assigned for the *feed hold* inputs, the following sequence of commands is necessary to re-configure it as a motion controller:

```
feedhold_input:d,0
axis:d,0
```

AXIS?

Synopsis:

Read the “mode” of the designated axis.

Syntax:

```
axis?:<axis>
```

Returns:

An ASCII string depicting the mode of the axis is made available to be read from the mailbox. The meaning of the mode is as follows:

`0`	Axis is set up for motor control.
`1`	Axis is set up for discrete digital output.
2	Axis is set up for switch scanning
3	Axis is reserved for the *feed rate override* feature

Example:

Assume the “c” axis had been set up for use as discrete digital outputs by means of the *axis* command. The following command issued:

```
axis?:c
```

As a result, the following of ASCII character is available to be read from the mailbox:

```
1
```

BIT

Synopsis:

Exercise simplified discrete digital output control from the designated axis.

Syntax:

```
bit:<axis>,<bit number>,<logic level>
```

The “mode” of the designated axis must have been set to “1” using the *axis* command.

The `<bit number>` argument specifies the axis output signal lines as follows:

`<bit number>`	**Signal Line Name**
`0`	*Step*
`1`	*Direction*
`2`	*Reduced Current*
`3`	*All Windings Off*

The <logic level> argument specifies the output signal level as follows:

`<logic level>`	**Output Voltage**
`1`	TTL high
`0`	TTL low

Side Effects:

The signal line specified by `<bit number>` on the designated axis assumes the designated logic level. The mode of the designated axis must be set to `1` using the *axis* command in order for this command to function.

Returns:

An ASCII numeric character is available to be read from the mailbox which indicates the logic level that has been established.

If the axis has not been set to “mode=1” using the *axis* command, the *bit* command will not manipulate the digital output and the fol-

lowing message will appear in the mailbox:

```
error,mode
```

Example:

Assume that the mode of the “b” axis has been changed using the following command:

```
axis:a,1
```

Set the signal line named “reduced current” on the “a” axis to logic level 1 (TTL high):

```
bit:a,2,1
```

After execution of this command the *reduced current* output of the “a” axis assumes a corresponding TTL high voltage. The following ASCII character appears in the mailbox:

```
1
```

BIT?

Synopsis:

Return the logic level of the specified signal.

Syntax:

```
bit?:<axis>,<bit number>
```

The `<bit number>` argument specifies the axis output signal lines as follows:

`<bit number>`	Signal Line Name
`0`	*Step*
`1`	*Direction*
`2`	*Reduced Current*
`3`	*All Windings Off*

Returns:

This command returns the logic level of the signal specified by the `<bit number>` argument for the designated axis.

It should be noted that depending on the construction of the printer card being used, this command may read the actual voltage level present on the signal pin. Consequently, if a short circuit connection forces the potential of the signal pin to ground, the return value may be `0` even if the pin was set to logic `1` by another **Indexer LPT** command.

Example:

The mode of axis “c” has been set to `1` and bit number 2 (*Reduced Current*, pin 3 on the connector) has been set to logic `1` using the *bit* command. Approximately 5 volts (TTL high) is present on this signal. The following command is issued:

```
bit?:c,2
```

As a result, the following ASCII character is placed in the mailbox:

1

CAL

Synopsis:

Force timing calibration to a particular port.

Syntax:

```
cal:<axis>
```

Side Effects:

Indexer LPT calibration timing is accomplished to the port of the specified axis. Timing numbers are saved to the system **Registry**, and used whenever **Indexer LPT** is loaded. Calibration timing for this axis applies to all axes.

Returns:

A character reflecting the name of the axis that was used for calibration

appears in the mailbox.

CIRCLE

Synopsis:

Use the designated axes to traverse a circle around the specified center point. Optionally traverse up to two additional axes proportionately to the subtended angle (helical interpolation).

Syntax:

```
circle:<direction>,<axis>,<steps to cp>,
      <axis>,<steps to cp>
      [,<axis>,<steps>[,<axis>,<steps>]]
```

The direction in which the circle is drawn, clockwise or counter clockwise, is determined by the first command argument and the order in which the axes appear on the command line. “Clockwise” and “counterclockwise” are perceived in three dimensional space by “wrapping” the first axis into the second axis with the fingers of the right hand and viewing the circular motion from the direction of the thumb.

`<direction>`	**Meaning**
`cw`	arc is subtended clockwise
`ccw`	arc is subtended counterclockwise

`<steps to cp>` Signed magnitude of steps from the present position to the position of the center point.

Side Effects:

The circle is traversed with the specified axes at a vector velocity determined by the motion parameters set up in the *feed_lowspeed*, *feed_highspeed* and *feed_accel* registers. If a limit switch closure of a member axis is detected in an associated direction of movement, all motion is terminated and position tracking is lost.

The arc or helix is approximated by linear segments. The segment angle is determined by the *arcseg_degrees* register. Linear segments approximating the geometry are loaded into and executed from the queue buffer. Motion along the paths of the interpolation are governed by setup parameters in the *feed_lowspeed*, *feed_highspeed*, and *feed_accel* registers.

If a queue load sequence has been initiated with the *q_begin* command, the segments comprising this command are appended to the end of the queue buffer.

If a queue load sequence has not been initiated, then this command automatically begins the queue load sequence, loads the interpolation segments into the queue buffer, closes the sequence, and exe-

cutes the motion.

Returns:

If position tracking is in effect, an ASCII string representing the final position of each axis is placed in the mailbox. Position information concerning each axis is separated by a colon and appears in the order in which the axes are called out on the command line.

Example:

Assume the present position of the “a” axis is 1000 and the present position of the “b” axis is 2000. To draw a circle in the clockwise direction using a center point located 100 steps in the negative “a” direction and 500 steps in the “b” direction, the following command is issued:

```
circle:cw,a,-100,b,500
```

After the circle is traversed, the following message appears in the mailbox.

```
1000:2000
```

Assume the conditions of this example. Now **Indexer LPT** senses a closure of the high limit switch of the “a” axis when the “a” axis is moving in the positive direction. In this case the “circle” command is abruptly terminated and the following message appears in the mailbox:

```
limit,a,+:unknown position
```

Example

This example traverses a circle in the “a” and “b” axes, while simultaneously moving the “c” and “d” axes to their destination locations along a trajectory that is proportional to the angle being subtended:

```
circle:ccw,a,0,b,100,c,50,d,75
```

If all axes started from the home position, the following message, reporting the positions of the axes that were moved, appears in the mailbox:

```
0:0:50:75
```

COMMAND_MEM?

Synopsis:

Determine how much memory a command type would occupy if entered into the queue buffer.

Syntax:

```
command_mem?:<command>[command argument]
```

Returns:

An ASCII numeric string designating the amount of queue memory in bytes which a command would occupy when loaded into the queue buffer is placed in the mailbox.

If the command is not one which can be queued, the following message appears in the mailbox:

```
not supported
```

Side Effects:

When the *feed* command is tested using this command, the arguments to the *feed* command can optionally be included.

When circular interpolation commands are being tested, such as *circle*, *arc_to_point*, and *arc_to_angle*, their arguments must be included, since the arguments determine the amount of memory which these commands would occupy.

Example:

The following command is issued:

```
command_mem?:feed
```

As a result, an ASCII numeric string designating the amount of memory a *feed* command occupies in the queue buffer is placed in the mailbox.

The following command would yield the same result:

```
command_mem?:feed:a,1000,b,-500
```

Example:

Consider the following command:

```
command_mem?:arc_to_point:cw,a,0,b,1000,180
```

As a result, an ASCII numeric string designating the amount of memory this command would occupy in the queue buffer is placed in the mailbox. Since the amount of memory required for circular

interpolation is based on the number of segments used to approximate the geometry, under normal circumstances the following command will yield a value that is half that of the following command:

```
command_mem?:arc_to_point:cw,a,0,b,1000,90
```

Example:

The following command is issued:

```
command_mem?:set_lowspeed
```

As a result, the following message appears in the mailbox:

```
not supported
```

DWELL

Synopsis:

Delay for the amount of time designated by `<value>` in hundreths of a second.

Syntax:

```
dwell:<value>
```

Side Effects:

This command will cause the computer to delay (do nothing) for the specified amount of time. The maximum delay time possible using this command is four (4) seconds.

If the <value> argument is out of range (above 400), or if it cannot be interpreted, this command will delay for the maximum amount of 4 seconds.

DANGER

Avoid unexpected motion after extended delay. In applications such as machine tool control, allowing the machine to dwell for an extended period of time and resuming motion without warning could present a dangerous condition.

Do not use this command in critical timing applications. Extending delay time by repetitive execution of this command will result in a cumulative error in the total delay time.

Returns:

After successful execution the following message appears in the mailbox:

```
finished
```

If the <value> argument is over 400, or if it is not decipherable, the following message appears in the mailbox after the maximum period of delay:

```
error,range
```

FEATURES?

Synopsis:

Determine the features installed in the Hardware Assist Module.

Syntax:

```
features?
```

Returns:

An ASCII string designating the features installed into the **Hardware Assist Module** (HAM) appears in the mailbox.

Associated characters and features are as follows:

X Extended Queue Buffer.

E Limited to 4 simultaneous axes (to meet export regulations).

G License includes "G Code Controller" product.

H License includes "HPGL Controller" product.

(Check the **README.TXT** file on the distribution media for addtional features that may not be included in this manual).

Example:

The following string is written to Indexer LPT:

Example:

If the HAM supports Indexer LPT/XQ (extended queue), the "G Code Controller" and the "HPGL Controller", when the following command is issued:

```
features?
```

The mailbox will report the following string:

```
XGH
```

FEED

Synopsis:

Simultaneously move the selected axes by specified amounts such that the “best” fit line is traversed. Accelerate to the vector velocity determined by the *feed_highspeed* register. This command uses a velocity profile determined by the *feed_lowspeed*, *feed_highspeed*, and *feed_accel* registers.

Syntax:

```
feed:<axis>,<steps> ... [,<axis>,<steps>]
```

Side Effects:

Limit switch closure - If a limit switch closure is encountered from a limit switch associated with the direction of travel, motion is abruptly terminated. Limit switch closure on any axis in motion will arrest all motion. When a limit switch stop is encountered, position tracking is lost on all axes in motion.

Motion Parameters - The syntax and use of the *feed* command is similar to the *move* command. Unlike the *move* command, the maximum linear velocity along the path of traversal of the combined movement (vector velocity) is governed by the parameter set up in the *feed_highspeed* register. (Using the *move* command, the combined movement is governed by the motion parameters of the dominant axis. The *move* command is usually used for rapid traversal). The *feed* command is used where the vector velocity must be controlled.

Acceleration and starting velocity of the dominant axis is governed by the *feed_accel* and *feed_lowspeed* registers, respectively. The rate of the dominant axis after acceleration is such that the vector velocity specified in the *feed_highspeed* register is attained.

Returns:

An ASCII string reports the status of each axis in a similar manner to the *move* command. Information regarding participating axes appears in a message which may be read from the mailbox. Information relating to each axis is separated by a colon (:) and appears in the order in which the axes are called out on the command line.

Example:

Assume a custom engraver is being controlled by the “a” and “b” axes. Each step translates to .001 inches of movement along the associated axis. Using the *set_feed_lowspeed* command, the *feed_lowspeed* register is set to 200 steps per second. Using the

set_feed_accel command, the *feed_accel* register is set to 3500 steps per second squared, and using the *set_feed_highspeed* command the *feed_highspeed* register is set to 500 steps per second.

In this example, the contents of the *feed_highspeed* register establishes an actual feed rate of .5 inches per second.

The following command is issued:

```
feed:a,9000,b,12000
```

The “b” axis, which is dominant due to its greater number of steps, will start moving at 200 steps per second and accelerate to 400 steps per second at a rate of 3500 steps per second-per second. (**Indexer LPT** calculates the rate of 400 steps per second of the dominant axis in order to establish a cutting speed of .5 inches per second along the path of the cut).

Example:

A cutting tool is controlled in three axes. Assume each step represents .001 inch of travel. The value set in the *feed_highspeed* register is 1000 steps per second, which translates into an actual feed rate of 1 inch per second. The following command is issued:

```
feed:a,3000,b,4000,c,6000
```

After acceleration, the dominant “c” axis will maintain a rate of 768 steps per second in order to accomplish the feed rate of 1 inch per second along the path of the cut.

In this example, assume the positions of the “a”, “b”, and “c” axes were 50, 70, and 90 respectively. After the execution of the command, the following message appears in the mailbox:

```
3050:4070:6090
```

FEED_ACCEL?

Synopsis:

Read the contents of the *feed_accel* register.

Syntax:

```
feed_accel?
```

Returns:

An ASCII numeric string designating the contents of the *feed_accel* register in steps per second-per second is placed in the mailbox. This value is either the default value, or the value which was set by the *set_feed_accel* command.

Example:

Assume the value of the *feed_accel* register has been previously set to 495. To make this value available to be read from the mailbox the following command is issued:

```
feed_accel?
```

As a result, the following string of ASCII characters is placed in the mailbox:

```
495
```

FEED_HIGHSPEED?

Synopsis:

Read the contents of the *feed_highspeed* register.

Syntax:

```
feed_highspeed?
```

Returns:

An ASCII numeric string designating the value of the *feed_highspeed* register is placed in the mailbox. This value is either the default value, or the value which was set by a *set_feed_highspeed* command.

Example:

Assume the value of the *feed_highspeed* control register has been previously set to 750. To make this value available to be read from the mailbox, the following command is issued:

```
feed_highspeed?
```

As a result, the following string of ASCII characters appears in the mailbox.

```
750
```

FEED_LOWSPEED?

Synopsis:

Read the contents of the *feed_lowspeed* register.

Syntax:

```
feed_lowspeed?
```

Returns:

An ASCII numeric string designating the initial velocity in steps per second of the *feed_lowspeed* register is placed in the mailbox. This value is either the default value, or the value which was set by a *set_feed_lowspeed* command.

Example:

Assume the value of the *feed_lowspeed* control register has been previously set to 250. To make this value available to be read from the mailbox the following command is issued:

```
feed_lowspeed?
```

As a result, the following string of ASCII characters appears in the mailbox:

```
250
```

FEEDHOLD?

Synopsis:

Return the logical condition of the *feed hold* input.

Syntax:

```
feedhold?
```

Returns:

The mailbox contains an ASCII numeric character:

`1` if the *feed hold* input is activated

`0` if the *feed hold* input is not activated

The *feed hold* input is “activated” when there exists an open circuit on the high limit switch input (TTL high) of the axis designated to support the *feed hold* features. The *feed hold* input is “not activated” when this pin is connected to ground (TTL low).

If this command is issued and the *feed hold* feature has not been enabled, the following message appears in the mailbox:

```
error,disabled
```

Example:

Assume the “d” axis has been specified to support the *feed hold* feature by means of the following command sequence:

```
axis:d,1
feedhold_input:d,1
```

In this example also assume the *high limit switch* input of the “d” axis, which now serves as the *feed hold* input, is at TTL low potential due to a circuit path to ground through a normally closed switch. (In this situation the *feed hold* feature is not activated, and motion commands will not be suspended). The following command is issued:

```
feedhold?
```

As a result, the following ASCII character appears in the mailbox:

```
0
```

Assume that a motion command is NOT in progress and the switch in this example is open, causing the *feed hold* input to be activated. After the *feedhold?* command is issued, the following ASCII character appears in the mailbox:

```
1
```

It should be noted that **Indexer LPT is** not accessible when motion is in the process of being suspended using the *feed hold* feature. Consequently, the *feedhold?* command cannot be used to determine the status of the *feed hold* input when the *feed hold* feature is actively suspending another command.

FEEDHOLD_INPUT

Synopsis:

Assign the *feed hold* feature to be controlled by the limit switch inputs of the designated axis.

Syntax:

```
feedhold_input:<axis>,<status>
```

<status>	Meaning
0	Disable limit switches for use with *feed hold* features.
1	Enable limit switches for use with *feed hold* features.

Side Effects:

This command converts the *high* and *low limit switch* inputs on the designated axis to special use as *feedhold* and *abort* inputs respectively. The axis must either be internally assigned to "mode 3", or it must be placed in "mode 1" using the *axis* command as follows:

```
axis:<axis>,1
```

Only one axis can be designated to support the *feed hold* feature. The axis which supports the *feed hold* feature may be changed by subsequent use of this command.

Once an axis has been converted for use with the *feed hold* feature, the *high limit switch* input serves as the *feed hold* input. The *low limit switch* input serves as the *abort* input. TTL low voltage applied to the *feed hold* input is required for motion on any or all axes. An open circuit on the *feed hold* input, resulting in TTL high voltage, suspends motion. Motion resumes if the *feed hold* input is returned to TTL low. If motion is being suspended, TTL low applied to the *abort* input breaks out of (aborts) the **Indexer LPT** motion command which is being suspended.

Returns:

If the designated axis is in neither "mode 3" nor "mode 1", the *feedhold_input* command will have no effect. The following message appears in the mailbox:

```
error,mode
```

Otherwise, an ASCII character representing the *feed hold* status will appear in the mailbox.

If the *feed hold* feature is enabled and a motion command is suspended because of the *feed hold* input line, and if the motion com-

mand is terminated by means of the *abort* input, the following message appears in the mailbox:

```
abort
```

Example:

Using the following command, the “d” axis is converted from its default mode as a motor controller to the mode supporting discrete digital output:

```
axis:d,1
```

The following command is issued to specify the *limit switch* inputs of the “d” axis for special use to support the *feed hold* feature:

```
feedhold_input:d,1
```

The following ASCII character appears in the mailbox:

```
1
```

Subsequently, the following **Indexer LPT** command is issued:

```
move:a,35000,b,10000
```

While the “a” and “b” axes are in motion, **Indexer LPT** senses an open circuit on the *feed hold* input line. The motion on the “a” and “b” axes is decelerated to a controlled stop. Motion is arrested until the *feed hold* input line is brought to a ground potential (de-activated). Once the *feed hold* input is de-activated, the **Indexer LPT** completes the *move* command.

If the *abort* input line is brought to ground potential before the *feed hold* input line is de-activated, the *move* command will not be completed and the following ASCII string will appear in the mailbox:

```
abort
```

Example:

Assume the “d” axis is in “mode 0” or “mode 2”, and the following command is issued:

```
feedhold_input:d,1
```

The *feed hold* feature will NOT be enabled for this axis, and the following string of ASCII characters appears in the mailbox:

```
error,mode
```

FEEDHOLD_INPUT?

Synopsis:

Determine which axis, if any, is set up to support the *feed hold* feature.

Syntax:

```
feedhold_input?
```

Returns:

An ASCII character representing the axis which is designated to support the *feed hold* feature appears in the mailbox. If no axis has been designated, the following message appears in the mailbox:

```
none
```

Example:

The “d” axis is designated to support the *feed hold* feature by means of the following command sequence:

```
axis:d,1
feedhold_input:d,1
```

The following command is then issued:

```
feedhold_input?
```

As a result the following message appears in the mailbox:

```
d
```

FRO?

Read the calculated percentage of feed rate override that is applied to motion, consisting of the combined effect of the FRO input voltage and the value of the *fro_offset* register.

Syntax:

```
fro?
```

Example:

Assume the *fro_highvolt* register is set to 256, the *fro_high* register is set to 130 and the *fro_offset* register contains the default value of 0. If 5.0 volts is applied to pin 11 of the HAM and the following command is issued:

```
fro?
```

these ASCII characters will appear in the mailbox:

```
130
```

In this example, if the *fro_offset* register contained a value of -10, the *fro*? command would return 120. However, if the *fro_offset* register contained a value of 10, the *fro*? command would return 130, since the amount of feed rate override cannot exceed the value set up in the *fro_high* register.

FRO_DELAY?

Synopsis:

Read the contents of the *fro_delay* register.

Syntax:

```
fro_delay?
```

Returns:

An ASCII numeric string designating the setting of the *fro_delay* registerappears in the mailbox. The value of the *fro_delay* register can be changed by means of the *set_fro_delay* command.

FRO_HIGH?

Synopsis:

Read the contents of the *fro_high* register.

Syntax;

```
fro_high?
```

Returns:

An ASCII numeric string designating the setting of the *fro_high* register appears in the mailbox. The value of the *fro_high* register can be changed by means of the *set_fro_high* command.

FRO_HIGHVOLT?

Synopsis:

Read the contents of the *fro_highvolt* register.

Syntax:

```
fro_highvolt?
```

Returns:

An ASCII numeric string designating the setting of the *fro_highvolt* register appears in the mailbox. The value of the *fro_highvolt* register can be changed by means of the *set_fro_highvolt* command.

FRO_LOW?

Synopsis:

Read the contents of the *fro_low* register.

Syntax:

```
fro_low?
```

Returns:

An ASCII numeric string designating the setting of the *fro_low* register appears in the mailbox. The value of the *fro_low* register can be changed by means of the *set_fro_low* command.

FRO_LOWVOLT?

Synopsis:

Read the contents of the *fro_lowvolt* register.

Syntax:

```
fro_lowvolt?
```

Returns:

An ASCII numeric string designating the setting of the *fro_lowvolt* register appears in the mailbox. The value of the *fro_lowvolt* register can be changed by means of the *set_fro_lowvolt* command.

FRO_RES?

Synopsis:

Read the contents of the *fro_res* register.

Syntax:

```
fro_res?
```

Returns:

An ASCII numeric string designating the setting of the *fro_res* register appears in the mailbox. The value of the *fro_res* register can be changed by means of the *set_fro_res* command.

FRO_VOLT?

Synopsis:

Read the voltage from FRO input pin 11 of the **Hardware Assist Module**.

Syntax:

```
fro_volt?
```

Side Effects:

Voltage is read from pin 11 with reference to ground (pins 18-25) in increments of 5/256 = 0.01953 Volts over a range of 0 to 5 volts, corresponding to 0 to 256 increments.

Returns:

ASCII Numerical values over a range of 0 to 256.

Example:

If 1.0 Volts is presented at pin 11 when the following commandis issued:

```
fro_volt?
```

the following ASCII characters will appear in the mailbox:

```
51
```

corresponding to the nearest increment to the value 1/(5/256)

HAM_TYPE?

Synopsis:

Report the numerical type of the **Hardware Assist Module**.

Syntax:

```
ham_type?
```

Returns:

An ASCII numerical value designatingthe type of **Hardware Assist Module** that is installed in the system appears in the mail-box

HIGHSPEED?

Synopsis:

Read the contents of the *highspeed* register.

Syntax:

```
highspeed?:<axis>
```

Returns:

An ASCII numeric string designating the *highspeed* setting of selected axis in steps per second appears in the mailbox. This value is either the default value, or the value which was set by a *set_highspeed* command.

Example:

Assume the value of the highspeed control register for the “c” axis has been previously set to 755. To make this value appear in the mailbox the following command is issued:

```
highspeed?:c
```

As a result the following string of ASCII characters appears in the mailbox:

```
750
```

HOME

Synopsis:

Move the selected axis to the *home* reference position established by the *set_home* command.

Syntax:

```
home:<axis>
```

Side Effects:

The axis moves to the home position. Velocity and acceleration are governed by the associated *lowspeed*, *highspeed*, and *acceleration* registers. This command will only function if the *set_home* command has been previously issued for the selected axis.

Returns:

After successful completion, an ASCII `0` (zero character) appears in the mailbox. If the axis reference home position has not been established with the *set_home* command, the following error message appears in the mailbox:

```
error,position
```

JOG

Synopsis:

Move the selected axis the desired number of steps using a constant velocity profile.

Syntax:

```
jog:<axis>,<steps>
```

Side Effects:

If a limit switch closure is encountered on the limit switch associated with the direction of travel, motion is abruptly terminated. Unlike other motion commands, position tracking information is preserved.

Returns:

If the *position* register for the axis being moved has been previously initialized with the *set_home* command, an ASCII numeric string representing the value of the *position* register is returned.

If a limit switch closure is encountered from the limit switch associated with the direction of travel, motion is abruptly terminated. In this case the return string of the *jog* command represents the position where the motor has stopped.

DANGER

The limit switch detection features of this program are designed only to provide limit detection within the normal operating region of the device being controlled and NOT to provide over-travel protection in cases where equipment damage or personal injury may result. In cases where equipment damage or personal injury is possible due to over-travel, other means of limit protection is imperative.

Example:

The “a” axis motor is at a position of 200 steps relative to *home*. To move the “a” axis 500 steps in the negative direction issue the command:

```
jog:a,-500
```

The motor will move 500 steps in the negative direction and the following ASCII string appears in the mailbox:

```
-300
```

Example:

The position of the “b” axis is 500 when the motor is instructed to move to position 800 by the following command:

```
jog:b,300
```

After the motor moves 80 steps, motion is arrested by the closure of the high limit switch. The following ASCII string appears in the mailbox:

```
limit,b,+
```

The position of the axis may be determined now by issuing the following command:

```
position?:b
```

The following string now appears in the mailbox:

```
580
```

As long as this limit switch remains closed, the motor cannot be moved in the positive direction. In this example, the motor is “backed away” from the high limit switch by moving it in the negative direction using the following command:

```
move:b,-200
```

The motor moves 200 steps in the negative direction and the following string appears in the mailbox:

```
380
```

JOGSPEED?

Synopsis:

Read the contents of the *jogspeed* register.

Syntax:

```
jogspeed?:<axis>
```

Returns:

An ASCII numeric string designating the jog speed setting of selected axis in steps per second appears in the mailbox. This value is either the default value, or the value which was set by a *set_jogspeed* command.

Example:

Assume the value of the jogspeed control register for the “c” axis has been previously set to 250. To make this value appear in the mailbox the following command is issued:

```
jogspeed?:c
```

As a result the following string of ASCII characters appears in the mailbox:

```
250
```

JOYSTICK_INPUT

Synopsis:

Define the operation of the joystick switches.

Syntax:

```
joystick_input:<joystick axis>,
      <selection>,<s/d axis>,
      <s/d increment>,<r/a axis>,
      <r/a increment>
```

or

```
joystick_input:clear
```

`<joystick axis>` - Axis which is used to scan the joystick control switches.

`<selection>` `0` or `1`

`0` defines joystick behavior for the case when a scan between the joystick axis *all windings off* output and the joystick axis *low limit switch* input determines an open circuit condition.

`1` defines joystick behavior for the case when a scan between the joystick axis *all windings off* output and the joystick axis *low limit switch* input determines an closed circuit condition.

`<s/d axis>` Axis which is controlled by scanning the joystick axis *step* and *direction* outputs into the joystick axis *high limit switch* input.

`<s/d increment>` When a scan between the joystick axis *reduced current* output and the joystick axis *low limit switch* input indicates a closed circuit connection, this parameter determines the number of steps which can be issued over the s/d axis between the .5 second latency period.

<r/a axis> Axis which is controlled by scanning the joystick axis *reduced current* and *all windings off* outputs into the joystick axis *high limit switch* input.

<r/a increment> When a scan between the joystick axis *reduced current* output and the joystick axis *low limit switch* input indicates a closed circuit connection, this parameter determines the number of steps which can be issued over the r/a axis between the .5 second latency period.

Note: The "mode" of the joystick axis must have been set to `2` using the *axis* command.

Side Effects:

This command defines the operation of the *joystick_go* command by designating the joystick control axis and the associated axes to be controlled. Otherwise, when used with the `clear` argument, this command clears set-up parameters stored from previous *joystick_input* commands.

Returns:

Upon successful completion, a string of characters representing the values which were stored appears in the mailbox:

```
<joystick axis>,<selection>,<s/d axis>,
      <s/d increment>,<r/a axis>,
      <r/a increment>
```

Otherwise, the following error messages may appear in the mailbox:

`error,mode`

The joystick axis is not in mode `2`, or one of the other axes is not in mode `0` (The default mode for all axes is `0`. The mode of any axis can be changed using the *axis* command).

`error,assignment`

Only one axis on the system can be specified for use with the joystick input. This error message is generated if an axis has been assigned for use as the joystick input by a previous *joystick_input* command, and the present command attempts to specify a different one.

`joystick is cleared`

This message is generated as a response to the following command:

`joystick_input:clear`

Example:

Consider a motion system where the "e" axis is to be used for joystick switch scanning. The following conditions are desired:

Axes "a" and "b" are to be controlled under selection 0. (Selection `0` is physically set (by an open switch circuit), and scanning between the joystick axis *all windings off* output and the joystick axis *low limit switch* input determines an open circuit connection.

Axes "c" and "d" are to be controlled under selection `1`. (Selection 1 is physically set (by a closed switch circuit), and scanning

between the joystick axis *all windings off* output and the joystick axis *low limit switch* input determines a closed circuit connection).

When scanning between the joystick axis *reduced current* output and the joystick axis *low limit switch* input detects a closed circuit condition (placing the control feature in "nudge" mode), the desired number of burst steps set up for axis “a” is 10, ”b” is 11, “c” is 12, and “d” is 13.

The complete software setup is as follows:

```
# Make sure axes to be controlled are in
#default mode
axis:a,0
axis b,0

# Joystick control axis in scanning mode
axis:e,2

# Setup for open circuit selection switch
joystick_input:e,0,a,10,b,11

# Setup for closed circuit selection switch
joystick_input:e,1,c,12,d,13
```

To activate the joystick the following command is issued:

```
joystick_go
```

JOYSTICK_INPUT?

Synopsis:

Read the setup information established by means of the *joystick_input* command.

Syntax:

```
joystick_input:<selection>
```

`<selection>` `0` or `1`

`0` read joystick parameters defined for the case when a scan between the joystick axis *all windings off* output and the joystick axis *low limit switch input* determines an open circuit condition.

`1` read joystick parameters defined for the case when a scan between the joystick axis *all windings off* output and the joystick axis *low limit switch input* determines an closed circuit condition.

Returns:

```
<joystick axis>,<selection>,<s/d axis>,
            <s/d increment>,<r/a axis>,
            <r/a increment>
```

`<s/d axis>` Axis which is controlled by scanning the joystick axis *step* and *direction* outputs into the joystick axis *high limit switch input*.

`<s/d increment>` When a scan between the joystick axis *reduced current* output and the joystick axis *low limit switch input* indicates a closed circuit connection, this parameter determines the number of steps which can be issued over the s/d axis between the .5 second latency period.

`<r/a axis>` Axis which is controlled by scanning the joystick axis *reduced current* and *all windings off* outputs into the joystick axis *high limit switch input*.

`<r/a increment>` When a scan between the joystick axis *reduced current* output and the joystick axis *low limit switch input* indicates a closed circuit connection, this parameter determines the number of steps which can be issued over the r/a axis between the .5 second latency period.

Example:

Assume following *joystick_input* commands have been issued:

```
joystick_input:e,0,a,10,b,11
joystick_input:e,1,c,20,none,none
```

When followed by this command:

```
joystick_input?:0
```

Indexer LPT places the following message in the mailbox:

```
e,0,a,10,b,11
```

When this command is issued:

```
joystick_input?:1
```

Indexer LPT places the following message in the mailbox:

```
e,1,c,20,none,none
```

JOYSTICK_GO

Synopsis:

Continuously scan the axis designated for joystick input.

Syntax:

```
joystick_go
```

Side Effects:

If a joystick axis has not been designated by means of the *joystick_input* command, the *joystick_go* command exits immediately, leaving an error message in the mailbox. Otherwise, joystick control is performed as per parameters set up via *joystick_input* command(s).

This command causes the four outputs of the designated joystick axis to be continuously scanned into the *high limit switch input* and the *low limit switch input* of the designated joystick axis. Axes set up for motion control under the *joystick_inpu*t command are caused to move according to the status of the switch scan and the setup parameters.

Scanning is discontinued when a scan between the *direction* output into the *low limit switch input* indicates a closed circuit condition.

This command occupies the computer until scanning is terminated.

Returns:

In normal operation, once terminated this command fills the mailbox with the name and position of each axis which has been set up for joystick control via the *joystick_input* command. The format is as follows:

```
<axis>,<position>...[,<axis>,<position>]
```

From one to four axes may be included in the message, depending on how many axes have been set up to be controlled by the joystick. If no axis has been set up for joystick control via the *joystick_input* command, the following message appears in the mailbox:

```
error,joystick is disabled
```

Example:

Consider the following sequence of commands:

```
axis:e,2
```

```
joystick_input:e,0,a,10,b,10
joystick_input:e,1,none,none,c,20
set_home:a
set_home:b
set_home:c
joystick_go
```

Assume that following the *joystick_go* command, the joystick is used to move the "a" axis 1250 steps in the positive direction, the "b" axis 447 steps in the negative direction, and the "c" axis 223 steps in the positive direction.

Scanning continues until contact is made between the *direction* output of the "e" axis and the *low limit switch input* of the "e" axis. by means of a pushbutton switch wired for this purpose.

The following message subsequently appears in the mailbox:

```
a,1250,b,-447,c,223
```

-LIMIT?

Synopsis:

Determine if the low (-) limit switch is closed.

Syntax:

```
-limit?:<axis>
```

Side Effects:

When TTL low, the *low limit switch* input arrests motion in the negative (-) direction. This command simply queries the condition of the *low limit switch* input.

Returns:

The mailbox contains an ASCII numeric character:

`0` if the low limit switch is open, allowing the voltage at the associated input pin to be internally pulled TTL high.

`1` if the low limit switch is closed, grounding the associated input pin.

It should be noted that unlike some of the other query commands, this command returns the logical condition of the switch and NOT the TTL logic level of the input pin.

DANGER

The limit switch detection features of this program are designed only to provide limit detection within the normal operating region of the device being controlled and NOT to provide over-travel protection in cases where equipment damage or personal injury may result. In cases where equipment damage or personal injury is possible due to over-travel, other means of limit protection is imperative.

Example:

Using the *move* command, the motion of the “a” axis is interrupted in the negative direction by the closure of the low limit switch. The physical member being controlled by the “a” axis is still in contact with the limit switch when the following command is executed:

```
-limit?:a
```

The following character appears in the mailbox:

```
1
```

The limit switch closure is then removed and the following command is once again issued:

```
-limit?:a
```

The following character appears in the mailbox:

```
0
```

+LIMIT?

Synopsis:

Determine if the high (+) limit switch is closed.

Syntax:

```
+limit?:<axis>
```

Side Effects:

When TTL low, the *high limit switch* input arrests motion in the positive (+) direction . This command simply queries the condition of the high limit switch.

Returns:

The mailbox contains an ASCII numeric character:

`0` if the high limit switch is open, allowing the voltage at the associated input pin to be internally pulled TTL high.

`1` if the high limit switch is closed, grounding the associated input pin.

It should be noted that unlike some of the other query commands, this command returns the logical condition of the switch and NOT the TTL logic level of the input pin.

DANGER

The limit switch detection features of this program are designed only to provide limit detection within the normal operating region of the device being controlled and NOT to provide over-travel protection in cases where equipment damage or personal injury may result. In cases where equipment damage or personal injury is possible due to over-travel, other means of limit protection is imperative.

Example:

Using the *move* command, the motion of the “a” axis is interrupted in the positive direction by the closure of the high limit switch. The physical member being controlled by the “a” axis is still in contact with the limit switch when the following command is executed:

```
+limit?:a
```

The following character appears in the mailbox:

```
1
```

The limit switch closure is then removed and the following command is once again issued:

```
+limit?:a
```

The following character appears in the mailbox:

```
0
```

LOWSPEED?

Synopsis:

Read the contents of the *lowspeed* register.

Syntax:

```
lowspeed?:<axis>
```

Returns:

An ASCII numeric string designating the *lowspeed* setting of selected axis in steps per second appears in the mailbox. This value is either the default value, or the value which was set by a *set_lowspeed* command.

Example:

Assume the value of the *lowspeed* register for the “c” axis has been previously set to 250. To make this value appear in the mailbox the following command is issued:

```
lowspeed?:c
```

As a result the following string of ASCII characters appears in the mailbox:

```
250
```

MAX_Q_MEM?

Synopsis:

Determine the amount of memory which can be made available for use in the queue buffer.

Syntax:

```
max_q_mem?
```

Returns:

An ASCII numeric string designating the amount of memory which can be made available for use in the queue buffer is placed in the mailbox.

The amount of queue buffer memory is limited by the amount that Windows makes available for use, and is determined by the physical memory capacity of your computer.

Side Effects:

This command only reports the amount of memory that CAN be made available to **Indexer LPT**. To make a portion or all of this memory available to **Indexer LPT**, use the *set_q_mem* command.

Example

The following string is written to **Indexer LPT**:

```
max_q_mem?
```

As a result an ASCII string designating the amount of memory available for use by **Indexer LPT** appears in the mailbox, such as

```
4235642
```

MAX_SPEED?

Synopsis:

Determine the maximum velocity in steps per second which can be accomplished by the host computer system. This represents the maximum value which may be entered in the *highspeed* or the *feed_highspeed* register for any axis.

Syntax:

```
max_speed?
```

Returns:

An ASCII numeric string appears in the mailbox representing the value of the absolute maximum speed which can be obtained from **Indexer LPT** on the computer which is being used. The return value for this command will vary depending upon the speed of the host computer. The faster the computer, the greater the maximum possible speed will be.

Since calibration of the timing parameters occurs each time the program is loaded, and since calibration is somewhat asynchronous in nature, the value may vary somewhat each time the program is run. The faster the computer, the less pronounced the variations tend to be.

MOVE

Synopsis:

Simultaneously move the selected axes the specified amount using the “best fit” straight line strategy. This command uses a constant acceleration profile. One to six axes may be moved simultaneously.

Syntax:

```
move:<axis>,<steps> ... [,<axis>,<steps>]
```

Side Effects:

Limit switch closure - If a limit switch closure is encountered from a limit switch associated with the direction of travel, motion is abruptly terminated and position tracking information is lost. Limit switch closure on any axis in motion will arrest all motion. When a limit switch stop is encountered, position tracking is lost on all axes in motion.

Motion parameters - Parameters associated with the dominant axis (the axis which is being moved the greatest amount of steps) govern the coordinated movement. The other axes are controlled according to the “best fit” straight line strategy. The dominant axis moves according to parameters stored in its associated *lowspeed*, *highspeed*, and *accel* registers. The *move* command is generally used to implement the most rapid traversal.

Returns:

If position tracking has been initialized, an ASCII numeric string representing the value of the position register appears in the mailbox. Otherwise, the following message appears:

```
unknown position
```

For multiple axis moves, information on each axis is separated by a colon (:) and appears in the order in which the axes are called on the command line.

If a limit switch closure is encountered from the limit switch associated with the direction of travel, motion is abruptly terminated and position tracking is lost. The position report string associated with the axis that caused the limit switch stop appears in the following format:

```
limit,<axis>,<direction>
```

DANGER

The limit switch detection features of this program are designed only to provide limit detection within the normal operating region of the device being controlled and NOT to provide over-travel protection in cases where equipment damage or personal injury may result. In cases where equipment damage or personal injury is possible due to over-travel, other means of limit protection is imperative.

Example:

The “a” axis motor is at a position of 200 steps relative to the home position, which has been previously set by the command:

```
set_home:a
```

To move the “a” axis 500 steps in the negative direction issue the command:

```
move:a,-500
```

The motor will move 500 steps in the negative direction and the following ASCII string appears in the mailbox:

```
-300
```

Example:

The “b” axis motor *home* position has been previously initialized using the command:

```
set_home:b
```

However, after the last command to move the motor in the positive direction, the motion of the motor was abruptly stopped by the closure of the high limit switch. The following message appears in the mailbox:

```
limit,b,+
```

As long as this limit switch remains closed, the motor cannot be moved in the positive direction. In this example, the motor is backed away from the high limit switch by moving it in the negative direction using the following command:

```
move:b,-200
```

The motor moves 200 steps in the negative direction and the following string appears in the mailbox:

```
unknown position
```

Example:

Assume a motion system is comprised of three axes of motion controlling linear motion of a stylus in the three dimensions. The three **Indexer LPT** axes which are used are “a”, “b”, and “c”. The following command is issued:

```
move:a,1200,b,5000,c,-350
```

The “b” axis is the dominant axis, since it is required to move the greatest amount. The stylus moves from its present position along the “best fit” straight line path to a position 1200 steps in the “a” direction, 5000 steps in the “b” direction and -350 steps in the negative “c” direction. The “b” axis will move and accelerate according to motion parameters set up for the “b” axis. The other axes will move as necessary to accomplish the linear interpolation. Assume, for example, that all three axes were located at *home* position before the move. After the motion is completed, the following message appears in the mailbox:

```
1200:5000:-350
```

If the motion in this example was terminated by the closure of the high limit switch of the “b” axis, the following message would appear in the mailbox:

```
unknown position:limit,b,+:unknown position
```

Example:

Consider the following command:

```
move:a,12,b,500,c,-350,d,100,e,2000,f,-250
```

Six axes are moved simultaneously. Motion parameters set up for the dominant “e” axis govern the dynamic characteristics for the motion.

If all axes in this example had started movement from respective *home* positions, the following message would appear in the mailbox:

```
12:500:-350:100:2000:-250
```

POSITION?

Synopsis:

Read the contents of the *position* register.

Syntax:

```
position?:<axis>
```

Side Effects:

None

Returns:

An ASCII numeric string designating the number of steps from the *home* reference previously established with the *set_home* command is placed in the mailbox. This number can be either positive or negative.

If the *home* reference position has not been initialized, or if the reference position has been lost by a limit switch closure, the following message appears in the mailbox:

```
unknown position
```

Example:

The following sequence of commands is executed:

```
set_home:c
move:c,5000
move:c,-200
position?:c
```

After the completion of this sequence the following ASCII numeric string appears in the mailbox:

```
4800
```

Q_BEGIN

Synopsis:

Append subsequent commands into the queue buffer.

Syntax:

```
q_begin
```

Side Effects:

This command initiates a queuing procedure. All appropriate motion commands following this command will be entered into the queue buffer. The queuing procedure is terminated by means of the *q_end* command.

Following this command and before the *q_end* command, only commands appropriate for queue processing will be entered into the queue buffer. Other commands which may be used during the queuing procedure are *q_empty?*, *q_mem?*, *command_mem?*, and *q_reset*. The *q_reset* command terminates the queuing procedure and clears the queue buffer.

The number of commands which can be accepted into the queue buffer depends upon the size of the commands which are queued and the size of the queue buffer.

Returns:

An ASCII numeric string designating the amount of memory which is available in the queue buffer is placed in the mailbox. The ASCII string comprises numeric characters followed by a space character and the word “remains”.

As each command is entered into the queue during the queuing procedure, the amount of memory which remains available in the queue buffer is reported in the mailbox.

If an attempt is made to queue a command when insufficient memory is available in the queue buffer to receive it, the following message appears in the mailbox:

```
error,queue full
```

If an attempt is made to queue a command which is unsuitable for queue processing, the following message appears in the mailbox:

```
error,queue
```

Example:

The following sequence queues three *feed* commands:

```
q_begin
feed:a,1000,b,500
feed:a,3000,b,700
feed:a,400,b,300,c,200
q_end
```

Assume in this example that 10000 bytes of memory is available for use by the queue. After the *q_begin* command is issued, the following message appears in the mailbox:

```
10000 remains
```

After each command is queued, a similar message is placed in the mailbox indicating the amount of remaining memory at that time.

After this queuing sequence, three *feed* commands occupy the queue buffer. Assume that the "d" axis must be moved before execution of this sequence. The following command is issued:

```
feed:d,400
```

The "d" axis is immediately moved 400 steps.

Now assume that another command is to be added to the queue. The following commands are issued:

```
q_begin
feed:a,1000,b,-400,c,-1000
q_end
```

Finally, when the *q_go* command is issued, the following four commands are executed from the queue:

```
feed:a,1000,b,500
feed:a,3000,b,700
feed:a,400,b,300,c,200
feed:a,1000,b,-400,c,-1000
```

Q_EMPTY?

Synopsis:

Determine if there are any commands in the queue buffer.

Syntax:

```
q_empty?
```

Returns:

The mailbox contains an ASCII numeric character

`1` the queue buffer is empty

`0` the queue buffer is not empty

Example:

The following commands are issued:

```
q_reset
q_empty?
```

After this sequence the following ASCII character appears in the mailbox:

`1`

Q_END

Synopsis:

End entry of commands into the queue buffer.

Syntax:

```
q_end
```

Side Effects:

This command ends the queuing procedure which was initiated by the *q_begin* command, and must be issued before the *q_go* command. If the queue is empty, this command has no effect.

Following the *q_end* command and before the *q_go* command, commands which set parameters governing queuable commands cannot be issued. For example, *set_feed_highspeed*, *set_feed_lowspeed*, *set_feed_accel*, and *set_vshift* cannot be used unless the queue buffer is empty. Other commands will be executed immediately.

Following termination of the queuing procedure by means of the *q_end* command, the queuing procedure may be resumed by issuing a *q_begin* command. Queuable commands will be appended to the end of the queue. Once resumed, the queuing procedure must be terminated by means of another *q_end* command before the queue can be executed by means of *q_go*.

Returns:

An ASCII numeric string designating the amount of unused memory which is available in the queue buffer is placed in the mailbox. The ASCII string comprises numeric characters followed by a space character and the word “remains”.

Example:

The following sequence queues three feed commands:

```
q_begin
feed:a,1000,b,500
feed:a,3000,b,700
feed:a,400,b,300
q_end
```

The *q_end* command closes the queue entry procedure.

Example

In a cutting tool application where the depth of the cutting tool is controlled on the “d” axis, assume that it is desirable to load the queue with the cutting tool retracted, lower the cutting tool, execute the queue, then raise the cutting tool in preparation for the next queue of commands. The procedure begins with the cutting tool retracted so that the time spent loading the queue occurs while the tool is not in contact with the work. (The “d” axis must be moved before execution of the queued sequence). If the tool is extended via negative steps on the “d” axis, the following sequence is used:

```
q_begin
feed:a,1000,b,500
feed:a,3000,b,700
feed:a,400,b,300
q_end
feed:d,-400
q_go
feed:d,400
```

Example:

Assume the following command is issued following *q_end* but before *q_go*:

```
set_feed_highspeed:4000
```

The following message appears in the mailbox:

```
error,queue
```

Example:

The following commands are issued:

```
q_begin
feed:a,1000,b,-400,c,-1000,d,100
feed:a,200,b,1000
q_end
move:e,5000
q_begin
feed:a,700,b,1800
q_end
q_go
```

In this sequence, the “e” axis is moved first. After the “e” axis is finished moving, the last *feed* command is appended to the end of the queue. Finally, the *q_go* command causes the following com-

mands to be executed from the queue:

```
feed:a,1000,b,-400,c,-1000,d,100
feed:a,200,b,1000
feed:a,700,b,1800
```

Q_GO

Synopsis:

Execute the remaining commands in the queue buffer.

Syntax:

```
q_go
```

Side Effects:

The commands in the queue buffer are executed using this command. The *q_go* command begins executing the commands in the queue, starting with the first command in the buffer.

If queue execution has been suspended by means of an aborted *feed hold* procedure, *q_go* will complete the execution of the remaining commands in the queue buffer.

Returns:

After executing the last command in the queue buffer, the following message appears in the mailbox:

```
finished
```

This message also appears in the mailbox if *q_go* is executed when the queue buffer is empty.

Example:

Assume in this example that all associated axes begin from the home position.

The following sequence queues and executes three *feed* commands:

```
q_begin
feed:a,1000,b,500
feed:a,1000,b,700
feed:a,1000,b,300,c,200
q_end
q_go
```

The *q_end* command closes the queue entry procedure and prepares the queue for execution. The *q_go* command begins execution of the queue, starting with the first *feed* command. If the queue is executed to completion, the following message appears in the mailbox:

```
finished
```

Note that the position of the “a” axis upon completion would be

3000.

Now assume in this example that the *feed hold* feature is enabled, and the *feed hold* input is activated to bring motion to rest about half way into the second command. At this point, all processes are suspended. If the *feed hold* input is released, the queue will continue execution to completion, following the same path as in the previous example. Let us assume, however, that the *abort* input is activated. The following message appears in the mailbox:

```
abort
```

Now the *feed hold* input is released and the following command is issued:

```
move:a,750
```

The “a” axis immediately moves 750 steps. Now this command is issued:

```
q_go
```

The queue will continue execution as if the *feed hold* input had been released from its first suspended state, except that the “a” axis will have an offset of 750 steps. Upon completion of the queue execution, the position of the “a” axis will be 3750.

This example demonstrates the following two concepts:

1) If queue execution is suspended by means of a *feed hold*, the *q_go* command resumes processing from the point in the contour where motion stopped.

2) If motion commands are used in the interim, and if the positional integrity of the contour is to be preserved, then associated axes must be returned to the position from which they were moved.

Example

In a practical application, assume that a cutting tool is being operated under control of an application program. Seeing that the cutting tool is coming dangerously close to a clamp, the operator activates the *feed hold* input. Motion comes to a controlled stop. After determining a remedy is in order, the operator activates the *abort* input. Sensing the *abort* input via the message in the mailbox, the application program reads the positions of all associated axes with the *position?* command, then activates the joystick control by means of *joystick_go*. Using the joystick, the operator moves the cutting tool out of the way and repositions the clamp. Now the operator uses the joystick to move the cutting tool close to the position where the *feed hold* input interrupted the contour. When the operator activates the switch which discontinues joystick action, the application program reads the position of the axes again and constructs a *move* command which brings the cutting tool to the

exact position where the contour was interrupted. The application program then issues a *q_go* command to proceed cutting the contour. Positional integrity throughout the path of the contour is thereby maintained.

(It should be noted that the means of releasing *feed hold* and releasing joystick control depends on the type of machine being designed. In the interest of operator safety, some machines may require self latching relays, multiple “start” push-buttons, or other hardware features).

Q_MEM?

Synopsis:

Determine how much memory is left in the queue buffer.

Syntax:

```
q_mem?
```

Returns:

An ASCII numeric string designating the amount of unused memory which is available in the queue buffer is placed in the mailbox. The ASCII string comprises numeric characters followed by a space character and the word “remains”.

Example:

Assume the size of the queue buffer is 10000 bytes, and 6000 bytes are occupied. The following command is issued:

```
q_mem?
```

As a result, the following message appears in the mailbox:

```
4000 remains
```

Q_RESET

Synopsis:

Clear the queue buffer. Discontinue queue processing.

Syntax:

```
q_reset
```

Side Effects:

This command makes all the memory in the queue buffer available.

If this command is executed from within a queuing procedure initiated by *q_begin*, the queuing procedure is terminated.

Returns:

An ASCII numeric string designating the amount of memory which is available in the queue buffer is placed in the mailbox. The ASCII string comprises numeric characters followed by a space character and the word “remains”.

Example:

Assume the size of the queue buffer is 10000 bytes, and 6000 bytes are occupied. The following command is issued:

```
q_mem?
```

As a result, the following message is placed in the mailbox:

```
4000 remains
```

Now the following command is issued:

```
q_reset
```

As a result, the queue buffer is cleared, and the following message is placed in the mailbox:

```
10000 remains
```

Example:

The following sequence of commands is issued:

```
q_begin
feed:a,1000,b,500
feed:a,1000,b,700
q_reset
```

When *q_reset* is issued, the queue is emptied and queuing is terminated. If for example, a *feed* command is issued following *q_reset*, the *feed* command would be executed immediately and not entered into the queue buffer. To resume queuing, *q_begin* must be issued.

Example:

The following sequence queues and executes four *feed* commands:

```
q_begin
feed:a,1000,b,1000
feed:a,1000,b,400
feed:a,1000,b,-1400
feed:a,-3000
q_end
q_go
```

The *q_end* command closes the queue entry procedure and prepares the queue for execution. The *q_go* command begins execution of the queue starting with the first *feed* command.

Assume in this example that the *feed hold* feature is enabled and that the *feed hold* input is activated, bringing motion to rest before the queue is completely executed. At this time the *abort* input is activated. The process will immediately return with the following message in the mailbox:

```
abort
```

The queue buffer is not empty. If queuing is initiated by means of *q_begin*, ensuing commands would be added to the end of the existing queue. A *q_end*, *q_go* sequence would therefore begin execution from the point at which the *feed hold* suspended operation. If a fresh queue is desired, *q_reset* must be issued before *q_begin*.

Example:

Assume from the previous example the operation of a cutting tool where motion begins at location 0 for the “a” and “b” axes. Before execution of the queue is complete, *feed hold* is activated and terminated by means of the *abort* input. In this case a cutting tool must be moved back to its original position and the operation started from scratch. Following the *abort* input, the following sequence may be used:

```
home:a
home:b
q_reset
```

```
q_begin
feed:a,1000,b,1000
feed:a,1000,b,400
feed:a,1000,b,-1400
feed:a,-3000
q_end
q_go
```

Q_WHERE?

Synopsis:

Report the entity in the queue buffer from which a *feed-hold - abort sequence* has occurred.

Syntax:

```
q_where?
```

Side Effects:

This command is not appropriate when circular interpolation commands are used in the queue buffer, as the circular interpolation commands are comprised of varying numbers of *feed* entities.

This command is useful in determining the location in the queue buffer that a *feed-hold* operation has occurred. An *abort* action is necessary to allow the application program to access **Indexer LPT**. A typical sequence is as follows:

1) The queue buffer is executed by means of a *q_go* command.

2) Before completion, motion is arrested by means of the *feed-hold* input.

3) Control is returned to the application program by means of the *abort* input.

4) The *q_where?* command is written to **Indexer LPT**.

5) An ASCII numeric value representing a one (1) based index of the *feed* entity from which motion was arrested appears in the mailbox.

Returns:

If there are no remaining entities in the queue buffer, the following ASCII numeric value appears in the mailbox:

```
0
```

Otherwise, a one (1) based index into the *feed* entity list appears in the mailbox.

Example:

The following sequence is written to Indexer LPT

```
:
q_begin
feed:a,100,b,500
feed:a,200,b,10
```

```
feed:a,300,b,40
q_end
q_go
```

During the execution of the third *feed* command, the *feed-hold* input is activated, followed by the *abort* input. The following message appears in themailbox

```
abort
```

The following command is then written to Indexer LPT

```
q_where?
```

The message shown below appears in the mailbox

```
3
```

The "3" means that an abort from a feed-hold condition occurred within the third *feed* entry of the queue buffer.

Example:

The queue buffer is empty because it has not been loaded, or it has run to completion, or it has been reset using a *q_reset* command.

The following command is written to **Indexer LPT**:

```
q_where?
```

The message below, which is an ASCII zero character, appears in the mailbox:

```
0
```

REDUCED_CURRENT

Synopsis:

This command controls the *reduced current* output signal line.

Syntax:

`reduced_current:<axis><logic level>`

`<logic level>`	Meaning
`1`	output voltage level is TTL high
`0`	output voltage level is TTL low

Side Effects:

For X group axes (see Figure 1, page 27) this command controls the associated output voltage on connector pin 4. For Y group axes this command controls the associated output voltage on connector pin 8.

Returns:

An ASCII numeric character appears in the mailbox which indicates the logic level which has been established.

Example:

A computer system is configured such that a printer port which has the base address of 378 (hex) is used as an **Indexer LPT** motor controller. Consequently, pin 4 of this port connector corresponds to the *reduced current* pin for the “c” axis, and pin 8 of this port connector corresponds to the *reduced current* pin for the “d” axis. The following command causes pin 4 of this card to assume a TTL low voltage level:

```
reduced_current:c,0
```

After the execution of this command, the following ASCII character appears in the mailbox:

```
0
```

The following command causes pin 4 of this card to assume a TTL high voltage level:

```
reduced_current:c,1
```

After the execution of this command, the following ASCII character appears in the mailbox:

```
1
```

REDUCED_CURRENT?

Synopsis:

Read the logic level of the associated *reduced current* output line.

Syntax:

```
reduced_current?:<axis>
```

Returns:

The mailbox contains an ASCII numeric character:

`0` if the output voltage level is TTL low.

`1` if the output voltage level is TTL high.

Example:

Assume the *reduced current* output signal line for the “c “ axis has been previously set to a logic level of 1 (one). To make this value appear in the mailbox, the following command is issued:

```
reduced_current?:c
```

As a result the following ASCII character appears in the mailbox:

```
1
```

SAVE_FRO_ENABLE

Synopsis:

Save the current contents of the *fro_enable* register to the system **Registry**.

Syntax:

```
save_fro_enable
```

Side Effects:

The value saved will be restored to the *fro_enable* register the next time **Indexer LPT** loads.

Returns:

An ASCII numeric string representing the current value of the *fro_enable* register appears in the mailbox.]

SAVE_FRO_RES

Synopsis:

Save the contents of the *fro_res* register to the system **Registry**.

Syntax:

```
save_fro_res
```

Side Effects:

The value saved will be restored to the *fro_res* register the next time **Indexer LPT** loads.

Returns:

An ASCII numeric string representing the current value of the *fro_res* register appears in the mailbox.

SCAN

Synopsis:

Check for a closed circuit condition between the designated output and the designated input.

Syntax:

```
scan:<output axis>,<output bit>,
                  <input axis>,<input bit>
```

<output bit>	Signal Line Name
0	Step
1	Direction
2	Reduced Current
3	All Windings Off

<input bit>	Signal Line Name
0	Low Limit Switch Input
1	High Limit Switch Input
2	Auxiliary Input

Side Effects:

(Refer to the chapter entitled “Switch Scanning & Joystick” for a more complete description of the switch scanning procedure).

This command requires that the output axis be set to mode 2 using the *axis* command. When set to mode 2, all output bits associated with the output axis are normally "high" (5 volts).

The *scan* command accomplishes the following sequence:

1) The specified output bit is momentarily brought low (0 volts).

2) During the time period when the specified output bit is low, the specified input bit is sensed.

3) The output bit which was momentarily brought low is returned to its normally high condition.

Open circuit voltage on the limit switch inputs are considered stable because these signals are internally pulled up to 5 Volts through a resister on the printer card circuit. However, since on most print-

er cards the *auxiliary input* is a floating, it cannot be relied upon to read "high" during a open circuit condition without an external pull up resistor. To use as a scanning input, connect the *auxiliary input* line to 5 volts through of a 4.7K ohm resistor).

Returns:

If the input bit sensed to be "low" during the time that the output bit is "low", then the following message appears in the mailbox:

```
1
```

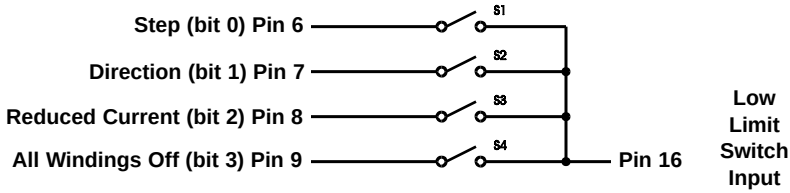
If the input bit sensed to be "high" during the time that the output bit is "low", then the following message appears in the mailbox:

```
0
```

Example:

Consider the wiring in the diagram below represents connections to the “e” axis. Also, assume that the “e” axis has been set up for scanning using the following command:

```
axis:e,2
```



Example Wiring for Axis “e”

The following command sequence occurs:

```
axis:e,2
scan:e,2,e,0
```

If switch **S3** is closed when the *scan* command is executed, the following message appears in the mailbox:

```
1
```

Otherwise, if switch **S3** is open, the following message is returned:

```
0
```

SET_ACCEL

Synopsis:

Set the *acceleration* register of the selected axis to the desired value in steps per second-per second.

Syntax:

```
set_accel:<axis>,<value>
```

Side Effects:

When, for example, a *move* command is issued, the dominant axis will begin moving at the rate set in the *lowspeed* register and accelerate towards the maximum velocity set in the *highspeed* register at an acceleration rate which is set by this command. Deceleration occurs at the same rate.

The value of the register cannot exceed the maximum capability of the system, nor can it be lower than the minimum value of 1 step per second-per second.

Returns:

If the new value is accepted, an ASCII string representing this value appears in the mailbox. If the new value is out of range the following message appears in the mailbox:

```
error,value
```

Example:

The *acceleration* register of the “a” axis is set such that subsequent *move* commands may accelerate the motor at a rate of 1500 steps per second-per second:

```
set_accel:a,1500
```

The following ASCII string appears in the mailbox:

```
1500
```

SET_ARCSEG_DEGREES

Synopsis:

Set the angle subtended over each segment of circular interpolation.

Syntax:

```
set_arcseg_degrees:<value>
```

Side Effects:

Circular interpolation commands, *circle*, *arc_to_point*, and *arc_to_angle*, use linear segments to approximate the true theoretical arc. Motion control follows the path of the chord of the arc segments designated by this command. The default value for each chordal segment is five (5) degrees. Consequently, using the default *arcseg_degrees* value, execution of a *circle* command would approximate a true circle with 360/5 = 72 linear segments.

The resolution of the arc is not affected by this command. For example, if the *arcseg_degrees* register is set to 5, and and the following command is issued:

```
arc_to_angle:ccw,a,0,b,1000,12
```

an arc of twelve (12) degrees will be subtended using three segments. Two segments of five (5) degrees, and on final segment of two (2) degrees.

The minimum value accepted by this command is one (1). Using the minimum value set up by this command of one (1) degree, a *circle* command would approximate a true circle using a polygon of 360 linear segments.

The maximum value accepted by this command is ninety (90). Using the maximum value set up by this command of ninety (90) degrees, the *circle* command would generate a polygon using of (4) linear segments.

Returns:

An ASCII sting reflecting the value of degrees per arc segment is placed in the mailbox

Example:

The following command sets the segment value to 2 degrees:

```
set_arcseg_degrees:2
```

The following string appears in the mailbox:

```
2
```

Subsequent to this command, an arc of 90 degrees is generated using the following command:

```
arc_to_angle:ccw,a,0,b,100,90
```

This command approximates the true theoretical arc using 90/2 = 45 linear segments.

SET_FEED_ACCEL

Synopsis:

Set the *feed acceleration* register to the desired value in steps per second-per second. This register applies to all axes.

Syntax:

```
set_feed_accel:<value>
```

Side Effects:

When a feed command is issued, the dominant axis (the axis which is to be moved the greatest extent) will begin motion at the rate set in the *feed_lowspeed* register and accelerate towards its final velocity at an acceleration rate determined by the contents of the *feed acceleration* register. Deceleration occurs at the same rate.

The value of the *feed acceleration* register cannot exceed the maximum capability of the system, nor can it be lower than a minimum value of 1 step per second-per second.

Returns:

If the new value is accepted, an ASCII string representing this value appears in the mailbox. If the new value is out of range the following message appears in the mailbox:

```
error,value
```

Example:

The *feed acceleration* register is set such that subsequent *feed* commands will accelerate the dominant axis at a rate of 1500 steps per second-per second:

```
set_feed_accel:1500
```

The following ASCII string appears in the mailbox:

```
1500
```

SET_FEED_HIGHSPEED

Synopsis:

The *feed_highspeed* register is modified using this command. This register applies to all axes.

Syntax:

```
set_feed_highspeed:<value>
```

Side Effects:

The maximum vector velocity after acceleration using the *feed* command is set to the desired value in steps per second.

If allowed sufficient distance to travel, the dominant axis of a *feed* command will accelerate such that the velocity along the path of travel will reach the vector rate specified by the value of the *feed_highspeed* register. The value of the *feed_highspeed* register cannot exceed the maximum capability of the system, nor can it be lower than the value set in the *feed_lowspeed* register. (The maximum speed the system is capable of can be determined using the *max_speed?* command.

Returns:

If the new value is accepted, an ASCII string representing this value appears in the mailbox. If the new value is out of range, or if the value is smaller than the *feed_lowspeed* value, the following message appears in the mailbox:

```
error, value
```

Example:

The *feed_highspeed* register is set such that subsequent *feed* commands may accelerate to a vector velocity of 1000 steps per second.

```
set_feed_highspeed:1000
```

The following ASCII string appears in the mailbox:

```
1000
```

After the following command is issued:

```
feed:a,50000,b,50000
```

the dominant axis accelerates to a rate of 707 steps per second, making the combined vector velocity 1000 steps per second.

SET_FEED_LOWSPEED

Synopsis:

The *feed_lowspeed* register is modified using this command. This register applies to all axes.

Syntax:

```
set_feed_lowspeed:<value>
```

Side Effects:

The starting *feed* velocity is set to the desired value in steps per second.

When a feed command is issued, the dominant axis begins motion at the rate set in the *feed_lowspeed* register. The default *feed_lowspeed* value is 250 steps per second. This register cannot be set to a value above the *feed_highspeed* register.

Returns:

If the new value is accepted, an ASCII string representing this value appears in the mailbox. If the new value is out of range, or if the value is greater than the *feed_highspeed* value, the following message appears in the mailbox:

```
error,value
```

Example:

Set the *feed_lowspeed* register such that the feed command will start the dominant axis at a rate of 500 steps per second:

```
set_feed_lowspeed:500
```

The following ASCII string appears in the mailbox:

```
500
```

SET_FRO_DELAY

Synopsis:

Change the value of the *fro_delay* register.

Syntax:

```
set_fro_delay:<value>
```

Side Effects:

The value of the *fro_delay* register affects the sensitivity of feed rate override controls. Lower values increase sensitivity. Higher values decrease sensitivity.

Returns:

An ASCII numeric string representing the value written to the *fro_delay* register appears in the mailbox.

SET_FRO_ENABLE

Synopsis:

Change the value of the *fro_enable* register.

Syntax:

```
set_fro_enable:<value>
```

Side Effects:

The *fro_enable* register can be set to a value of "1" or "0". A value of "1" enables the feed rate override feature. A value of "0" disables it.

Caution

The feed rate override feature should not be enabled if there exists an open circuit to the feed rate override input pin on the Hardware Assist Module, as this will result in erratic motor speeds.

The fro_enable register has no effect on software control over the FRO feature via manipulation of the fro_offset register.

Returns:

And ASCII numeric string representing the value written to the fro_enable register appears in the mailbox.

Example:

The following command is used to enable the feed rate override feature:

```
set_fro_enable:1
```

As a result, the following character string appears in the mailbox:

```
1
```

SET_FRO_HIGH

Synopsis:

Change the value of the *fro_high* register.

Syntax:

```
set_fro_high:<value>
```

Side Effects:

The value written represents the percentage of feed rate override that will be applied at the maximum limit.

Returns:

An ASCII numeric string representing the value written to the *fro_high* register appears in the mailbox.

SET_FRO_HIGHVOLT

Synopsis:

Change the value of the *fro_highvolt* register.

Syntax:

```
set_fro_highvolt:<value>
```

Side Effects:

The value written represents the input voltage at the high limit of the feed rate override input. This value is in units of 5/256 Volts, and cannot exceed 256 (corresponding to an upper limit of 5 Volts). This value must be greater than the value of the *fro_lowvolt* register.

Returns:

An ASCII numeric string representing the value written to the *fro_highvolt* register appears in the mailbox.

Example:

The following command sets the high limit voltage to 5.0 volts:

```
set_fro_highvolt:256
```

As a result, the number 256 is assigned to the *fro_highvolt* register and the following ASCII numeric string appears in the mailbox:

```
256
```

SET_FRO_LOW

Synopsis:

Change the value of the *fro_low* register.

Syntax:

```
set_fro_low:<value>
```

Side Effects:

The value written represents the percentage of feed rate override that will be applied at the minimum limit.

Returns:

An ASCII numeric string representing the value written to the *fro_low* register appears in the mailbox.

SET_FRO_LOWVOLT

Synopsis:

Change the value of the *fro_lowvolt* register.

Syntax:

```
set_fro_lowvolt:<value>
```

Side Effects:

The value written represents the input voltage at the low limit of the feed rate override input. This value is in units of 5/256 volts, and cannot exceed the value of the "fro_highvolt" register.

Returns:

An ASCII numeric string representing the value written to the *fro_lowvolt* register appears in the mailbox.

Example:

The following command sets the low limit voltage to approximately .02 Volts.

```
set_fro_lowvolt:5
```

As a result the the number 5, representing 5/256 = 0.01953 Volts, is assigned to the *fro_lowvolt* register and the following ASCII numeric string appears in the mailbox:

```
5
```

SET_FRO_OFFSET

Synopsis:

Change the value of the *fro_offset* register.

Syntax:

```
set_fro_offset:<value>
```

Side Effects:

The actual speed of operation will track the percentage override determined by the sum of the values of the *fro_offset* register and the operational (voltage controlled) feed rate override.

When the voltage controlled feed rate override (FRO) feature is disabled (contents of the fro_enable register is "0"), the operational FRO value is 100%. Otherwise, when FRO is enabled, the operational FRO value is determined by the external FRO input voltage.

The *set_fro_offset* command is acceptable for use within the queue buffer, and will take effect as part of the queue execution sequence (after *q_go*) provided there is at least one motion command preceding it. Otherwise, it will take effect immediately.

Using *set_fro_offset* within the queue buffer is an effective way to change vector speeds within a complex contour.

Example:

Assume that the FRO feature is disabled and the *feed_highspeed* register is 10000. Consider the following sequence:

```
q_begin
feed:a,82034
set_fro_offset:-20
feed:a,40000
q_end
q_go
```

Assume that during the first *feed* command acceleration to the value of the *feed_highspeed* register occurs. After the completion of the first *feed* command the motor speed will decelerate from its value of 10000 steps per second to the FRO value of 100%-20% = 80%, or 8000 steps per second.

If the FRO feature were enabled, after the *set_fro_offset* command the FRO value would be 20% less than the value established by the FRO input voltage,but will not be smaller than the value set up in the *fro_low* register.

SET_FRO_RES

Synopsis:

Change the value of the *fro_res* register.

Syntax:

```
set_fro_res:<value>
```

Side Effects:

The value of the *fro_res* register determines the number of intermediate speeds effected by feed rate override between the values set by the *fro_high* and *fro_low* registers.

Returns:

An ASCII numeric string representing the value written to the *fro_res* register appears in the mailbox.

SET_HIGHSPEED

Synopsis:

The *highspeed* register for the associated axis is modified using this command.

Syntax:

```
set_highspeed:<axis>,<value>
```

Side Effects:

The maximum *move* velocity after acceleration of the dominant axis is set to the desired value in steps per second.

If allowed sufficient distance of travel, the dominant axis of a *move* command will accelerate to a maximum rate specified by its associated *highspeed* register. The value of the *highspeed* register cannot exceed the maximum capability of the system, nor can it be lower than the value set in the associated *lowspeed* register. (The maximum speed the system is capable of appears in the mailbox after issuing a *max_speed?* command).

Returns:

If the new value is accepted, an ASCII string representing this value appears in the mailbox. If the new value is out of range, or if the value is smaller than the *lowspeed* value, the following message appears in the mailbox:

```
error,value
```

Example:

The *highspeed* register of the “a” axis is set such that subsequent *move* commands may accelerate the dominant axis to a maximum rate of 8500 steps per second:

```
set_highspeed:a,8500
```

The following ASCII string appears in the mailbox:

```
8500
```

SET_HOME

Synopsis:

Establish the *home* reference position for the selected axis.

Syntax:

```
set_home:<axis>
```

Side Effects:

This command initializes the selected axis for position tracking. It is also required if the *home* command is to be used.

Returns:

The following ASCII string, comprising four “zeros”, appears in the mailbox:

```
0000
```

SET_JOGSPEED

Synopsis:

The *jogspeed* register for the associated axis is modified using this command.

Syntax:

```
set_jogspeed:<axis>,<value>
```

Side Effects

The instantaneous *jog* velocity of the selected axis is set to the desired value in steps per second.The default *jogspeed* value is 250 steps per second.

Returns:

If the new value is accepted an ASCII string representing this value appears in the mailbox. If the new value is out of range the following message appears in the mailbox:

```
error,value
```

Example:

The *jogspeed* register of the “c” axis is set such that subsequent *jog* commands will cause the motor of the “c” axis to move at an instantaneous rate of 500 steps per second:

```
set_jogspeed:c,500
```

The following ASCII string appears in the mailbox:

```
500
```

SET_LOWSPEED

Synopsis:

The *lowspeed* register for the associated axis is modified using this command.

Syntax:

```
set_lowspeed:<axis>,<value>
```

Side Effects:

The starting *move* velocity of the selected axis is set to the desired value in steps per second.

The default *lowspeed* value is 250 steps per second for all axes. The *set_lowspeed* command allows this value to be changed to accommodate the dynamic characteristics of the application such as friction, stiction, and inertia. This register cannot be set to a value above the *highspeed* register.

Returns:

If the new value is accepted, an ASCII string representing this value appears in the mailbox. If the new value is out of range, or if the value is greater than the value in the associated *highspeed* register, the following message appears in the mailbox:

```
error,value
```

Example:

The *lowspeed* register “c” axis is set such that subsequent *move* commands will cause the motor to start moving at a rate of 500 steps per second:

```
set_lowspeed:c,500
```

The following ASCII string appears in the mailbox:

```
500
```

SET_Q_MEM

Synopsis:

Set the amount of memory to be used by the queue buffer.

Syntax:

```
set_q_mem:<value>
```

Side Effects:

Memory allocated for the queue buffer is used exclusively used by **Indexer LPT**, and is not shared with other programs in the system. The argument passed in this command designates the amount of memory requested. The amount of memory that Windows reserves will usually differ somewhat from the amount requested.

The amount of memory reserved using this command is written to the system Registry, and requested again the next time **Indexer LPT** loads. It therby remains in effect until this command is used again, even after the machine is turned off.

Returns:

The actual amount of memory that Windows reserved is returned in an ASCII numeric string, available to be read from the mailbox.

Example:

The following command is used to reserve five megabytes in the queue buffer:

```
set_q_mem:5000000
```

The mailbox returns the amount of memory actually allocated, such as:

```
5001216
```

SET_VSHIFT

Synopsis:

Set the value governing the maximum instantaneous velocity shift which can occur when a queue of commands is executed.

Syntax:

```
set_vshift:<value>
set_vshift:default
```

Side Effects:

During execution of a queue of commands, an instantaneous shift in velocity is usually necessary to maintain position integrity when passing from command to command. In order to avoid over-stressing an axis, the velocity at the transition point is regulated so that each axis does not exceed an allowable limit. This limit, called the *maximum instantaneous velocity shift*, is determined by the values of the *feed_lowspeed*, *feed_highspeed* and *vshift* registers.

The `<value>` represented in this command is in units of steps per second and represents one half the *maximum instantaneous velocity shift* which is allowed to occur at maximum running speed (*feed_highspeed*).

The *maximum allowable instantaneous velocity shift* is proportionally greater at lower speeds. The speeds at which the *maximum allowable instantaneous velocity shift* are greatest are those which are at or below the starting velocity (*feed_lowspeed*). At or below *feed_lowspeed*, the value of the *maximum allowable instantaneous velocity shift* is two times the starting velocity (two times *feed_lowspeed*).

The default value of the *vshift* register is calculated on the basis of the values in the *feed_lowspeed*, *feed_highspeed*, and *feed_acce*l registers. Whenever the values of these registers are changed, the default value of the *vshift* register is calculated and set.

Issuing the command:

```
set_vshift:default
```

also sets the value of the *vshift* register to the default setting.

The maximum value which can be entered into the *vshift* register using the *set_vshift* command is the current value of the *feed_lowspeed* register.

The command to command transition velocity is regulated for the purpose of limiting acceleration forces which may overstress the holding force of the step motors. The *vshift* register allows the reg-

ulation to be adjusted for systems having different torque/speed and inertia characteristics.

The optimum value chosen for the *vshift* register is often determined experimentally. Generally speaking, a higher value in the *vshift* register will result in more consistent vector speed throughout execution of the queue, and greater stresses when passing from command to command.

Returns:

If the new value is accepted, an ASCII string representing this value appears in the mailbox. Only whole number values are accepted. However, the default value may be a fractional value. In the case of the default setting, the numerical string returned represents the closest whole number value.

If the new value is out of range, the following message appears in the mailbox:

```
error,value
```

Example:

The following command is issued:

```
set_vshift:100
```

If the value of the *feed_lowspeed* register is at or above 100 steps per second, the *vshift* register is changed to 100, and the following ASCII string appears in the mailbox:

```
100
```

Example:

Assume the following settings: *feed_lowspeed* is 200, *feed_highspeed* is 1000, *vshift* is 100.

The following command sequence is executed:

```
q_begin
feed:a,1000
feed:a,-1000
q_end
q_go
```

Since axis “a” is reversing direction, it will decelerate to *feed_lowspeed* after the execution of the first *feed* command, then immediately begin execution of the second command. Consequently, the instantaneous shift in velocity at the completion

of the first command is two times the *feed_lowspeed* value, or 400 steps per second.

Example:

Assume the following settings: *feed_lowspeed* is 200, *feed_highspeed* is 1000, *vshift* is 100.

The following command sequence is executed:

```
q_begin
feed:a,10000,b,10000
feed:a,-8660,b,5000
q_end
q_go
```

Before control is passed from the first *feed* command to the next, the dominant axis automatically decelerates to avoid exceeding the maximum allowable shift in velocity.

Example:

Assume the following settings: *feed_lowspeed* is 200, *feed_highspeed* is 1000, *vshift* is 100.

The following command sequence is executed:

```
q_begin
feed:a,10000,b,1000
feed:a,10000,b,1050
q_end
q_go
```

When control is passed from the first *feed* command to the next, the dominant axis (the “a” axis) does not decelerate to stay within maximum limits on instantaneous rate changes at the transition point.

command: SN?

Synopsis:

Query the license serial number programmed into the **Hardware Assist Module**.

Syntax:

```
sn?
```

Returns:

An ASCII numeric string representing the user license serial number appears in the mailbox.

UNLOAD

Synopsis:

Unload the most memory intensive portion of **Indexer LPT**.

Syntax:

```
unload
```

Side Effects:

After execution of this command, the control program component, IXCTRLW.EXE, terminates, leaving only the file I/O interface portion of **Indexer LPT** in memory.

After IXCTRLW terminates, the following message appears in the mailbox:

```
pre-initialized
```

Indexer LPT may be re-started by executing IXCTRLW.EXE from a DOS command line, or by double snapping over it's Shortcut icon.

VSHIFT?

Synopsis:

Read the contents of the *vshift* register.

Syntax:

```
vshift?
```

Returns:

An ASCII numeric string representing the nearest whole number value of the *vshift* register appears in the mailbox. (Refer to the explanation of the *vshift* register in the section explaining the *set_vshift* command).

Example:

Assume the value in the *vshift* register is 500 steps per second when the following command is issued:

```
vshift?
```

As a result the following ASCII string appears in the mailbox:

```
500
```

Example:

Assume the *vshift* register contains a default value which is less than one step per second. (The only time the *vshift* register can contain a fractional value is by “default”). As a result of the *vshift?* command, the following ASCII value appears in the mailbox:

```
0
```

WINDING_POWER

Synopsis:

This command controls the *all windings off* output signal line.

Syntax:

```
winding_power:<axis>,<logic level>
```

<logic level>	Meaning
1	output voltage level is TTL high
0	output voltage level is TTL low

Side Effects:

For X group axes this command controls the associated output voltage on connector pin 5. For Y group axes this command controls the associated output voltage on connector pin 9.

Returns:

An ASCII numeric character appears in the mailbox which indicates the logic level which has been established.

Example:

A computer system is configured such that a printer port which has the base address of 378 (hex) is used as an **Indexer LPT** motor controller. Consequently, pin 5 of this port connector corresponds to the *all windings off* pin for the "c" axis, and pin 9 of this port connector corresponds to the *all windings off* pin for the "d" axis. The following command causes pin 5 of this card to assume a TTL low voltage level:

```
winding_power:c,0
```

After the execution of this command, the following ASCII character appears in the mailbox:

```
0
```

The following command causes pin 5 of this card to assume a TTL high voltage level:

```
winding_power:c,1
```

After the execution of this command, the following ASCII character appears in the mailbox:

```
1
```

WINDING_POWER?

Synopsis:

Read the logic level of the associated *all windings off* output line.

Syntax:

```
winding_power?:<axis>
```

Returns:

The mailbox contains an ASCII numeric character:

`0` if the output voltage level is TTL low.

`1` if the output voltage level is TTL high.

Example:

Assume the *winding power* output signal line for the “c “ axis has been previously set to a logic level of 1 one. To make this value appear in the mailbox, the following command is issued:

```
winding_power?:c
```

As a result the following ASCII character appears in the mailbox.

```
1
```

Chapter 12

RESPONSE MESSAGE

< ASCII Numeric Value >

An ASCII numeric string which represents the position of the axis relative to the reference *home* position appears in the mailbox as a result of the *move, feed, jog,* or *position?* command. Position tracking, which is initialized with the *set_home* command, must be in effect.

<ASCII Numeric Value>:<ASCII Numeric Value>

After completion of multiple axis moves, axis position information appears in the mailbox in the order in which the axes were designated on the command line. Information relating to each axis is separated by a colon (:).

abort

A motion command which was in the process of being suspended by means of the *feed hold* input was terminated by means of the *abort* input.

error,axis

The `<axis>` argument of a command was not ‘a’, ‘b’, ‘c’, ‘d’, ‘e’, or ‘f’.

error,disabled

An attempt has been made to use either the *feedhold?* or *abort?* command without first enabling the *feed hold* feature.

error,feedhold is enabled

An attempt has been made to change the mode of an axis which has been set up to support the *feed hold* feature. (First the *feed hold feature* must be disabled. Only then can the mode be changed to support motor control).

error,HAM

The **Hardware Assist Module** is either disconnected or malfunctioning.

error,mode

The mode of the axis is incompatible with the command which was issued. This error is generated, for example, if motor control was attempted on an axis set up for digital output, or if digital output (using the *bit* command) was attempted on an axis set up for motor control.

error,portmissing

The printer port associated with the selected axis is not installed in the system.

error,position

A *home* command was issued for an axis having un-initialized or invalid position tracking.

error,queue

An attempt has been made to queue a command unsuitable for queuing, or an operation has been attempted which is inconsistent with queue processing procedure.

error,queue full

Insufficient memory remains in the queue buffer.

error,syntax

The command was not recognized.

error,value

The <value> argument of a command was either out of the allowable range, or contradicts a setup value.

finished

Normal completion of a command which supports no other means of mailbox communication.

limit,<axis>,<direction>

This message is given instead of an ASCII numeric string if a limit switch closure interrupts a motion command. The value for <axis> is the axis which has been interrupted: ‘a’, ‘b’, ‘c’, ‘d’, ‘e’, or ‘f’.

The value for <direction> is the direction of movement: ‘+’, or ‘-’.

none

The *feed hold* feature was not enabled before execution of the *feedhold_input?* command.

not supported

Response to an inappropriate query, such as:

```
command_mem?:set_highspeed
```

pre-initialized

The file I/O interface device driver has been loaded, but the control program component of **Indexer LPT** has either terminated (through an *unload* command), or has not been run.

unknown position

This message is given instead of an ASCII numeric string because axis position tracking has not been initialized by the *set_home* command, or because position tracking has been lost due to a limit switch closure.