CHAPTER

8

TRIO BASIC COMMANDS

Motion and Axis Commands

ACC

Type: Axis Command

Syntax: **ACC(rate of acc)**

Note: This command is provided to aid compatibility with older Trio controllers. Acceleration rate and deceleration rate are recommended to be set with the **ACCEL** and **DECEL** axis parameters.

Description: Sets both the acceleration and deceleration rate simultaneously

Parameters:

rate of acc:	The units of the parameter are dependant on the **UNITS** axis parameter. The acceleration factor is entered in UNITS/SEC/SEC.

Example: **ACC(100)**

ADD_DAC

Type: Axis Command

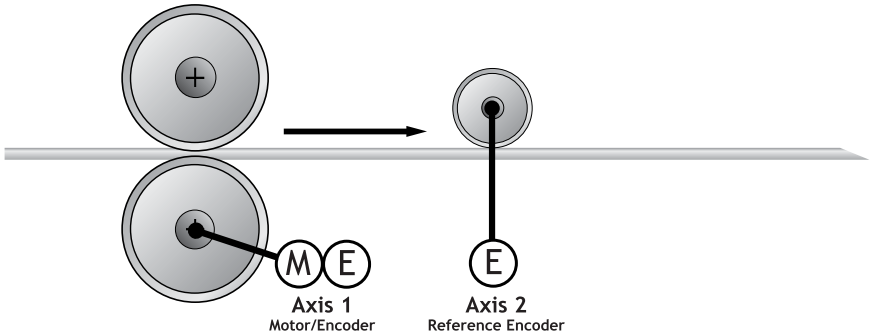
Syntax: **ADD_DAC(axis)**

Description: The **ADD_DAC** command is provided to allow a secondary encoder to be used on a servo axis to implement dual feedback control. This would typically be used in applications such as a roll-feed where you need a secondary encoder to compensate for slippage. The **ADD_DAC** function allows the output of 2 servo loops to be summed on to the speed demand output to a drive.

To use **ADD_DAC** it is necessary for the two axes with physical feedback to link to a common axis on which the required moves are executed. Typically this would be achieved by running the moves on a virtual axis and using **ADDAX** or **CONNECT** to produce a matching **DPOS** on BOTH axes. The servo algorithm gains are then set up on BOTH axes, and the output summed on to one physical output using **ADD_DAC**. Care should be taken if the required demand positions on both axes are not identical due to a difference in resolution between the 2 feedback devices.

Note that in the example below it would be necessary to set the **ATYPE** of the reference encoder axis as if it were a full servo axis. This is so that the software will perform the servo algorithm on that axis.

The **ADD_DAC** link can be terminated by setting **ADD_DAC(-1)**



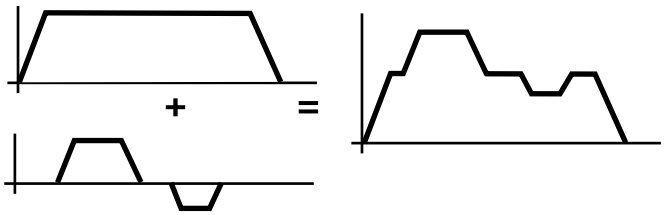
Example 1:
```
ADD_DAC(2) AXIS(1)   ' Combine axis(2) DAC with axis(1)
ADDAX(1) AXIS(2)   ' Superimpose axis 2 profile on axis 1
```

ADDAX

Type: Command

Syntax: **ADDAX(axis)**

Description: The **ADDAX** command is used to superimpose 2 or more movements to build up a more complex movement profile:



The **ADDAX** command takes the demand position changes from the specified axis and adds them to any movements running on the axis to which the command is issued. The specified axis can be any axis and does not have to physically exist in the system. After the **ADDAX** command has been issued the link between the two axes remains until broken and any further moves on the specified axis will be added to the base axis. To break the link an **ADDAX(-1)** command is issued.

The **ADDAX** command therefore allows an axis to perform the moves specified on TWO axes added together. When the axis parameter **SERVO** is set to **OFF** on an axis with an encoder interface the measured position **MPOS** is copied into the demand position **DPOS**. This allows **ADDAX** to be used to sum encoder inputs.

Parameter: **axis:** Axis to superimpose

Note: The **ADDAX** command sums the movements in encoder edge units.

Example 1:

```
UNITS AXIS(0)=1000
UNITS AXIS(1)=20
' Superimpose axis 1 on axis 0
ADDAX(1) AXIS(0)
MOVE(1) AXIS(0)
MOVE(2) AXIS(1)
'Axis 0 will move 1*1000+2*20=1040 edges
```

Example 2: Pieces are placed onto a continuously moving belt and further along the line are picked up. A detection system gives an indication as to whether a piece is in front of or behind its nominal position, and how far.

```
FORWARD AXIS(0)' set continuous move
ADDAX(2)
REPEAT
  GOSUB getoffset' Get offset to apply
  MOVE(offset) AXIS(2)
UNTIL IN(2)=ON
```

Axis 0 in this example executes a continuous **FORWARD** and a superimposed **MOVE** on axis 2 is used to apply offsets

Example 3: A **CAMBOX** movement is linked to an encoder input. In order to achieve registration a gradual offset is required to be applied *as if the link encoder is being moved*. This can be achieved by linking the **CAMBOX** to an unused imaginary axis on the controller. The encoder position is added to the imaginary axis with **ADDAX**. Offset moves to achieve the registration are run on the imaginary axis:

```
' Axis 0 runs CAMBOX linked to axis 2
' Axis 1 has encoder daughter board
' Axis 2 is "virtual" axis
SERVO AXIS(1)=OFF
ADDAX(1) AXIS(2)
...
CAMBOX(1000,1100,4,600,2) AXIS(0)
```

AXIS

Type: Modifier

Syntax: **AXIS(expression)**

Description: Assigns ONE command or axis parameter read or assignment to a particular axis.

Note: If it is required to change every subsequent command the **BASE** command should be used instead.

Parameters: **Expression:** Any valid Trio BASIC expression. The expression should be an axis number.

Example 1:
```
>>PRINT MPOS AXIS(3)
```

Example 2:
```
MOVE(300) AXIS(2)
```

Example 3:
```
REPDIST AXIS(3)=100
```

The **AXIS** command may be used to modify any axis parameter expression and the axis dependent commands: **ACC**, **ADDAX**, **CANCEL**, **CAM**, **CAMBOX**, **CONNECT**, **DATUM**, **DEC**, **DEFPOS**, **FORWARD**, **MOVEABS**,**MOVECIRC**, **MHELICAL**, **MOVELINK**, **MOVE**, **MOVEMODIFY**, **REGIST**,**REVERSE**, **SP**, **WAIT IDLE**, **WAIT LOADED**

See Also: **BASE()**

BASE

Type: Motion Command

Syntax: **BASE(axis no<,second axis><,third axis>...)**

Description: The **BASE** command is used to direct subsequent motion commands and axis parameter read/writes to a particular axis, or group of axes. The default setting is a sequence: zero, one, two...

Each process has its own BASE group of axes and each program can set values independently.

The Trio BASIC program is separate from the MOTION GENERATOR program which controls motion in the axes. The motion generator has separate functions for each axis, so each axis is capable of being programmed with its own speed, acceleration, etc. and moving independently and simultaneously OR they can be linked together by interpolation or linked moves.

The **AXIS()** command also redirects commands to different axes but applies to just the single command it precedes, and to a single axis. The **BASE()** command redirects all subsequent commands unless they are specified with **AXIS**.

Parameters:

axis numbers:	The number of the axis or axes to become the new base axis array, i.e. the axis/axes to send the motion commands to or the first axis in a multi axis command.

Example 1:

```
BASE(1)
UNITS=2000' unit conversion factor
SPEED=100'Set speed axis 1
ACCEL=5000'acceleration rate
BASE(2)
UNITS=2000'unit conversion factor
SPEED=125'Set speed axis 2
ACCEL=10000'acceleration rate
```

Example 2:
```
BASE(0,4,6)
MOVE(100,-23.1,1250)
```

Note: In example 2 axis 0 will move 100 units, axis 4 will move -23.1 and axis 6 will move 1250 units. The axes will move along the resultant path at the speed and acceleration set for axis 0.

Note 2: The **BASE** command sets an internal array of axes held for each process. The default array for each process is 0,1,2.up to the number of controller axes. If the **BASE** command does not specify all the axes, the **BASE** command will "fill in" the remaining values automatically. Firstly it will fill in any remaining axes above the last declared value, then it will fill in any remaining axes in sequence:

Example 3:
```
'Set BASE array on a 16 axis MC216 controller
BASE(2,6,10)
```

This will set the internal array of 16 axes to:

2,6,10,11,12,13,14,15,0,1,3,4,5,7,8,9

Note 3: On the command line process ONLY, the **BASE** array may be seen by typing:

```
>>BASE
(0,2,3,1,4,5,6,7)
>>
```

This example is from an MC206 with 8 axes.

CAM

Type: Axis Command

Syntax: **CAM(start point, end point, table multiplier, distance)**

Description: The **CAM** command is used to generate movement of an axis according to a table of POSITIONS which define a movement profile. The table of values is specified with the **TABLE** command. The movement may be defined with any number of points from 2 to 16000 (8000 on MC202). The controller interpolates between the values in the table to allow small numbers of points to define a smooth profile.

Parameters:

start point:	The cam table may be used to hold several profiles and/or other information. To allow freedom of use each command specifies where to start in the table.
end point:	Specifies end of values in table. Note that 2 or more **CAM()** commands executing simultaneously can use the same values in the table.
table multiplier:	The table values are absolute positions from the start of the motion and are normally specified in encoder edges. The table multiplier may be set to any value to scale the values in the table.
distance:	The distance factor controls the speed of movement through the table. The time taken to execute the **CAM()** command is dependent on the current axis **SPEED** and this distance (which is in user units).
	Say for example the system is being programmed in mm and the speed is set to 10mm/sec. If a distance of 100mm is specified the CAM command will take 10 seconds to execute. The speed may be changed at any time to any value as with other motion commands. The **SPEED** is ramped up to using the current **ACCEL** value. To obtain a **CAM** shape where **ACCEL** has no effect the value should be set to at least 1000 times the **SPEED** value (assuming the default **SERVO_PERIOD** of 1ms).

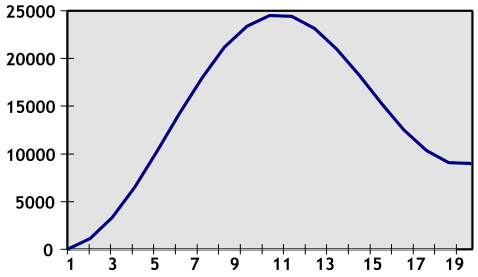
Example: Motion is required to follow the POSITION equation:

t(x) = x*25 + 10000(1-cos(x))
Where x is in degrees. This example table provides a simple oscillation superimposed with a constant speed. To load the table and cycle it continuously the program would be:

```
GOSUB camtable
loop:
    CAM (1,19,1,200)
GOTO loop
```

Note: The subroutine camtable loads the data into the cam TABLE, as shown in the graph below.

Table Position	Degrees	Value
1	0	0
2	20	1103
3	40	3340
4	60	6500
5	80	10263
6	100	14236
7	120	18000
8	140	21160
9	160	23396
10	180	24500
11	200	24396
12	220	23160
13	240	21000
14	260	18236
15	280	15263
16	300	12500
17	320	10340
18	340	9103
19	360	9000



Note 2: When the **CAM** command is executing the **ENDMOVE** parameter is set to the end of the PREVIOUS move

CAMBOX

Type: Axis Command

Syntax: **CAMBOX(start point, end point, table multiplier, link distance , link axis<,link options><, link pos>)**

Description: The **CAMBOX** command is used to generate movement of an axis according to a table of POSITIONS which define the movement profile. The motion is linked to the measured motion of another axis to form a continuously variable software gearbox. The table of values is specified with the **TABLE** command. The movement may be defined with any number of points from 2 to 16000 (8000 on MC202). The controller interpolates between the values in the table to allow small numbers of points to define a smooth profile.

Parameters:

start point:	The cam table may be used to hold several profiles and/or other information. To allow freedom of use each command specifies where to start in the table.
end point:	Specifies end of values in table. Note that 2 or more **CAMBOX** commands executing simultaneously can use the same values in the table.
table multiplier:	The table values are absolute positions from the start of the motion and are specified in encoder edges units. The table multiplier may be set to any value to scale the values in the table.
link distance:	The link distance specifies the distance the link axis must move to complete the specified output movement. The link distance is in the user units of the link axis and should always be specified as a positive distance.
link axis:	This parameter specifies the axis to link to. It should be set to 0..7 (MC206), 0..15 (MC216), 0..7 (Euro205), 0..2 (MC202)

link options: Bit Values:

1 - link commences exactly when registration event occurs on link axis

2 - link commences at an absolute position on link axis (see param 7)

4 - CAMBOX repeats automatically and bi-directionally when this bit is set. (This mode can be cleared by setting bit 1 of the REP_OPTION axis parameter)

8 - PATTERN mode. Advanced use of cambox: allows multiple scale values to be used. Normally combined with the automatic repeat mode. See example 4.

Note:

The start options (1 and 2) may be combined with the repeat-options (4 and 8).

link pos: This parameter is the absolute position where the **CAMBOX** link is to be started when parameter 6 is set to 2.

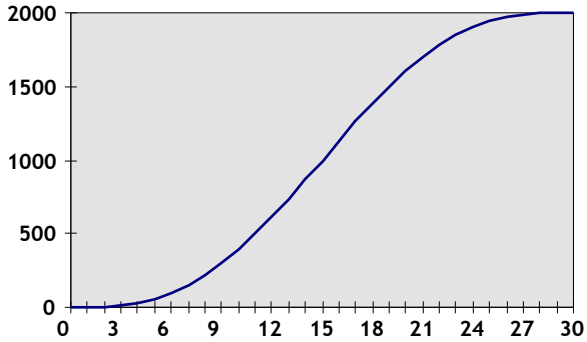
Note: When the **CAMBOX** command is executing the **ENDMOVE** parameter is set to the end of the PREVIOUS move. The **REMAIN** axis parameter holds the remainder of the distance on the link axis.

Parameters 6 and 7; link options and link pos, are optional.

Example 1:

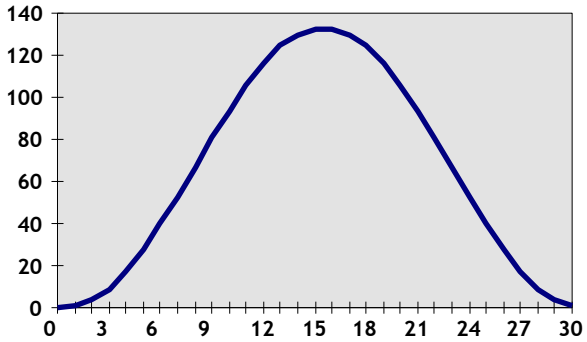
```
num_p=30
scale=2000
'
' Subroutine to generate a SIN shape speed profile
'
' Uses: p is loop counter
' num_p is number of points stored in tables pos 0..num_p
' scale is distance travelled scale factor

FOR p=0 TO num_p
  TABLE(p,((-SIN(PI*2*p/num_p)/(PI*2))+p/num_p)*scale)
NEXT p
```



This graph plots **TABLE** contents against table array position. This corresponds to motor POSITION against link POSITION when called using **CAMBOX**. The SPEED of the motor will correspond to the derivative of the position curve above:

Speed Curve



Example 2: A rotating drum feeding labels is activated when a product conveyor reaches a position held in the variable “start”. This example uses the table points 0.30 generated in Example 1:

```
CAMBOX(0,30,800,80,15,2,start)
```

Note:

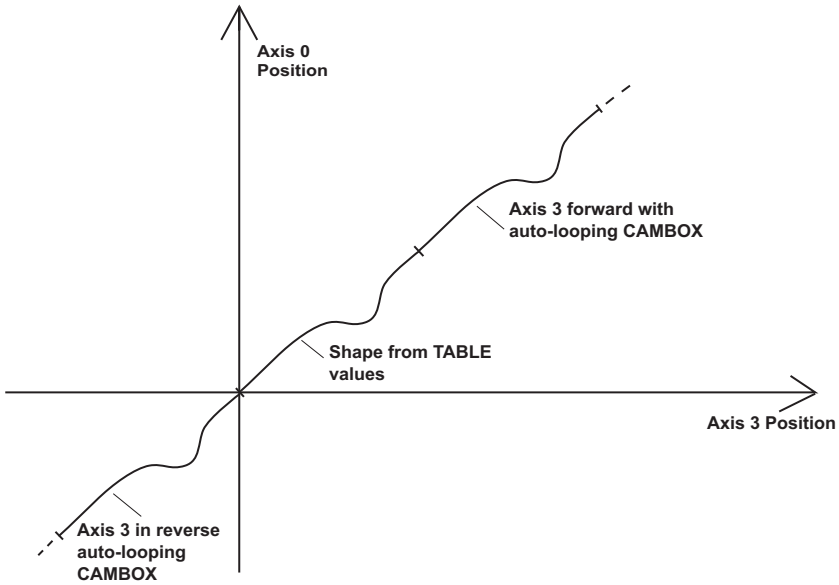
0	The start of the profile shape in the **TABLE**
30	The end of the profile shape in the **TABLE**

800	This scales the **TABLE** values. Each **CAMBOX** motion would therefore total 800*2000 encoder edges steps.
80	The distance on the product conveyor to link the motion to. The units for this parameter are the programmed distance units on the link axis.
15	This specifies the axis to link to.
2	This is the link option setting - Start at absolute position on the link axis.
"start"	variable "start". The motion will execute when the position "start" is reaches on axis 15.

Example 3: A motor on Axis 0 is required to emulate a rotating mechanical **CAM**. The position is linked to motion on axis 3. The "shape" of the motion profile is held in **TABLE** values 1000..1100.

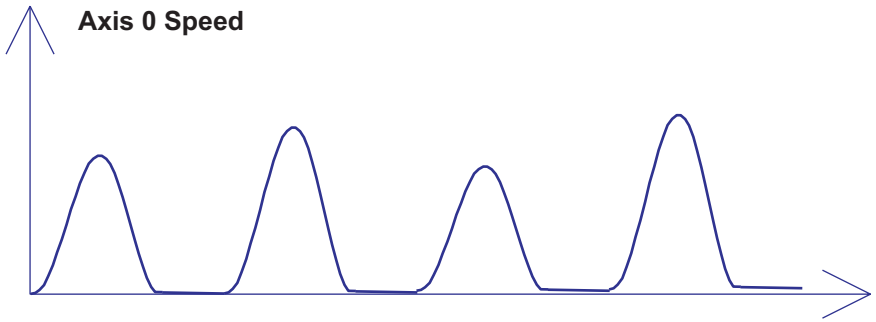
```
CAMBOX(1000,1100,45.68,360,3,4) AXIS(0)
REP_OPTION=2'    cancel repeating mode.
```

Note: The system software resets bit 1 of **REP_OPTION** to 0 when the mode has been cancelled.



Example 4: CAMBOX Pattern Mode

Setting bit 3 (value 8) of the link options parameter enables the **CAMBOX** pattern mode. This mode enables a sequence of scale values to be cycled automatically. This is normally combined with the automatic repeat mode, so the options parameter should be set to 12. This diagram shows a typical repeating pattern which can be automated with the CAMBOX pattern mode:



The parameters for this mode are treated differently to the standard **CAMBOX** function

```
CAMBOX(start, end, control block pointer, link dist, link axis,
options)
```

The start and end parameters specify the basic shape profile ONLY. The pattern sequence is specified in a separate section of the **TABLE** memory. There is a new **TABLE** block defined: The “Control Block”. This block of seven **TABLE** values defines the pattern position, repeat controls etc. The block is fixed at 7 values long.

Therefore in this mode only there are 3 independently positioned **TABLE** blocks used to define the required motion:

SHAPE BLOCK This is directly pointed to by the **CAMBOX** command as in any **CAMBOX**.

CONTROL BLOCK This is pointed to by the third **CAMBOX** parameter in this options mode only. It is of fixed length (7 table values). It is important to note that the control block is modified during the **CAMBOX** operation. It must therefore be re-initialised prior to each use.

PATTERN BLOCK The start and end of this are pointed to by 2 of the CONTROL BLOCK values. The pattern sequence is a sequence of scale factors for the SHAPE.

Control Block Parameters

		R/W	Description
0	CURRENT POSITION	R	The current position within the TABLE of the pattern sequence. This value should be initialised to the START PATTERN number.
1	FORCE POSITION	R/W	Normally this value is -1. If at the end of a SHAPE the user program has written a value into this TABLE position the pattern will continue at this position. The system software will then write -1 into this position. The value written should be inside the pattern such that the value: CB(2)<=CB(1)<=CB(3)
2	START PATTERN	R	The position in the **TABLE** of the first pattern value.
3	END PATTERN	R	The position in the **TABLE** of the final pattern value
4	REPEAT POSITION	R/W	The current pattern repeat number. Initialise this number to 0. The number will increment when the pattern repeats if the link axis motion is in a positive direction. The number will decrement when the pattern repeats if the link axis motion is in a negative direction. Note that the counter runs starting at zero: 0,1,2,3…
5	REPEAT COUNT	R/W	Required number of pattern repeats. If -1 the pattern repeats endlessly. The number should be positive. When the ABSOLUTE value of CB(4) reaches CB(5) the **CAMBOX** finishes if CB(6)=-1. The value can be set to 0 to terminate the **CAMBOX** at the end of the current pattern. See note below on REPEAT COUNT in the case of negative motion on the link axis.
6	NEXT CONTROL BLOCK	R/W	If set to -1 the pattern will finish when the required number of repeats are done. Alternatively a new control block pointer can be used to point to a further control block.

Note: READ/WRITE values can be written to by the user program during the pattern **CAMBOX** execution.

Example: A machine cycles an initialisation shape cycle 1000 times prior to running a pattern continuously until requested to stop at the end of the pattern.

The same shape is used for the initialisation cycles and the pattern. This shape is held in **TABLE** values 100..150

The running pattern sequence is held in **TABLE** values 1000..4999

The initialisation pattern is a single value held in **TABLE(160)**

The initialisation control block is held in **TABLE(200)..TABLE(206)**

The running control block is held in **TABLE(300)..TABLE(306)**

```
' Set up Initialisation control block:
TABLE(200,160,-1,160,160,0,1000,300)

' Set up running control block:
TABLE(300,1000,-1,1000,4999,0,-1,-1)

' Run whole lot with single CAMBOX:
' Third parameter is pointer to first control block

CAMBOX(100,150,200,5000,1,20)
WAIT UNTIL IN(7)=OFF

TABLE(305,0) ' Set zero repeats: This will stop at end of
pattern
```

Note: Negative motion on link axis:

The axis the **CAMBOX** is linked to may be running in a positive or negative direction. In the case of a negative direction link the pattern will execute in reverse. In the case where a certain number of pattern repeats is specified with a negative direction link, the first control block will produce one repeat less than expected. This is because the **CAMBOX** loads a zero link position which immediately goes negative on the next servo cycle triggering a REPEAT COUNT. This effect only occurs when the **CAMBOX** is loaded, not on transitions from CONTROL BLOCK to CONTROL BLOCK. This effect can easily be compensated for either by increasing the required number of repeats, or setting the initial value of REPEAT POSITION to 1.

CANCEL

Type: Motion Command

Alternate Format: CA

Syntax: CANCEL / CANCEL(1)

Description: Cancels a move on an axis or an interpolating axis group. Velocity profiled moves (FORWARD, REVERSE, MOVE,MOVEABS,MOVECIRC, MHELICAL,MOVEMODIFY) will be ramped down at the programmed deceleration rate then terminated. Other move types will be terminated immediately.

CANCEL(1) clears a buffered move, leaving the current executing movement intact.

Note: Cancel will only cancel the presently executing move. If further moves are buffered they will then be loaded.

See also: RAPIDSTOP.

Example:
```
FORWARD
WA(10000)
CANCEL' stop movement after 10 seconds
```

Example 2:
```
MOVE(1000)
MOVEABS(3000)
' now change your mind:
’ move to 4000 not 3000
CANCEL(1)
MOVEABS(4000)
’ MOVEMODIFY would be better for this !
```

CONNECT

Type: Axis Command

Syntax: CONNECT(ratio , driving axis)

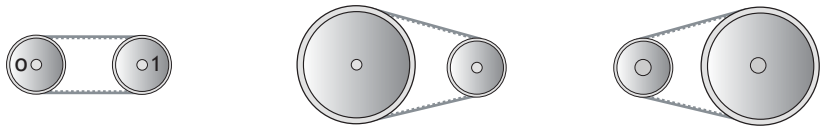
Alternate Format: CO(ratio, driving axis)

Description: CONNECT the demand position of the base axis to the measured movements of the driving axes to produce an electronic gearbox.

Parameters:

ratio:	This parameter holds the number of edges the base axis is required to move per increment of the driving axis. The ratio value can be either positive or negative and has sixteen bit fractional resolution. The ratio is always specified as an encoder edge ratio.
driving axis:	This parameter specifies the axis to link to.

Note: The ratio can be changed at any time by issuing another CONNECT command which will automatically update the ratio without the previous CONNECT being cancelled. The command can be cancelled with a CANCEL or RAPIDSTOP command.



CONNECT(1,1) CONNECT(2,1) CONNECT(0.5,1)

Example: In a press feed a roller is required to rotate at a speed one quarter of the measured rate from an encoder mounted on the incoming conveyor. The roller is wired to the master axis 0. An encoder daughter board monitors the encoder pulses from the conveyor and forms axis 1.

```
SERVO AXIS(1)=OFF ' This axis is used to monitor the conveyor
SERVO=ON
CONNECT(0.25,1)
```

Note 2: To achieve an exact connection of fractional ratio's of values such as 1024/3072. The MOVELINK command can be used with the continuous repeat link option set on.

DATUM

Type: Command

Syntax: **DATUM(sequence no)**

Description: Performs one of 7 datuming sequences to locate an axis to an absolute position. The creep speed used in the sequences is set using **CREEP**. The programmed speed is set with the **SPEED** command.

Parameter:

Seq.	Description
0	The current measured position is set as demand position (this is especially useful on stepper axes with position verification). The DATUM(0) command clears bit 1 (Following error warning), 2 (Remote drive comms error), 3 (Remote drive error), 8 (Following error) and 11 (Cancelling move) in the AXISSTATUS on ALL axes.
1	The axis moves at creep speed forward till the Z marker is encountered. The Demand position is then reset to zero and the Measured position corrected so as to maintain the following error.
2	The axis moves at creep speed in reverse till the Z marker is encountered. The Demand position is then reset to zero and the Measured position corrected so as to maintain the following error.
3	The axis moves at the programmed speed forward until the datum switch is reached. The axis then moves backwards at creep speed until the datum switch is reset. The Demand position is then reset to zero and the Measured position corrected so as to maintain the following error.
4	The axis moves at the programmed speed reverse until the datum switch is reached. The axis then moves at creep speed forward until the datum switch is reset. The Demand position is then reset to zero and the Measured position corrected so as to maintain the following error.

Seq.	Description
5	The axis moves at programmed speed forward until the datum switch is reached. The axis then reverses at creep speed until the datum switch is reset. It then continues in reverse looking for the Z marker on the motor. The demand position where the Z input was seen is then set to zero and the measured position corrected so as to maintain the following error.
6	The axis moves at programmed speed reverse until the datum switch is reached. The axis then moves forward at creep speed until the datum switch is reset. It then continues forward looking for the Z marker on the motor. The demand position where the Z input was seen is then set to zero and the measured position corrected so as to maintain the following error.

Note: The datuming input set with the **DATUM_IN** is active low so is set when the input is OFF. This is similar to the **FWD**,**REV** and **FHOLD** inputs which are designed to be "fail-safe".

Example:
```
DATUM_IN=10
DATUM(5)
```

DEC

Type: Axis Command

Syntax: **DEC(rate)**

Description: Sets the deceleration rate for an axis. The **DEC()** command is provided to maintain compatibility with older controllers. The Axis Parameter **DECEL** provides the same functionality and is preferred.

Parameters: **rate:** The units of the parameter are dependant on the unit conversion factor. The deceleration factor is entered in UNITS/SEC/SEC.

Note: **ACC()** sets both the acceleration and deceleration rates. **DEC()** sets only the deceleration rate.

DEFPOS

Type: Motion Command.

Syntax: **DEFPOS(pos1 [,pos2[, pos3[, pos4.....]]])**

Alternate Format: **DP(pos1 [,pos2[, pos3[, pos4]]])**

Description: Defines the current position as a new absolute value. The command is typically used after a **DATUM** sequence as these always set a datum of zero. It may however be used at any time, even whilst a move is in progress.

Parameters:

pos1: Absolute position to set on current base axis in user units.

pos2: Abs. position to set on the next axis in **BASE** array in user units.

pos3: Abs. position to set on the next axis in **BASE** array in user units.

Note: As many parameters as axes on the system may be specified.

Example:

```
DATUM(5)
BASE(2)
DATUM(4)
BASE(1)
WAIT IDLE
DEFPOS(-1000,-3500)
```

The last line defines the current position, reset to (0,0) by the two **DATUM** statements as (-1000,-3500) in user units.

Note: See also **OFFPOS** which performs a relative adjustment of position.

Note 2: Changes to the axis positions made via **DEFPOS** and **OFFPOS** are made on the next servo update. This can potentially cause problems as the user program may continue to execute commands after the **DEFPOS** is complete, but before the next servo update. For example, the following sequence could easily fail to move to the correct absolute position because the **DEFPOS** will not have been completed when the **MOVEABS** is loaded.

Example 2:

```
DEFPOS(100)
MOVEABS(0)' DEFPOS may not have occurred yet
```

DEFPOS statements are internally converted into **OFFPOS** position offsets which provide an easy way to avoid the problem described:

```
DEFPOS(100)
WAIT UNTIL OFFPOS=0' Ensures DEFPOS is complete before next line
MOVEABS(0)
```

FORWARD

Type: Axis Command

Alternate Format: `FO`

Description: Sets continuous forward movement.

Note: The forward motion can only be stopped by issuing a `CANCEL`, `RAPIDSTOP` or by hitting the forward, or datum limits.

Example:
```
start:
FORWARD
'WAIT FOR STOP SIGNAL
WAIT UNTIL IN(0)=ON
CANCEL
```

MATCH

Type: Axis Command

Syntax: `MATCH(count, table address)`

Description: Instructs the controller to perform a pattern comparison between a stored pattern of registration input transitions and the pattern held following a `REGIST` command.

Parameters:

`count`	Number of Transitions to include in the pattern match.
`table address`	Address where pattern to use for comparison has been recorded.

See also: `REGIST` and `RECORD`

Example:
```
dec_dist=SPEED*SPEED*0.5/DECEL
length=10
REP_DIST=100*length
DEFPOS(0)
REGIST(5,length)
MOVE(2*length)
WAIT UNTIL MARK
IF TRANSITIONS>4 AND TRANSITIONS<12 THEN
  MATCH(8,10)
```

```
  IF REG_MATCH>0.8 THEN
    IF REG_POS<(MPOS+dec_dist-length) THEN
      MOVEMODIFY(2*length+REG_POS)
    ELSE
      MOVEMODIFY(length+REG_POS)
    ENDIF
  ELSE
    PRINT "marks give too poor fit"
  ENDIF
ELSE
  PRINT "marks not seen"
ENDIF
WAIT IDLE
PRINT REG_POS,ENDMOVE,TRANSITIONS,REG_MATCH
```

MHELICAL

Type: Motion Command.

Syntax: **MHELICAL(end1, end2, centre1,centre2, direction, distance 3)**

Alternate Format: **MH()**

Description: Performs a helical move.

Moves 2 orthogonal axes in such a way as to produce a circular arc at the tool point with a simultaneous linear move on a third axis. The first 5 parameters are similar to those of an **MOVECIRC()** command. The sixth parameter defines the simultaneous linear move. Finish 1 and centre 1 are on the current **BASE** axis. Finish 2 and centre 2 are on the following axis. The first 4 distance and the sixth parameter are scaled according to the current unit conversion factor for each axis.

Parameters:

end1:	position on **BASE** axis to finish at.
end2:	position on next axis in **BASE** array to finish at.
centre1:	position on **BASE** axis about which to move.
centre2:	position on next axis in **BASE** array about which to move.

direction: The "direction" is a software switch which determines whether the arc is interpolated in a clockwise or anti- clockwise direction. The parameter is set to 0 or 1. See **MOVECIRC**.

distance3: The distance to move on the third axis in the **BASE** array axis in user units

MOVE

Syntax: **MOVE(distance1 [,distance2[,distance3[,distance4...]]])**

Type: Motion Command

Alternate Format: **MO()**

Description: Incremental move. Axis or axes move at the programmed speed and acceleration to a position specified as an increment from the end of the last specified move.

Note: The **MOVE** command can interpolate up to 3 axes on MC202, 8 axes on MC206 or Euro205, and 16 axes on MC216. The values specified are scaled using the UNIT CONVERSION FACTOR, axis parameter **UNITS**. Therefore if, for example, an axis has 4000 encoder edges/mm then UNITS for that axis are 4000. The command **MOVE(12.5)** would move 12.5 mm. The first parameter in the list is sent to the **BASE** axis or can be re-directed with the **AXIS()** command, the second to the next axis in the BASE array, etc. By changing the axis uninterpolated, unsynchronised multi-axis motion can be achieved. Incremental moves can be merged together for profiled continuous path movement. **MERGE** should be set to **ON**. In multi-axis systems the speed and acceleration employed for the movement are taken from the first axis in the group.

Parameters:

distance1: distance to move on base axis from current position.

distance2: distance to move on next axis in BASE array from current position.]

[distance3: distance to move on next axis in BASE array from current position.]

[distance4: distance to move on next axis in BASE array from current position.]

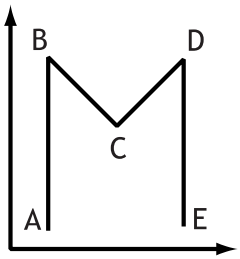
The number of parameters can increase to the number of axes on the controller

Example 1: A system is working with a unit conversion factor of 1 and has a 1000 line encoder. It is therefore necessary to give the instruction **MOVE(40000)** to incrementally move 10 turns on the motor. (A 1000 line encoder gives 4000 edges/turn)

Example 2:
```
MOVE(10) AXIS(5)
MOVE(10) AXIS(4)
MOVE(10) AXIS(3)
```

In this example axes 3,4 and 5 are moving independently (without interpolation). Each axis will move at its programmed **SPEED** etc.

Example 3: An X-Y plotter can write text at any position within its working envelope. Individual characters are defined as a sequence of moves relative to a start point so that the same commands may be used no matter what the plot position. The command subroutine for the letter 'M' might be:




```
m:
  MOVE(0,12)'move A >  B
  MOVE(3,-6)'move B >  C
  MOVE(3,6)' move C >  D
  MOVE(0,-12)'move D > E
```

MOVEABS

Type: Motion Command.

Syntax: **MOVEABS(1 pos [, 2 pos , 3 pos[, 4 pos...]]])**

Alternate Format: **MA()**

Description: Absolute position move. Move an axis or axes to position(s) referenced to the zero position.

Parameters:

1 pos: position to move to on base axis.

2 pos: position to move to on next axis in BASE array.

3 pos: position to move to on next axis in BASE array.

n pos: position to move to on next axis in BASE array

Note: The **MOVEABS** command can interpolate up to the full number of axes available on the controller.

The values specified are scaled using the UNIT CONVERSION FACTOR, axis parameter **UNITS**. Therefore if, for example, an axis has 4000 encoder edges/mm the **UNITS** for that axis is 4000. The command **MOVEABS(6)** would move to a position 6 mm from the zero position.

The first parameter in the list is sent to the axis specified with the AXIS command or to the current **BASE** axis, the second to the next axis, etc. By changing the **BASE** axis uninterpolated, unsynchronised multi-axis motion can be achieved. Absolute moves can be merged together for profiled continuous path movement. Axis parameter **MERGE** should be set to ON. In multi-axis systems the speed, acceleration and deceleration employed for the movement are taken from the BASE AXIS for the group.

Note2: The position of the axis zero positions can be moved by the commands: **OFFPOS**,**DEFPOS**,**REP_DIST**,**REP_OPTION**, and **DATUM**.

Example 1: An X-Y plotter has a pen carousel whose position is fixed relative to the plotter absolute zero position. To change pen an absolute move to the carousel position will find the target irrespective of the plot position when commanded.

```
MOVEABS(20,350)
```

Example 2: A pallet consists of a 6 by 8 grid in which gas canisters are inserted 85mm apart by a packaging machine. The canisters are picked up from a fixed point. The first position in the pallet is defined as position 0,0 using the **DEFPOS()** command. The part of the program to position the canisters in the pallet is:

```
FOR x=0 TO 5
  FOR y=0 TO 7
    'MOVE TO PICK UP POINT:
     MOVEABS(-340,-516.5)
    'PICK UP SUBROUTINE:
    GOSUB pick
    PRINT "MOVE TO POSITION: ";x*6+y+1
```

```
            MOVEABS(x*85,y*85)
            'PLACE DOWN SUBROUTINE:
            GOSUB place
        NEXT y
NEXT x
```

MOVECIRC

Syntax: **MOVECIRC(finish1, finish2, centre1, centre2, direction)**

Type: Motion Command.

Alternate Format: **MC()**

Description: Moves 2 orthogonal axes in such a way as to produce a circular arc at the tool point. The length and radius of the arc are defined by the five parameters in the command line. The move parameters are always incremental from the end of the last specified move. This is the start position on the circle circumference. Axis 1 is the current **BASE** axis. Axis 2 is the following axis in the **BASE** array. The first 4 distance parameters are scaled according to the current unit conversion factor for each axis.

Parameters:

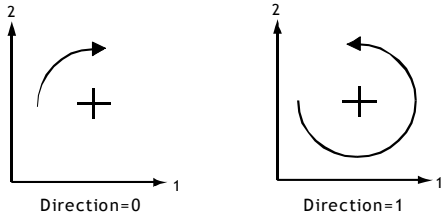
finish1: position on BASE axis to finish at.

finish2: position on next axis in BASE array to finish at.

centre1: position on BASE about which to move.

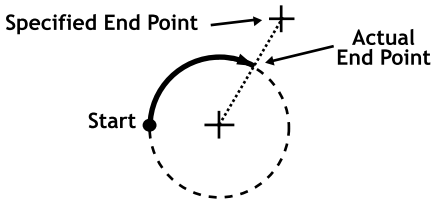
centre2: position on next axis in **BASE** array about which to move.

direction: The "direction" is a software switch which determines whether the arc is interpolated in a clockwise or anti- clockwise direction.



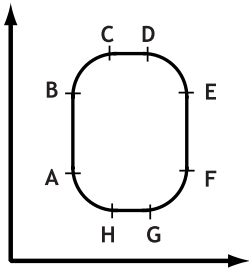
Note: In order for the **MOVECIRC()** command to be correctly executed, the two axes generating the circular arc must have the same number of encoder pulses/linear axis distance. If this is not the case it is possible to adjust the encoder scales in many cases by adjusting with **PP_STEP**.

Note2: If the end point specified is not on the circular arc. The arc will end at the angle specified by a line between the centre and the end point.



Example: The command sequence to plot the letter '0' might be:

```
MOVE(0,6)'      move A -> B
MOVECIRC(3,3,3,0,1)' move B -> C
MOVE(2,0)'      move C -> D
MOVECIRC(3,-3,0,-3,1)' move D -> E
MOVE(O,-6)'  move E -> F
MOVECIRC(-3,-3,-3,0,1)' move F -> G
MOVE(-2,0)'  move G -> H
MOVECIRC(-3,3,0,3,1)' move H -> A
```



MOVELINK

Syntax: **MOVELINK (distance, link dist, link acc, link dec, link axis[, link options] [, link start]).**

Type: Motion Command.

Alternate Format: **ML()**

Description: The linked move command is designed for controlling movements such as:

- Synchronization to conveyors
- Flying shears
- Thread chasing, tapping etc.
- Coil winding

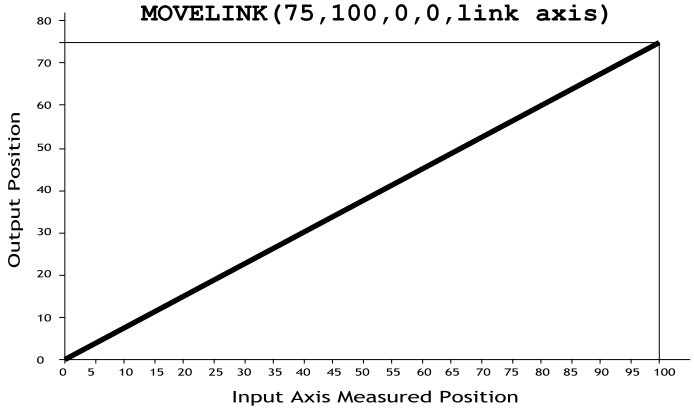
The motion consists of a linear movement with separately variable acceleration and deceleration phases linked via a software gearbox to the MEASURED position (**MPOS**) of another axis.

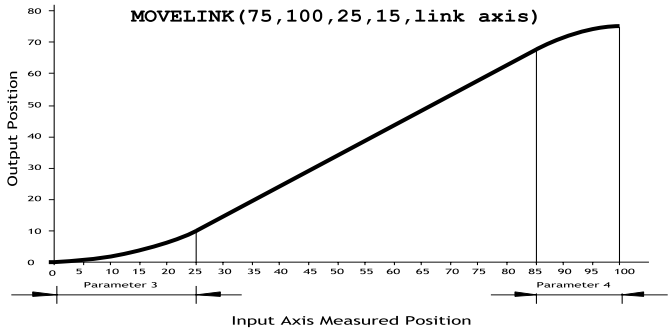
Parameters:

distance:	incremental distance in user units to be moved on the current base axis, as a result of the measured movement on the "input" axis which drives the move.
link dist:	positive incremental distance in user units which is required to be measured on the "link" axis to result in the motion on the base axis.
link acc:	positive incremental distance in user units on the input axis over which the base axis accelerates.
link dec:	positive incremental distance in user units on the input axis over which the base axis decelerates. N.B. If the sum of parameter 3 and parameter 4 is greater than parameter 2, they are both reduced in proportion until they equal parameter 2.
link axis:	Specifies the axis to "link" to. It should be set to 0...number of available axes
link options:	**1** link commences exactly when registration event occurs on link axis **2** link commences at an absolute position on link axis (see **link pos** parameter) **4** **MOVELINK** repeats automatically and bi-directional when this bit is set. (This mode can be cleared by setting bit 1 of the **REP_OPTION** axis parameter)
link pos:	This parameter is the absolute position where the **MOVELINK** link is to be started when parameter 6 is set to 2.

Note: The command uses the **BASE()** and **AXIS()**, and unit conversion factors in a similar way to other move commands.
The "link" axis may move in either direction to drive the output motion. The link distances specified are always positive.

Note 2: Parameters 6 and 7 are optional.





Example 2: A flying shear cuts a roll of paper every 160m whilst moving at the speed of the paper. The shear is able to travel up to 1.2 metres of which 1m is used in this example. The paper distance is measured by an encoder, the unit conversion factor being set to give units of metres on both axes: (Note that axis 7 is the link axis)

```
MOVELINK(0,150,0,0,7)'wait distance
MOVELINK(0.4,0.8,0.8,0,7)'accelerate
MOVELINK(0.6,1.0,0,0.8,7)'match speed then decel
WAIT LOADED'wait till previous move started
OP(8,ON)'activate cutter
MOVELINK(-1,8.2,0.5,0.5,7)'move back
```

In this program the controller firstly waits for the roll to feed out 150m in the first line. After this distance the shear accelerates up to match the speed of the paper coasts at the same speed then decelerates to a stop within the 1m stroke. This movement is specified using two separate **MOVELINK** commands. The program then waits for the next move buffer to be clear **NTYPE=0**. This indicates that the acceleration phase is complete. Note that the distances on the measurement axis (link distance in each **MOVELINK** command): 150,0.8,1.0 and 8.2 add up to 160m. To ensure that speed and positions of the cutter and paper match during the cut process the parameters of the **MOVELINK** command must be correct: It is normally easiest to consider the acceleration, constant speed and deceleration phases separately then combine them as required:

Rule 1: In an acceleration phase to a matching speed the link distance should be twice the movement distance. The acceleration phase could therefore be specified alone as:

```
MOVELINK(0.4,0.8,0.8,0,1)' move is all accel
```

Rule 2: In a constant speed phase with matching speed the two axes travel the same distance so distance to move should equal the link distance. The constant speed phase could therefore be specified as:

```
MOVELINK(0.2,0.2,0,0,1)' all constant speed
```

The deceleration phase is set in this case to match the acceleration:

```
MOVELINK(0.4,0.8,0,0.8,1)' all decel
```

The movements of each phase could now be added to give the total movement.

```
MOVELINK(1,1.8,0.8,0.8,1)' Same as 3 moves above
```

But in the example above the acceleration phase is kept separate:

```
MOVELINK(0.4,0.8,0.8,0,1)
MOVELINK(0.6,1.0,0,0.8,1)
```

This allows the output to be switched on at the end of the acceleration phase.

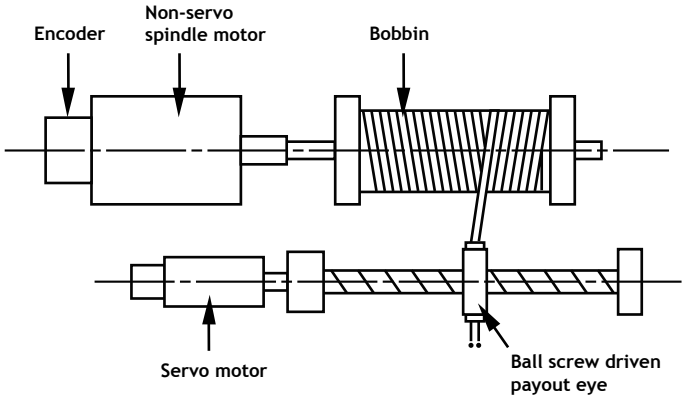
Example 3: Exact Ratio Gearbox

MOVELINK can be used to create an exact ratio gearbox between two axes. Suppose it is required to create gearbox link of 4000/3072. This ratio is inexact (1.30208333) and if entered into a **CONNECT** command the axes will slowly creep out of synchronisation. Setting the “link option” to 4 allows a continuously repeating **MOVELINK** to eliminate this problem:

```
MOVELINK(4000,3072,0,0,linkaxis,4)
```

Example 4: Coil Winding

In this example the unit conversion factors **UNITS** are set so that the payout movements are in mm and the spindle position is measured in revolutions. The payout eye therefore moves 50mm over 25 revolutions of the spindle with the command **MOVELINK(50,25)**. If it were desired to accelerate up over the first spindle revolution and decelerate over the final 3 the command would be **MOVELINK(50,25,1,3)**. **MOVELINK** and **CAMBOX** can be programmed to commence automatically relative to an absolute position on the link axis.




```
' Trio BASIC Coil Winding Example Program:
OP(motor,ON)' - Switch spindle motor on
FOR turn=1 TO 10
  MOVELINK(50,25,0,0,1)
  MOVELINK(-50,25,0,0,1)
NEXT turn
WAIT IDLE
OP(motor,OFF)
```

MOVEMODIFY

Type: Axis Command.

Syntax: **MOVEMODIFY(absolute position)**

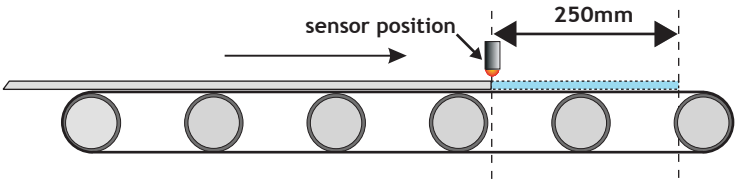
Alternate Format: `MM()`

Description: This move type changes the absolute end position of the current single axis linear move (`MOVE, MOVEABS`). If there is no current move or the current move is not a linear move then `MOVEMODIFY` is loaded as a `MOVEABS`.

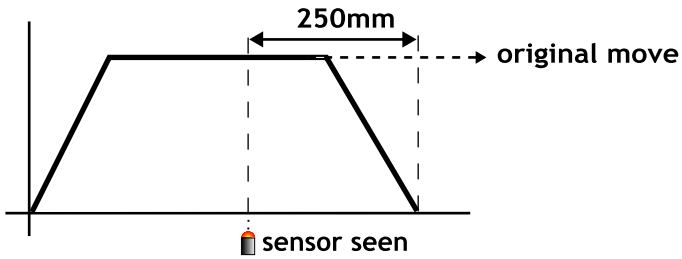
See also: `ENDMOVE`

Parameters: `absolute position:` The absolute position to be set as the new end of move.

Example: A sheet of glass is fed on a conveyor and is required to be stopped 250mm after the leading edge is sensed by a proximity switch. The proximity switch is connected to the registration input:




```
MOVE(10000)' Start a long move on conveyor
REGIST(3)'   set up registration
WAIT UNTIL MARK 'MARK will be true when proximity seen
OFFPOS=-REG_POS'set position where mark seen to 0
MOVEMODIFY(250)'change move to stop at 250mm
```



REGIST

Type: Axis Command

Syntax: `REGIST(mode,{distance})`

Description: The regist command captures an axis position when it sees the registration input or the Z mark on the encoder. The capture is carried out by hardware so software delays do not affect the accuracy of the position capture. The capture is initiated by executing the **REGIST()** command. If the input or Z mark is seen as specified by the mode within the specified window the **MARK** parameter is set **TRUE** and the position is stored in **REG_POS**. On the MC206 built-in axes 2 registration registers are provided for each axis. This allows 2 registration sources to be captured simultaneously and their difference in position determined. To use this dual registration mode the REGIST commands "mode" parameter is set in the range 6..9. Two additional axis parameters REG_POSB and MARKB hold the results of the Z mark registration in this mode.

Parameters:

mode: Determines the position to capture:

1 - Absolute position when Z Mark Rising

2 - Absolute position when Z Mark Falling

3 - Absolute position when Registration Input Rising

4 - Absolute position when Registration Input Falling

5 - Sets pattern recognition mode

6 - R Input Rising into REG_POS and Z Mark Rising into REG_POSB.

7 - R Input Rising into REG_POS and Z Mark Falling into REG_POSB.

8 - R Input Falling into REG_POS and Z Mark Rising into REG_POSB.

9 - R Input Falling into REG_POS and Z Mark Falling into REG_POSB

(mode = 6 to 9 are only available on MC206 built-in axes)

distance: The distance parameter is used for the pattern recognition mode ONLY, and specifies the distance over which to record transitions

Note: Windowing Functions

Add **256** to the above mode values to apply inclusive windowing function:

When the windowing function is applied signals will be ignored if the axis measured position is not in the range:

Greater than **OPEN_WIN** and Less than **CLOSE_WIN**

Add **768** to the above values to apply exclusive windowing function:

When the windowing function is applied signals will be ignored if the axis measured position is not in the range:

Less than **OPEN_WIN** or Greater than **CLOSE_WIN**

Note: The **REGIST** command must be re-issued for each position capture.

Example:
```
REGIST(3+256)
WAIT UNTIL MARK
PRINT "Registration Input Seen at:";REG_POS
```

RAPIDSTOP

Type: Motion Command

Alternate Format: **RS**

Description: Rapid Stop. The **RAPIDSTOP** command cancels the currently executing move on all axes. Velocity profiled move types **(MOVE, MOVEABS, MOVEMODIFY, FORWARD, REVERSE, MOVECIRC,MHELICAL)** will be ramped down. Others will be immediately cancelled. The next-move buffers and the process buffers are NOT cleared.

REVERSE

Type: Axis Command

Alternate Format: **RE**

Description: Sets continuous reverse movement on the specified or base axis.

Parameters: None.

Note: The reverse motion can only be stopped by issuing a **CANCEL** or by hitting the reverse, inhibit or datum limits.

Example:
```
back:
  REVERSE
  'Wait for stop signal:
  WAIT UNTIL IN(0)=ON
  CANCEL
```

Input / Output Commands

AIN

Type: Function

Syntax: **AIN(analog chan)**

Description Up to 4 analog input modules (P325) may be connected on the *Motion Coordinator*'s built in CAN bus port. Each P325 has 8 channels of +/-10v analog inputs which return 2047..-2048. The values from each P325 channel are updated every 10msec. On controllers with a built-in analog channel such as the MC206 channel 0 from the CAN bus is replaced by the built-in channel 0.

Parameters: **analog chan:** analog input channel number 0.31

Example: The speed of a production line is to be governed by the rate at which material is fed onto it. The material feed is via a lazy loop arrangement which is fitted with an ultra-sonic height sensing device. The output of the ultra-sonic sensor is in the range 0V to 4V where the output is at 4V when the loop is at its longest.

```
MOVE(-5000)
REPEAT
  a=AIN(1)
  IF a<0 THEN a=0
  SPEED=a*0.25
UNTIL MTYPE=0
```

Note: Note that the analog input value is checked to ensure it is above zero even though it always should be positive. This is to allow for any noise on the incoming signal which could make the value negative and cause an error because a negative speed is not valid for any move type except **FORWARD** or **REVERSE**.

AIN0..7 / AINBI0..7

Type: System Parameter

Description: These system parameters duplicate the AIN() command.

They provide the value of the analog input channels in system parameter format to allow the **SCOPE** function (Which can only store parameters) to read the analog inputs.

The value returned is a decimal representation of the voltage input and is in the range 0 to 4095 corresponding to voltage inputs in the range 0V to 4.096V. With the alternative forms AINBI0..AINBI3 The value returned is a decimal representation of the voltage input and is in the range -2048 to 2047 corresponding to voltage inputs in the range -2.048V to 2.047V.

CURSOR

Type: Command

Description: The **CURSOR** command is used in a print statement to position the cursor on the Trio membrane keypad and mini-membrane keypad. **CURSOR(0)**, **CURSOR(20)**, **CURSOR(40)**,**CURSOR(60)** are the start of the 4 lines of the 4 line display. **CURSOR(0) and CURSOR(20)** are the start of the 2 line display.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	45	47	48	49	50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79

4 Line Display as featured on the Membrane Keypad

Example:
```
PRINT#3,CURSOR(60);">Bottom line";
```

Example2: The following code

```
PRINT #kpd,CHR(12);CHR(14);CHR(20);
PRINT #kpd,CURSOR(00);"<=|General Setup1|=>";
PRINT #kpd,CURSOR(20);"Cut Length : ";VR(50+cut_length)[0];
```

would produce the following display:

<	=	|	G	e	n	e	r	a	l		S	e	t	u	p	1	|	=	>
C	u	t		L	e	n	g	t	h	:	1	3	0						
																E	X	I	T

CHR

Type: Command

Description: The **CHR(x)** command is used to send individual ASCII characters which are referred to by number. **PRINT CHR(x);** is equivalent to PUT(x) in some other version of BASIC.

Example:
```
>>PRINT CHR(65);
A
```

DEFKEY

Type: Command

Description: Under most circumstances this command is not required and it is recommended that the values of keys are input using a **GET#4** sequence. A **GET#4** sequence does not use the **DEFKEY** table. In this example a number representing which key has been pressed is put in the variable k:

```
GET#4,k
```

The **DEFKEY** command can be used to re-define what numbers are to be put in the variable when a key is pressed on a MEMBRANE keypad or Mini-Membrane keypad interfaced using an FO-VFKB module. To use the **DEFKEY** table the values are read using **GET#3**:

GET#3,k
The key numbers of the membrane keypad are shown in chapter 5 of this manual. To each of these key numbers is assigned a value by the **DEFKEY** command that is returned by a **GET#3** command.

0	1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20	21
22	23	24	25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40	41	42	43
44	45	46	47	48	49	50	51	52	53	54

Parameters:

key no:	start key number
keyvalue1:	value returned by start key through a **GET** or **GET#3** command.
keyvalue2.. keyvalue11:	values returned by successive keys through a **GET** or **GET#3** command.

Example: The command **DEFKEY (33,13)** would therefore be used to generate 13 when the first key on row 3 of a pad was pressed. Note **DEFKEY** can only be used to redefine input on channel#3.

FLAG

Type: Command/Function

Syntax: **FLAG(flag no [,value])**

Description: The **FLAG** command is used to set and read a bank of 32 flag bits. The **FLAG** command can be used with one or two parameters. With one parameter specified the status of the given flag bit is returned. With two parameters specified the given flag is set to the value of the second parameter. The **FLAG** command is provided to aid compatibility with earlier controllers and is not recommended for new programs.

Parameters: **flag no:** The flag number is a value from 0..31.

value: If specified this is the state to set the given flag to i.e. ON or OFF. This can also be written as 1 or 0.

Example 1:
```
FLAG(27,ON)' Set flag bit 27 ON
```

FLAGS

Type: Command/Function

Syntax: **FLAGS([value])**

Description: Read/Set the FLAGS as a block. The **FLAGS** command is provided to aid compatibility with earlier controllers and is not recommended for new programs. The 32 flag bits can be read with **FLAGS** and set with **FLAGS(value)**.

Parameters: **value:** The decimal equivalent of the bit pattern to set the flags to

Example: Set Flags 1,4 and 7 ON, all others OFF

Bit #	7	6	5	4	3	2	1	0
Value	128	64	32	16	8	4	2	1

```
FLAGS(146)' 2 + 16 + 128
```

Example 2: Test if FLAG 3 is set.

```
IF (FLAGS and 8) <>0 then GOSUB somewhere
```

GET

Type: Command.

Description: Waits for the arrival of a single character on the default serial port 0. The ASCII value of the character is assigned to the variable specified. The user program will wait until a character is available.

Example:
```
GET  k
```

GET#

Type: Command

Description: Functions as **GET** but the input device is specified as part of the command. The device specified is valid only for the duration of the command.

Parameters

n:
- 0 Serial port 0
- 1 Serial port 1
- 2 Serial port 2
- 3 Fibre optic port (value returned defined by **DEFKEY**)
- 4 Fibre optic port (returns raw keycode of key pressed)
- 5 *Motion* Perfect user channel
- 6 *Motion* Perfect user channel
- 7 *Motion* Perfect user channel
- 8 Used for *Motion* Perfect internal operations
- 9 Used for *Motion* Perfect internal operations
- 10+ Fibre optic network data

x: Variable

Example:
```
GET#3,k 'Just for this command input taken from fibre optic
```

Note: Channels 5 to 9 are logical channels which are superimposed on to Serial Port A by *Motion* Perfect.

Example 2: Get a key in a user menu routine

```
g_edit:
    PRINT #kpd,CHR(12);CHR(14);CHR(20);
    PRINT #kpd,CURSOR(00);"<=|General Setup1|=>";
    PRINT #kpd,CURSOR(20);"Cut Length : ";VR(clength)
    GET #kpd,option
    IF option=lastmenu OR option=f1 THEN RETURN
    IF option=menu_l2 THEN GOSUB set_cut_length
GOTO g_edit
```

HEX

Type: Command

Description: The **HEX** command is used in a print statement to output a number in hexadecimal format. The HEX command is available on MC206/MC224 only.

Example:
```
PRINT#5,HEX(IN(8,16))
```

IN()/IN

Type: Function.

Syntax: **IN(input no<,final input>)/IN**

Description: Returns the value of digital inputs. If called with no parameters, IN returns the binary sum of the first 24 inputs (if connected). If called with one parameter whose value is less than the highest input channel, it returns the value (1 or 0) of that particular input channel. If called with 2 parameters **IN()** returns in binary sum of the group of inputs. In the 2 parameter case the inputs should be less than 24 apart.

Parameters:

input no:	input to return the value of/start of input group
<final input>:	last input of group

Example 1: In this example a single input is tested:

```
test:
  WAIT UNTIL IN(4)=ON
  GOSUB place
```

Example 2: Move to the distance set on a thumb wheel multiplied by a factor. The thumb wheel is connected to inputs 4,5,6,7 and gives output in BCD.

```
moveloop:
  MOVEABS(IN(4,7)*1.5467)
  WAIT IDLE
GOTO moveloop
```

Note how the move command is constructed:

Step 1: IN(4,7) will get a number 0..15
Step 2: multiply by 1.5467 to get required distance
Step 3: absolute MOVE by this distance

Note: **IN** is equivalent to **IN(0,23)**

Example: Test if either input 2 or 3 is ON.

```
If (IN and 12) <> 0 THEN GOTO start
' (Bit 2 = 4 + Bit 3 = 8) so mask = 12
```

INPUT

Type: Command.

Description: Waits for a string to be received on the current input device, terminated with a carriage return <CR>. If the string is valid its numeric value is assigned to the specified variable. If an invalid string is entered it is ignored, an error message displayed and input repeated. Multiple inputs may be requested on one line, separated by commas, or on multiple lines, separated by <CR>.

Example1:
```
INPUT num
PRINT "BATCH COUNT=";num[0]
```
On terminal:

```
123 <CR>
BATCH COUNT=123
```

Example2:
```
getlen:
  PRINT ENTER LENGTH AND WIDTH ?";
  INPUT VR(11),VR(12)
```

This will display on terminal:
```
ENTER LENGTH AND WIDTH ? 1200,1500 <CR>
```

Note: This command will not work with the serial input device set to 3 or 4, i.e. the fibre optic port, as the received codes are not ASCII 0..9. It is also not possible for a program to use the serial port 0 as the command line process will remove the characters. Programs needing a “terminal” style interface should use one of the channel 6 to channel 7 ports if using *Motion* Perfect.

INPUTS0 / INPUTS1

Type: System Parameter

Description: The **INPUTS0** parameter holds 24 volt Input channels 0..15 as a system parameter. **INPUTS1** parameter holds 24 volt Input channels 16..31 as a system parameter. Reading the inputs using these system parameters is not normally required. The IN(x,y) command should be used instead. They are made available in this format to make the input channels available accessible to the **SCOPE** command which can only store parameters.

INVERT_IN

Type: Command.

Syntax: **INVERT_IN(input,on/off)**

Description: The INVERT_IN command allows the input channels 0..31 to be individually inverted in software. This is important as these input channels can be assigned to activate functions such as feedhold. The INVERT_IN function sets the inversion for one channel ON or OFF. It can only be applied to inputs 0..31.

Example1:
```
>>? IN(3)
0.0000
>>INVERT_IN(3,ON)
>>? IN(3)
1.0000
>>
```

KEY

Type: Function.

Description: Returns **TRUE** or **FALSE** depending on whether a character has been received on an input device or not. This command does not read the character but allows the program to test if any character has arrived. A true result will be reset when the character is read with **GET**.

The **KEY** command checks the channel specified by **INDEVICE** or by a # channel number.

Input device:

Chan	Input device:-
0	Serial port 0
1	Serial port 1
2	Serial Port 2
3	Fibre optic port (value returned defined by DEFKEY)
4	Fibre optic port (returns raw keycode of key pressed)
5	*Motion* Perfect user channel
6	*Motion* Perfect user channel
7	*Motion* Perfect user channel
8	Used for *Motion* Perfect internal operations
9	Used for *Motion* Perfect internal operations
10	Fibre optic network data

Example 1:
```
main:
  IF KEY#1 THEN GOSUB read
...
read:
  GET#1  k
RETURN
```

Example 2: To test for a character received from the fibre optic network:

```
  IF KEY#4 THEN GET#4,ch
```

LINPUT

Type: Command

Syntax: **LINPUT variable**

Description: Waits for an input string and stores the ASCII values of the string in an array of variables starting at a specified numbered variable. The string must be terminated with a carriage return <CR> which is also stored. The string is not echoed by the controller.

Parameters: None.

Example:
```
LINPUT VR(0)
```

Now entering: **START<CR>** will give:

VR(0)	83	ASCII'S'
VR(1)	84	ASCII 'T'
VR(2)	65	ASCII 'A'
VR(3)	82	ASCII 'R'
VR(4)	84	ASCII 'T'
VR(5)	13	ASCII carriage return

OP

Type: Command/Function.

Syntax: **OP[([output no,] value)]**

Description: Sets output(s) and allows the state of the first 24 outputs to be read back. The command has three different forms depending on the number of parameters. A single output channel may be set with the 2 parameter command. The first parameter is the channel number 8-95 and the second is the value to be set 0 or 1.
If the command is used with 1 parameter the parameter is used to simultaneously set the first 24 outputs with the binary pattern of the number. If the command is used with no parameters the first 24 outputs are read back. This allows multiple outputs to be set without corrupting others which are not to be changed.

Note: The first 8 outputs (0 to 7) do not physically exist on the *Motion Coordinator* so if they are written to nothing will happen and if they are read back they will always return 0.

Parameters:

output no:	Output number to set.
value:	Output value to be set. 0/1 for 2 parameter command, decimal equivalent of binary number to set on outputs for one parameter command

Example 1:
```
OP(44,1)
```
This is equivalent to OP(44,ON)

Example 2:
```
OP (34*256)
```

This sets the bit pattern 10010 on the first 5 physical outputs, outputs 13-31 would be cleared. Note how the bit pattern is shifted 8 bits by multiplying by 256 to set the first available outputs as 0 to7 do not exist.

Example 3:
```
read_op:
  VR(0)=OP
  'SET OUTPUTS 8..15 ON SIMULTANEOUSLY
  VR(0)=VR(0) AND 65280
  OP(VR(0))
```
Note how this example can also be written:
```
  'SET OUTPUTS 8..15 ON SIMULTANEOUSLY
  OP(OP AND 65280)
```

PRINT

Type: Command.

Description: The **PRINT** command allows the Trio BASIC program to output a series of characters to either the serial ports or to the fibre optic port (if fitted). The **PRINT** command can output parameters, fixed ascii strings, and single ascii characters. Multiple items to be printed can be put on the same **PRINT** line provided they are separated by a comma or semi-colon. The comma and semi- colon are used to control the format of strings to be output.

Example 1:
```
PRINT "CAPITALS and lower case CAN BE PRINTED"
```
Suppose VR(1)=6 and variab=1.5: print output will be:

Example 2:
```
>>PRINT 123.45,VR(1)
123.4500 1.5000
>>
```
Note how the comma separator forces the next item to be printed into the next tab column. The width of the field in which a number is printed can be set with the use of [w,x] after the number to be printed. Where w=width of column and x=number of decimal places.

Example 3:
```
PRINT VR(1)[4,1];variab[6,2]
6.0  1.50
```

Note that the numbers are right justified in the field with any unused leading characters being filled with spaces. If the number is too big then the field will be filled with asterisks to signify that there was not sufficient space to display the number. The maximum field width allowable is 127.

Example 4:
```
length:
PRINT "DISTANCE=";mpos
DISTANCE=123.0000
```
Note how in this example the semi-colon separator is used. This does not tab into the next column, allowing the programmer more freedom in where the print items are put. The **PRINT** command prints variables with 4 digits after the decimal point. The number of decimal places printed can be set by use of [x] after the item to be printed. Where x is the number of decimal places from 1..4

```
params:PRINT "DISTANCE=";mpos[0];" SPEED=";v[2];
DISTANCE=123 SPEED=12.34
```

Example 5:
```
15 PRINT "ITEM ";total" OF ";limit;CHR(13);
```
The **CHR(x)** command is used to send individual ASCII characters which are referred to by number. The semi-colon on the end of the print line suppresses the carriage return normally sent at the end of a print line. ASCII (13) generates CR without a line feed so the line above would be printed on top of itself if it were the only print statement in a program.

PRINT CHR(x); is equivalent to PUT(x) in some other versions of BASIC.

Note: The **PRINT** statements are normally transmitted to serial port 0. They can be redirected to other output ports by using **PRINT#**.

PRINT#

Type: Command

Description: This performs the same function as **PRINT** but the serial output device is specified as part of the command. The device is selected for the duration of the **PRINT#** command only. When execution is complete the output device reverts back to that specified by the common parameter **OUTDEVICE**.

Parameters:

n:	Output device:-
0	Serial port 0
1	Serial port 1

n:	Output device:-
2	Serial port 2
3	Fibre optic port
4	Fibre optic port duplicate
5	RS-232 port A - channel 5
6	RS-232 port A - channel 6
7	RS-232 port A - channel 7
8	RS-232 port A - channel 8 - reserved for use by *Motion* Perfect
9	RS-232 port A - channel 9 - reserved for use by *Motion* Perfect
10..24	send text string to fibre optic network node 1..15

Example: `PRINT#10,"SPEED=";SPEED[6,1];`

PSWITCH

Type: Command

Syntax: `PSWITCH(sw,en,[,axis,opno,opst,setpos,rspos])`

Description: The **PSWITCH** command allows an output to be fired when a predefined position is reached, and to go OFF when a second position is reached. There are 16 (8 on MC202) position switches each of which can be assigned to any axis, and can be assigned ON/OFF positions and OUTPUT numbers.

Multiple PSWITCH's can be assigned to a single output. The result on the output will be the OR of the position switches and the standard BASIC OP setting.

The command must be used with all 7 parameters to enable a switch, just the first 2 parameters are required to disable a switch.

Parameters:

sw: The switch number in the range 0..15

en: Switch enable -

1 or ON to enable software **PSWITCH**
0 or OFF to disable **PSWITCH**
3 to enable hardware **PSWITCH**
(hardware PSWITCH can only be used with a P242 daughter board

axis: Axis number which is to provide the position input in the range 0..number of axes on the controller. For a hardware **PSWITCH** it should be set to the axis slot number.

opno: Selects the physical output to set, should be in range 8..31. For a hardware **PSWITCH** it should be set to 0..3.

opst: Selects the state to set the output to, if 1 then output set **ON** else set it **OFF**

setpos: The position at which output is set, in user units

rspos: The position at which output is reset, in user units

Example: A rotating shaft has a cam operated switch which has to be changed for different size work pieces. There is also a proximity switch on the shaft to indicate TDC of the machine. With a mechanical cam the change from job to job is time consuming but this can be eased by using the **PSWITCH** as a software 'cam switch'. The proximity switch is wired to input 7 and the output is fired by output 11. The shaft is controlled by axis 0 of a 3 axis system. The motor has a 900ppr encoder. The output must be on from 80° after TDC for a period of 120°. It can be assumed that the machine starts from TDC.

The **PSWITCH** command uses the unit conversion factor to allow the positions to be set in convenient units. So first the unit conversion factor must be calculated and set. Each pulse on an encoder gives four edges which the controller counts, therefore there are 3600 edges/rev or 10 edges/°. If we set the unit conversion factor to 10 we can then work in degrees.

Next we have to determine a value for all the **PSWITCH** parameters.

sw The switch number can be any one we chose that is not in use so for the purpose of this example we will use number 0.

en The switch must be enabled to work, therefore this must be set to 1.

axis We are told that the shaft is controlled by axis 0, thus axis is set to 0.

opno We are told that output 11 is the one to fire, so set opno to 11.

opst When the output is set it should be on so set to 1.

setpos The output is to fire at 80° after TDC hence the set position is 80 as we are working in degrees.

rspos The output is to be on for a period of 120° after 80° therefore it goes off at 200°. So the reset position is 200.

This can all be put together to form the two lines of Trio BASIC code that set up the position switch:

```
switch:
  UNITS AXIS(0)=10'   Set unit conversion factor (°)
  REPDIST=360
  REP_OPTION=ON
  PSWITCH(0,ON,0,11,ON,80,200)
```

This program uses the repeat distance set to 360 degrees and the repeat option ON so that the axis position will be maintained in the range 0..360 degrees.

Note: After switching the **PSWITCH** off, the output may remain ON if the state was ON when the **PSWITCH** was switched off. The **OP()** command can be used to force an output OFF:

```
PSWITCH(2,OFF)'Switch OFF pswitch controlling OP 14
OP(14,OFF)
```

READPACKET

Type: Command

Syntax: **READPACKET(port#,vr#,vr count, format)**

Description: **READPACKET** is used to transmit numbers from an external computer into the global variables of the *Motion Coordinator* over a serial communications port. The data is transmitted from the PC in binary format with a CRC checksum. A detailed description of the READPACKET format can be downloaded from WWW.TRIOMOTION.COM

Parameters:

Port Number	This value should be 0 or 1
VR Number	This value tells the *Motion Coordinator* where to start setting the variables in the **VR()** global memory array.
VR count.	The number of variables to download
Format	The number format for the numbers being downloaded

RECORD

Type: Command

Syntax: **RECORD(count, table address)**

Description: The **RECORD** command is part of the pattern recognition system built into the *Motion Coordinator*. Following the recording of a sequence of transitions the **RECORD** command is used to:

1 - Reduce the number of transitions to a number defined by the programmer
2 - Store the transition pattern for subsequent comparison with **MATCH**

Parameters:

count	Number of transitions to record. The actual transitions seen may be greater than this number but the shortest ones are removed so that only the programmed transition count is stored.
table address	This value tells the *Motion Coordinator* where to store the pattern information in the global **TABLE** memory. Table used is address to address+24.

Note: See the **MATCH** command for an example of a complete recognition sequence.

SEND

Type: Command

Syntax: **SEND(n,type,data1[,data2])**

Description: Outputs a fibre-optic network message of a specified type to a given node.

Parameters:

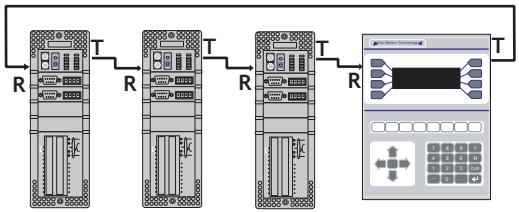
n:	Number from 10 to 24 defining the destination node.
type:	Message type: 1 - Direct variable transfer 2 - Keypad offset
data1:	If message type 1, data1 is the numbered variable number, 0..250, on the destination *Motion Coordinator* to modify. If message type 2; data1 is the number of nodes from the keypad that the key characters are to be sent. In the range 10..24, where 10 is the next node and 24 is the fifteenth node away from the keypad.
data2:	Only used if message is type 1. In this case it contains the value to change the specified variable to.

Example 1: Two *Motion Coordinator*s are fibre-optic networked together. One is acting under instruction from the other. Instructions are given by setting variable 100 to different values and the receiving *Motion Coordinator* jumping to a subroutine determined by the variable value. The program on the controlling *Motion Coordinator* would have the following send routine:

```
SEND(10,1,100,value)' Set vr(100) on dest. to value
```

Example 2: Any network containing membrane keypad(s) must initialise the keypads first to tell them where to send their output and to set them into network mode. To do this a keypad offset message is sent to the membrane keypad. Consider a network with four nodes. Three *Motion Coordinator*s and one membrane keypad connected as follows:

MCa ---> MCb---> MCc ---> Keypad ---> (back to MCa)



	MCa	MCb	MCc	Keypad
Offset from MCa	0	10	11	12
Offset from Keypad	10	11	12	0

If MCa is to initialise the keypad (offset of 2 from MCa) but MCc is to receive the keypad output (Offset of 0,1,2 from Keypad to MCc).

```
SEND(10+2,2,10+2)
```

SETCOM

Type: Command

Syntax: `SETCOM(baudrate,databits,stopbits,parity,port,mode)`

Description: Permits the serial communications parameters to be set by the user.

By default the controller set the RS232-C port to 9600 baud, 7 data bits, 2 stop bits and even parity.

Parameters:

baudrate:	1200, 2400,4800, 9600,19200 or 38400
databits:	7or 8
stopbits:	1or 2
parity:	0=none, 1=odd, 2=even
port number:	0,1 or 2
mode:	This switch is available on serial ports #1 and #2 ONLY. 0=XON/XOFF inactive,1=XON/XOFF active,4=MODBUS protocol
Note:	On power up the controllers always set the serial ports to 9600,7,2,even, XON/XOFF active.

Example 1:

```
' Set port 1 to 19200 baud, 7 data bits, 2 stop bits
' even parity and XON/XOFF enabled

SETCOM(9600,7,2,2,1,1)
```

Example 2: The Modbus protocol is initialised by setting the mode parameter of the **SETCOM** instruction to 4. The **ADDRESS** parameter must also be set *before* the Modbus protocol is activated.

```
' set up RS485 port at 19200 baud, 8 data, 1 stop, even parity
' and enable the MODBUS comms protocol

ADDRESS=1
SETCOM(19200,8,1,2,2,4)
```

Program Loops and Structures

BASICERROR

Type: Program Structure

Description: This command may only be used as part of an `ON... GOSUB` or `ON... GOTO` command. When used in this context it defines a routine to be run when an error occurs in a Trio BASIC command.

Example:

```
ON BASICERROR GOTO error_routine
....(rest of program)

error_routine:
  PRINT "The error ";RUN_ERROR[0];
  PRINT " occurred in line ";ERROR_LINE[0]
STOP
```

ELSE

Type: Program Structure

Description: This command is used as part of a multi-line `IF` statement.

See Also `IF, THEN, ENDIF`

ELSEIF

Type: Program Structure

Syntax:

```
IF  <condition1> THEN
  commands
ELSEIF <condition2> THEN
  commands
ELSE
  commands
ENDIF
```

Description: The command is used within an **IF .. THEN .. ENDIF**. It evaluates a second (or subsequent) condition and if TRUE it executes the commands specified, otherwise the commands are skipped. MC206 and MC224 only.

Parameters:

condition(s): Any logical expression.

commands: Any valid Trio BASIC commands including further **IF..THEN ..{ELSEIF}..{ELSE} ENDIF** sequences

Example 1:

```
IF IN(stop)=ON THEN
  OP(8,ON)
  VR(cycle_flag)=0
ELSEIF IN(start_cycle)=ON THEN
  VR(cycle_flag)=1
ELSEIF IN(step1)=ON THEN
  VR(cycle_flag)=99
ENDIF
```

Example 2:

```
IF key_char=$31 THEN
  GOSUB char_1
ELSEIF key_char=$32 THEN
  GOSUB char_2
ELSEIF key_char=$33 THEN
  GOSUB char_3
ELSE
  PRINT "Character unknown"
ENDIF
```

Note: The **ELSE** sequence is optional. If it is not required, the **ENDIF** is used to mark the end of the conditional block.

See Also **IF, THEN, ELSE, ENDIF**

ENDIF

Type: Program Structure

Description: The **ENDIF** command marks the end of a multi-line **IF** statement.

Example:
```
IF count >= batchsize THEN
  PRINT #3,CURSOR(20);"   BATCH COMPLETE   ";
  GOSUB index ' Index conveyor to clear batch
  count=0
ENDIF
```

See Also **IF, THEN, ELSE**

FOR..TO.. STEP..NEXT

Type: Program Structure

Syntax:
```
FOR variable=start TO end [STEP increment]
  ...
  block of commands
  ...
NEXT variable
```

Description: On entering this loop the variable is initialized to the value of start and the block of commands is then executed.

Upon reaching the **NEXT** command the variable defined is incremented by the specified **STEP**. The **STEP** parameter is optional. If not defined then it is assumed to be 1. The **STEP** value may be positive or negative.

If the value of the variable is less than or equal to the end parameter then the block of commands is repeatedly executed until this is so.

Once the variable is greater than the end value the program drops out of the **FOR..NEXT** loop.

Parameters:

variable:	A valid Trio BASIC variable. Either a global VR variable, or a local variable may be used.
start:	A valid Trio BASIC expression.

end: A valid Trio BASIC expression.

increment: A valid Trio BASIC expression. (Optional)

Example 1:
```
FOR opnum=10 TO 18
    OP(opnum,ON)
NEXT opnum
```
This loop sets outputs 10 to 18 ON.

Example 2:
```
loop:
  FOR dist=5 TO -5 STEP -0.25
    MOVEABS(dist)
    GOSUB pick_up
  NEXT dist
```

Example 3: **FOR.. NEXT** statements may be nested (up to 8 deep) provided the inner **FOR** and **NEXT** commands are both within the outer **FOR..NEXT** loop:

```
FOR x=1 TO 8
  FOR y=1 TO 6
    MOVEABS(x*100,y*100)
    WAIT IDLE
    GOSUB operation
  NEXT l2
NEXT l1
```

Note: **FOR..NEXT** loops can be nested up to 8 deep in each program.

GOSUB

Type: Program Structure

Syntax: **GOSUB label**

Description: Stores the position of the line after the **GOSUB** command and then branches to the line specified. Upon reaching the **RETURN** statement, control is returned to the stored line.

Parameters: `label:` A valid label that occurs in the program. If the label does not exist an error message will be displayed during structure checking at the beginning of program run time and the program execution halted.

Example:
```
main:
  GOSUB routine1
  GOSUB routine2
GOTO main

routine1:
  PRINT "Measured Position=";MPOS;CHR(13);
RETURN

routine2:
  PRINT "Demand Position=";DPOS;CHR(13);
RETURN
```

Note: Subroutines on each process can be nested up to 8 deep.

GOTO

Type: Program Structure

Syntax: `GOTO label`

Description: Identifies the next line of the program to be executed.

Parameters: `label:` A valid label that occurs in the program. If the label does not exist an error message will be displayed during structure checking at the beginning of program run time and the program execution halted.

Example:
```
loop:
  PRINT "Measured Position=";MPOS;CHR(13);
  WA(1000)
  GOTO loop
```

Note: Labels may be character strings of any length. Only the first 15 characters are significant. Alternatively line numbers may be used as labels.

IDLE

Type: Command Modifier

Description: Only used in conjunction with the **WAIT** command, **WAIT IDLE** suspends program execution until the base axis has finished executing its current move and any further buffered move.

Note: This does not necessarily imply that the axis is stationary in a servo motor system.

Example:
```
MOVE(100)
WAIT IDLE
PRINT "Move Complete"
```

IF..THEN..ELSE.. ENDIF

Type: Program Structure

Syntax:
```
IF  <condition> THEN
  commands
ELSE
  commands
ENDIF
```

Description: The command evaluates the condition and if it is true it executes the commands specified, otherwise the commands are skipped. If the condition is false and an **ELSE** command sequence is specified then this command sequence is executed.

Parameters:

condition: Any logical expression.

commands: Any valid Trio BASIC commands including further **IF..THEN {ELSE} ENDIF** sequences

Note: **IF.THEN {ELSE} ENDIF** sequences can be nested without limit other than program memory size

Example 1:
```
IF MPOS>(0.22*VR(0)) THEN GOTO ex_length
```

Example 2:
```
IF IN(0)=ON THEN
  count=count+1
  PRINT "COUNTS=";VR(1)
  fail=0
ELSE
  fail=fail+1
ENDIF
```

Note: For a multi-line **IF..THEN** construction there must not be any statement after the **THEN** keyword. If there is the controller will assume it is a single line IF construction. The single line construction should not use **ENDIF**.

The **ELSE** sequence is optional. If it is not required the **ENDIF** is used to mark the end of the conditional block.

NEXT

Type: Program Structure

Description: Used to mark the end of a **FOR..NEXT** loop. See **FOR**.

ON.. GOSUB

Type: Program Structure

Syntax: **ON expression GOSUB label[,label[,...]]**

Description: The expression is evaluated and then the integer part is used to select a label from the list. If the expression has the value 1 then the first label is used, 2 then the second label is used, and so on. If the value of the expression is less than 1 or greater than the number of labels then an error occurs. Once the label is selected a **GOSUB** is performed.

Example:
```
REPEAT
  GET #3,char
UNTIL 1<=char AND char<=3
ON char GOSUB mover,stopper,change
```

ON.. GOTO

Type: Program Structure

Syntax: `ON expression GOSUB label[,label[,...]]`

Description: The expression is evaluated and then the integer part is used to select a label from the list. If the expression has the value 1 then the first label is used, 2 then the second label is used, and so on. If the value of the expression is less than 1 or greater than the number of labels then an error occurs. Once the label is selected a `GOTO` is performed.

Example:

```
REPEAT
  GET #3,char
UNTIL 1<=char and char<=3
ON char GOTO mover,stopper,change
```

REPEAT.. UNTIL

Type: Program Structure

Syntax: `REPEAT commands UNTIL condition`

Description: The `REPEAT..UNTIL` construct allows a block of commands to be continuously repeated until a condition becomes `TRUE`. `REPEAT..UNTIL` loops can be nested without limit.

Example: A conveyor is to index 100mm at a speed of 1000mm/s wait for 0.5s and then repeat the cycle until an external counter signals to stop by setting input 4 on.

```
cycle:
SPEED=1000
REPEAT
MOVE(100)
WAIT IDLE
WA(500)
UNTIL IN(4)=ON
```

RETURN

Type: Program Structure

Description: Instructs the program to return from a subroutine. Execution continues at the line following the **GOSUB** instruction.

Note: Subroutines on each process can be nested up to 8 deep.

Example:
```
' calculate in subroutine:
  GOSUB calc
  PRINT "Returned from subroutine"
  STOP

calc:
  x=y+z/2
RETURN
```

STEP

Type: Program Structure

Description: This optional parameter specifies a step size in a **FOR..NEXT** sequence. See **FOR**.

Example:
```
FOR x=10 TO 100 STEP 10
    MOVEABS(x) AXIS(9)
NEXT x
```

STOP

Type: Command

Description: Stops one program at the current line. A particular program may be specified or the selected program will be assumed.

Example 1:
```
>>STOP progname
```

Example 2:
```
'DO NOT EXECUTE SUBROUTINE AT label
STOP
label: PRINT var
RETURN
```

THEN

Type: Program Structure

Description: Forms part of an **IF** expression. See IF for further information.

Example:
```
IF MARK THEN
     offset=REG_POS
ELSE
     offset=0
ENDIF
```

Note: Comments should not be placed after a **THEN** statement

TO

Type: Program Structure

Description: Precedes the end value of a **FOR..NEXT** loop.

Example:
```
FOR x=10 TO 0 STEP -1
```

UNTIL

Type: Program Structure

Description: Defines the end of a **REPEAT..UNTIL** multi-line loop, or part of a **WAIT UNTIL** structure. After the **UNTIL** statement is a condition which decides if program flow continues on the next line or at the **REPEAT** statement. **REPEAT..UNTIL** loops can be nested without limit.

Example:
```
' This loop loads a CAMBOX move each time Input 0 comes on.
' It continues until Input 6 is switched OFF.
```

```
REPEAT
  WAIT UNTIL IN(0)=OFF
  WAIT UNTIL IN(0)=ON
  CAMBOX(0,150,1,10000,1)
UNTIL IN(6)=OFF
```

WA

Type: Command

Syntax: `WA(delay time)`

Description: Holds up program execution for the number of milliseconds specified in the parameter.

Parameters: `time:` The number of milliseconds to wait for.

Example:

```
OP(11,OFF)
WA(2000)
OP(17,ON)This turns output 17 off 2 seconds after switching
output 11 off.
```

WAIT IDLE

Type: Command

Description: Suspends program execution until the base axis has finished executing its current move and any further buffered move.

Note: This does not necessarily imply that the axis is stationary in a servo motor system.

Example:

```
MOVE(100)
WAIT IDLE
PRINT "Move Done"
```

WAIT LOADED

Type: Command

Description: Suspends program execution until the base axis has no moves buffered ahead other than the currently executing move

Note: This is useful for activating events at the beginning of a move, or at the end of a move when multiple moves are buffered together. WAIT LOADED is equivilent to WAIT UNTIL (PMOVE=0) AND (NTYPE=0)

Example: Switch output 45 ON at start of **MOVE(350)** and **OFF** at the end

```
MOVE(100)
MOVE(350)
WAIT LOADED
OP(45,ON)
MOVE(200)
WAIT LOADED
OP(45,OFF)
```

WAIT UNTIL

Type: Command

Syntax: **WAIT UNTIL** condition

Description: Repeatedly evaluates the condition until it is true then program execution continues.

Parameters: **condition:** A valid Trio BASIC logic expression.

Example 1:
```
WAIT UNTIL MPOS AXIS(0)>150
MOVE(100) AXIS(7)
```

In this example the program waits until the measured position on axis 0 exceeds 150 then starts a movement on axis 7.

Example 2: The expressions evaluated can be as complex as you like provided they follow the Trio BASIC syntax, for example:

```
WAIT UNTIL DPOS AXIS(2)<=0 OR IN(1)=ON
```

This waits until demand position of axis 2 is less than or equal to 0 or input 1 is on.

WHILE

Type: Program Structure

Syntax: **WHILE condition**

Description: The commands contained in the **WHILE..WEND** loop are continuously executed until the condition becomes **FALSE**.

Execution then continues after the **WEND**.

Parameters: **condition:** Any valid logical Trio BASIC expression

Example:

```
WHILE IN(12)=OFF
  MOVE(200)
  WAIT IDLE
  OP(10,OFF)
  MOVE(-200)
  WAIT IDLE
  OP(10,ON)
WEND
```

WEND

Type: Program Structure

Description: Marks the end of a **WHILE..WEND** loop.

See also: **WHILE**

Note: **WHILE..WEND** loop can be nested without limit other than program size.

System Parameters and Commands

ADDRESS

Type: System Parameter

Syntax: **ADDRESS=value**

Description: Sets the RS485 multi-drop address for the board. This parameter should be in the range of 1..32

Example: **ADDRESS=5**

APPENDPROG

Type: System Command (This function is used by the *Motion* Perfect editor)

Syntax: **APPENDPROG <string>**

Alternate Format: **@ <string>**

Description: This command appends a line to the currently selected program.

Parameters:

string: The text, enclosed in quotation marks, that is to be appended to the program

AUTORUN

Type: System Command

Description: Starts running all the programs that have been set to run at power up.

See Also: **RUNTYPE.**

AXISVALUES

Type: System Command

Syntax: **AXISVALUES(axis,bank)**

Description: Used by *Motion* Perfect to read axis parameters. Reads banks of axis parameters. There are 2 banks of parameters for each axis, bank 0 displays the data that is only changed by the Trio BASIC, bank 1 displays the data that is changed by the motion generator.

Parameters **The data is given in the format:**

<Parameter><type>=<value>

<Parameter> is the name of the parameter

<type> is the type of the value.

- **i** integer
- **f** float
- **c** float that when changed means that the bank 0 data must be updated
- **s** string
- **c** string of upper and lower case letters, where upper case letters mean an error

<value> an integer, a float or a string depending on the type

CAN

Type: System Function

Syntax: **CAN(channel,function#,{parameters})**

Description: This function allows the CAN communication channels to be controlled from the Trio BASIC programming system. All *Motion Coordinator*'s have a single built-in CAN channel which is normally used for digital and analog I/O using Trio's I/O modules. With up to 4 CAN daughter boards plus the built-in CAN channel the units can control a maximum of 5 CAN channels:

Channel:	Channel Number:	Maximum Baudrate:
Built-in CAN	-1	500 KHz
Daughter Slot 0	0	1 Mhz
Daughter Slot 1	1	1 Mhz
Daughter Slot 2	2	1 Mhz
Daughter Slot 3	3	1 Mhz

In addition to using the **CAN** command to control CAN channels, Trio is introducing specific protocol functions into the system software. These functions are dedicated software modules which interface to particular devices. The built-in CAN channel will automatically scan for Trio I/O modules if the system parameter CANIO_ADDRESS is set to its default value of 32.

The *Motion Coordinator* CAN hardware uses the Siemens 81C91 CAN interface chip. This chip can be programmed at a register level using the **CAN** command if necessary. To program in this way it is necessary to obtain a copy of the chip data sheet.

The **CAN** command provides access to 10 separate functions:

```
CAN(channel#,function#,...)
```

Channel#
The channel number is in the range -1 to 3 and specifies the hardware channel

Function #:
There are 10 CAN functions 0..9:

0	Read 81C91 Register:	`val=CAN(channel#,0,register#)`
1	Write 81C91 Register:	`CAN(channel#,1,register#,value#)`
2	Initialise Baudrate:	`CAN(channel#,2,baudrate)`
3	Check if msg received	`val=CAN(channel#,3,message#)`
4	Set transmit request	`CAN(channel#,4,message#)`
5	Initialise message	`CAN(channel#,5,message#,identifier, length)`
6	Read message	`CAN(channel#,6,message#,variable#)`
7	Write message	`CAN(channel#,7,message#,byte0,byte1..)`

8	Read CanOpen Object	**CAN(channel#,8,transbuf,recbuf,object, subindex,variable#)**
9	Write CanOpen Object	**CAN(channel#,9,transbuf,recbuf,format, object,subindex,value,{valuems})**

Notes: **register#** is the 81C91 register number.

Baudrate: 0=1Mhz, 1=500kHz, 2=250kHz etc.

The 81C91 has 16 message buffers(0..15). The **message#** is which message buffer is required to be used.

"**Identifier**" is the CAN identifier.

variable# is the number of the global variable to start loading the data into. The function will load a sequence of n+1 variables. The first variable holds the identifier. The subsequent values hold the data bytes from the can packet.

For more information about the CanOpen read and write object, see chapter xx of the Technical Reference Manual. Functions 8 and 9 are only available on the MC206 and MC224.

CANIO_ADDRESS

Type: System Parameter (Stored in FLASH Eprom)

Description: The **CANIO_ADDRESS** holds the address used to identify the *Motion Coordinator* when using the Trio CAN I/O networking. The value is held in flash eprom in the controller and for most systems does not need to be set from the default value of 32. The value may be changed to a different value in the range 33..47 but in this case the *Motion Coordinator* will not connect to CAN-I/O modules following reset. The value of CANIO_ADDRESS should be changed from 32 if it is required to use the built-in CAN channel for functions other than controlling Trio I/O modules.

CANIO_ENABLE

Type: System Parameter

Description: The **CANIO_ENABLE** should be set OFF to completely disable use of the built-in CAN interface by the system software. This allows users to program their own protocols in Trio BASIC using the **CAN** command. The system software will set CANIO_ENABLE to **ON** on power up if the **CANIO_ADDRESS** is set to 32 and any P315 CAN I/O or P325 CAN analog modules have been detected, otherwise it will be set to **OFF**.

CANIO_STATUS

Type: System Parameter

Description: A bitwise system parameter:

Bit 0 set indicates an error from the I/O module 0,3,6 or 9

Bit 1 set indicates an error from the I/O module 1,4,7 or 10

Bit 2 set indicates an error from the I/O module 2,5,8 or 11

Bit 3 set indicates an error from the I/O module 12,13,14 or 15

Bit 4 should be set to re-initialise the CANIO network

Bit 5 is set when initialisation is complete

CHECKSUM

Type: System Parameter (Read Only)

Description: The checksum parameter holds the checksum for the programs in battery backed RAM. On power up the checksum is recalculated and compared with the previously held value. If the checksum is incorrect the programs will not run.

CLEAR

Type: System Command

Description Sets all global (numbered) variables to 0 and sets local variables on the process on which command is run to 0.

Note: Trio BASIC does not clear the global variables automatically following a **RUN** command. This allows the global variables, which are all battery-backed to be used to hold information between program runs. Named local variables are always cleared prior to program running. If used in a program **CLEAR** sets local variables in this program only to zero as well as setting the global variables to zero.

CLEAR does not alter the program in memory.

Example:
```
VR(0)=44:VR(10)=12.3456:VR(100)=2
PRINT VR(0),VR(10),VR(100)
CLEAR
PRINT VR(0),VR(10),VR(100)
```

On execution this would give an output such as:

```
44.0000 12.345 62.0000
0.0000 0.0000 0.0000
```

CLEAR_PARAMS

Type: System Command

Description Clears all variables and parameters stored in flash eprom to their default values. On the MC202 CLEAR_PARAMS will erase all the VR's stored using FLASHVR. CLEAR_PARAMS cannot be performed if the controller is locked.

COMMSERROR

Type: System Parameter

Description: This parameter returns all the communications errors that have occurred since the last time that it was initialised. It is a bitwise value defined as follows:

Bit	Value
0	RX Buffer overrun on Network channel
1	Re-transmit buffer overrun on Network channel
2	RX structure error on Network channel
3	TX structure error on Network channel
4	Port 0 Rx data ready
5	Port 0 Rx Overrun
6	Port 0 Parity Error
7	Port 0 Rx Frame Error
8	Port 1 Rx data ready (MC202 port 1 Parity Error)
9	Port 1 Rx Overrun (MC202 port 1 Rx Frame Error)
10	Port 1 Parity Error (MC202 port 1 Rx Overrun)
11	Port 1 Rx Frame Error (MC202 - no function)
12	Port 2 Rx data ready
13	Port 2 Rx Overrun
14	Port 2 Parity Error
15	Port 2 Rx Frame Error
16	Error FO Network port
17	Error FO Network port
18	Error FO Network port
19	Error FO Network port

COMMSTYPE

Type: Slot Parameter

Syntax: **COMMSTYPE SLOT(slot#)**

Description: This parameter returns the type of communications daughter board in a controller slot. On the MC206, a communications daughter board will respond with its type if the COMMSTYPE is requested from slot(0). A table of types is provided in Appendix 1.

COMPILE

Type: System Command

Description: Forces compilation (to intermediate code) of the currently selected program. Program compilation is performed automatically by the system software prior to program **RUN** or when another program is **SELECT**ed. This command is not therefore normally required.

CONTROL

Type: System Parameter (Read Only)

Description: The Control parameter returns the type of *Motion Coordinator* in the system:

Controller	**CONTROL**
MC202:	202
Euro205	205
Euro205X	255
MC206	206
PCI208	208
MC216:	216
MC224:	224

Note: When a *Motion Coordinator* is LOCKED, 1000 is added to the above number.

COPY

Type: System Command

Description: Makes a copy of an existing program in memory under a new name

Example:
```
>>COPY "prog"  "newprog"
```

Note: *Motion* Perfect users should use the “Copy program...” function under the “Program” menu.

DATE

Type: System Parameter (MC216/MC224 Only)

Description: Returns/Sets the current date held by the MC216's real time clock. The number may be entered in DD:MM:YY or DD:MM:YYYY format.

Example 1:
```
>>DATE=20:10:98
or
>>DATE=20:10:2001
```

Example2:
```
>>PRINT DATE
36956
```

This prints the number representing the current day. This number is the number of days since 1st January 1900, with 1 Jan. 1900 as 1. Trio has issued a year 2000 compliance statement which describes the year 2000 issue in relation to all Trio products.

DATE$

Type: Command (MC216/MC224 Only)

Description: Prints the current date DD/MM/YY as a string to the port. A 2 digit year description is given.

Example:
```
PRINT #3,DATE$
```

This will print the date in format for example: 20/10/01

DAY

Type: System Parameter (MC216/MC224 only)

Description: Returns the current day as a number 0..6, Sunday is 0. The **DAY** can be set by assignment.

Example:
```
>>DAY=3
>>? DAY
3.0000
>>
```

DAY$

Type: System Command (MC216/MC224 only)

Description: Prints the current day as a string.

Example:
```
>>? DAY$
3.0000
>>
```

DEL

Type: System Command

Alternate Format: **RM**

Syntax: **DEL progname**

Description: Allows the user to delete a program from memory. The command may be used without a program name to delete a currently selected program.

Motion Perfect users should use “Delete program...” on Program menu.

Example:
```
>>DEL  "oldprog"
```

DEVICENET

Type: System Command

Syntax: **DEVICENET(slot,func,baud,mac id,poll base,poll inlen,poll outlen)**

Description: The command DEVICENET is used to start and stop the DeviceNet slave function which is built into the *Motion Coordinator*. Once the DEVICENET command is running the process can be used to execute other BASIC commands in the usual way. This command is available on MC206 and MC224 only.

Parameters:

slot:	Specifies the communications slot where the CAN daughter baord is placed. Set -1 for built-in CAN port and 0 for a CAN daughter board in the MC206.
func:	0 = Start the DeviceNet slave protocol on the given slot. 1 = Stop the DeviceNet protocol.
baud:	Set to 125, 250 or 500 to specify the baudrate in kHz.
mac id:	The ID which the *Motion Coordinator* will use to identify itself on the DeviceNet network. Range 0..63.
poll base:	The first TABLE location to be transfered as poll data
poll in len:	Number of words to be received during poll
poll out len:	Number of words to be sent during poll

Example 1:
```
' Start the DeviceNet protocol on the built-in CAN port
DEVICENET(-1,0,500,30,0,4,16)
```

Example 2:
```
' Stop the DeviceNet protocol on the CAN board in slot 2
DEVICENET(2,1)
```

DIR

Type: System Command

Alternate Format: **LS**

Description: Prints a list of all programs in memory, their size and their **RUNTYPE**. The alternative format **DIR F** may be used to list the programs stored in the FlashStick if present.

Note: This command should only be used on the *Motion Coordinator* Command Line

DISPLAY

Type: System Parameter

Description: Determines the I/O channels to be displayed on the front panel LEDS.

Certain controllers, such as the Euro205 and MC206 do not have LEDs for every I/O channel. The **DISPLAY** parameter may be used to select which bank of I/O should be displayed.

The parameter default value is 0.

Parameters:

0	Inputs 0-7
1	Inputs 8-15
2	Inputs 16-23
3	Inputs 24-31
4	Outputs 0-7 (unused on existing controllers)
5	Outputs 8-15
6	Outputs 16-23
7	Outputs 24-31
8	DeviceNet Status (MC206 only)

Example:
```
DISPLAY=5
' Show outputs 8-15
```

DLINK

Type: System Command

Syntax: **DLINK(function,…)**

Description: This is a specialised command, to allow access to the SLM™ digital drive interface. During the power sequence, when a SLM™ interface card is found, all the ASICs are initialised, starting the communications protocol.

The axis parameters have to be initialised by the **DLINK** function 2 command before the interface can be used for controlling an external drive.

Parameters:

Function:	Specifies the required function. 0 = Read a register on the SLM™ ASIC 1 = Write a register on the SLM™ ASIC 2 = Check for presence SLM module 3 = Check for presence of SLM servo drive 4 = Assign a *Motion Coordinator* axis to a SLM channel 5 = Read an SLM parameter 6 = Write an SLM parameter 7 = Write an SLM command 8 = Read a drive parameter 9 = Returns slot and asic number associated with an axis 10 = Read an EEPROM parameter

Read a register on the SLM™ ASIC.

Parameters:

Function	0
Slot	The communications slot in which the interface daughter board is inserted.
ASIC	The number of the ASIC to be used. Each SLM™ daughter board has 3 ASICs. The master ASIC is 0, the first slave is 1 and the second slave is 2.
Register	The number of the register to be read.

Example:

```
>>PRINT DLINK(0,0,0,3)
117.0000
>>
```

Write a register on the SLM™ ASIC.

Parameters:

Function	1
Slot	The communications slot in which the interface daughter board is inserted.
ASIC	The number of the ASIC to be used.
Register	The number of the register to be written to.
Value	The value to be written.

Example:

```
>>DLINK(1,0,0,1,244)
>>
```

Check for presence SLM module on rear of motor. Returns 1 if the SLM is answering, otherwise it returns 0.

Parameters:

Function	2
Slot	The communications slot in which the interface daughter board is inserted.
ASIC	The number of the ASIC to be used.

```
>>? DLINK(2,0,0)
1.0000
>>
```

Check for presence of SLM servo drive, such as MultiAx. Returns 1 if the drive is answering, otherwise it returns 0. **The current SLM software dictates that the drive MUST be powered up after power is applied to the** ***Motion Coordinator*** **/ SLM.**

Parameters:

Function	3
Slot	The communications slot in which the interface daughter board is inserted.
ASIC	The number of the ASIC to be used.

Example:

```
>>? DLINK(3,0,0)
0.0000
>>
```

Assign a *Motion Coordinator* axis to a SLM channel.

Parameters:

Function	4
Slot	The communications slot in which the interface daughter board is inserted.
ASIC	The number of the ASIC to be used.
Axis	The axis to be associated with this drive. If this axis is already assigned then it will fail. The ATYPE of this axis will be set to 11.

Example:

```
>>DLINK(4,0,0,0)
```

Read an SLM parameter

Parameters:

Function	5
Axis	The axis to be associated with this drive. If this axis is out of range, or is not of the correct type (see function 2) then the function will fail.
Parameter	The number of the SLM parameter to be read. This is normally in the range 0…127. See the drive documentation for further information.

Example:

```
>>PRINT DLINK(5,0,1)
463.0000
>>
```

Write an SLM parameter

Parameters:

Function	6
Axis	The axis to be associated with this drive.
Parameter	The number of the SLM parameter to be read. See Function 4
Value	The value to be set.

Example:

```
>>DLINK(6,0,0,0)
>>
```

Write an SLM command. If command is successful this function returns a TRUE, otherwise it returns FALSE

Parameters:

Function	7
Axis	The axis to be associated with this drive.
Command	The command number. (see drive documentation)

Example:

```
>>PRINT DLINK(7,0,250)
1.0000
>>
```

Read a drive parameter

Parameters:

Function	8
Axis	The axis to be associated with this drive.
Parameter	The number of the drive parameter to be read. This must be in the range 0…127. See the servo drive documentation for further information.

Example:
```
>>PRINT DLINK(8,0,53248)
20504.0000
>>
```

Return slot and asic number associated with an axis

Parameters:

Function	9
Axis	Axis number.
Returns	10 x slot number + ASIC number.

Example:
```
>>PRINT DLINK(9,2)
>>11.0000
```
This example is for slot 1, asic 1

Read an EEPROM parameter

Parameters:

Function	10
Axis	The axis to be associated with this drive/SLM.
Parameter	EEPROM parameter number. (see drive documentation)

Example:
```
>>PRINT DLINK(10,0,29)
>>62128.0000
```
Returns EEPROM parameter 29, the Flux Angle

EDPROG

Type: System Command

Alternate Format: &

Description: This is a special command that may be used to manipulate the programs on the controller. It is not normally used except by *Motion* Perfect.

It has several forms:

&C	Print the name of the currently selected program
&<line>D	Delete line <line> from the currently selected program
&<line>I,<string>	Insert the text <string> in the currently selected program at the line <line>. Note - you should NOT enclose the string in quotes unless they need to be inserted into the program.
&K	Print the checksum of the system software
&<start>,<end>L	Print the lines of the currently selected program between <start> and <end>
&N	Print the number of lines in the currently selected program
&<line>R,<string>	Replace the line <line> in the currently selected program with the text <string>. Note - you should NOT enclose the string in quotes unless they need to be inserted into the program.
&Z,<progname>	Print the CRC checksum of the specified program. This uses the standard CCITT 16 bit generator polynomial

EDIT

Type: System Command

Syntax: **EDIT [optional line sequence number]**

Description: The edit command starts the built in screen editor allowing a program in the controller memory to be modified using a VT100 terminal. The SELECTed program is edited.

The line sequence number may be used to specify where to start editing.

Quit Editor	-Control K then D
Delete line	-Control Y
Cursor Control	-Cursor Keys

EPROM

Type: System Command

Description: Stores the Trio BASIC programs in the controller in the FLASH EPROM. This information is be retrieved on power up if the **POWER_UP** parameter has been set to 1. The **EPROM(n)** functions are only usable on an MC206 and MC224..

EPROM or **EPROM(0)**	Stores application programs in ram into on board flash.
EPROM(1)	Stores application programs in ram into FlashStick.
EPROM(2)	Stores application programs in ram into the FlashStick and marks the EPROM request flag so that the programs are copied from the FlashStick into on board flash when the stick is inserted into a controller which is unlocked.
EPROM(3)	Deletes all programs in the FlashStick, leaves data sectors intact.

Note: This command should only be used on the command line. *Motion* Perfect performs the **EPROM** command automatically when the *Motion Coordinator* is set to "Fixed"

See Also: **STICK_WRITE, STICK_READ, DIR**

When using the Memory Stick users should refer to the overview in the MC206 Hardware Overview for a complete description of the Memory Stick functionality.

ERROR_AXIS

Type: System Parameter (Read Only)

Description: Returns the number of the axis which caused the enable **WDOG** relay to open when a following error exceeded its limit.

Example:
```
>>? ERROR_AXIS
```

ETHERNET

Type: System Command

Syntax:
```
ETHERNET(read/write, slot number, function [,data])
```

Description: The command ETHERNET is used to read and set certain functions of the Ethernet daughter board. The ETHERNET command should be entered on the command line with *Motion* Perfect in "disconnnected" mode via the serial port 0. This command is available on MC206, EURO205X and MC224 only.

Parameters:

read / write: Specifies the required action.
0 = Read
1 = Write to Flash EPROM
2 = Write to RAM

slot number: The daughter board slot where the Ethernet port has been installed. On the MC206 this is always slot 0.

function: Function number must be one of the following values.

0 = IP Address
1 = Static(1) or dynamic(0) addressing. (Only static addressing is supported.)
2 = Subnet Mask
3 = MAC address
4 = Default Port Number (initialised to 23)
5 = Token Port Number (initialised to 3240)
6 = Ethernet daughter board firmware version (read only)
7 = Modbus TCP mode. Integer (0) or Floating point (1). (daughter board firmware V1.0.3.1 or later)

data: The optional data is used when changing a parameter value.

When writing to the EPROM on the Ethernet daughter board, the new value will only be used after power has been cycled to the controller. Any data written to RAM is used straight away.

Example 1: Set the IP address for the Ethernet daughter board in slot 0.

```
ETHERNET(1,0,0,192,200,185,2)
```

Example 2: Read the firmware version number in the Ethernet daughter board in slot 2.

```
ETHERNET(0,2,6)
```

EX

Type: System Command

Description: Software reset. Resets the controller as if it were being powered up again.

Note: On **EX** the following actions occur:

- The global numbered (**VR**) variables remain in memory.
- The base axis array is reset to 0,1,2... on all processes
- Axis following errors are cleared
- Watchdog is set OFF
- Programs may be run depending on **POWER_UP** and **RUNTYPE** settings
- ALL axis parameters are reset.

EX may be included in a program. This can be useful following a run time error. Care must be taken to ensure it is safe to restart the program.

Note2: When running *Motion* Perfect executing an **EX** command will prevent communication between the controller and the PC. The same effect as an **EX** can be obtained by using “Reset the controller...” under the “Controller” menu in *Motion* Perfect.

EXECUTE

Type: System Command

Description: Used to implement the remote command execution via the USB interface. For more details see the section on using the OCX control.

FEATURE_ENABLE

Type: System Function

Syntax: **FEATURE_ENABLE(feature number)**

Description: The EURO205,EURO205X, PCI208 and MC206 *Motion Coordinator*s, have the ability to unlock additional axes by entering a “Feature Enable Code”. This function is used to enable such protected features of a controller. It is recommended to use *Motion* Perfect 2 to enter and store the feature enable codes.

Note: To add servo axes to a EURO205 stepper base card the DAC chip must be added to the Euro205 in addition to enabling the feature.

Feature:	Function:
0	Stepper generation hardware on axis 0
1	Stepper generation hardware on axis 1
2	Stepper generation hardware on axis 2
3	Stepper generation hardware on axis 3
4	Servo generation hardware on axis 0
5	Servo generation hardware on axis 1
6	Servo generation hardware on axis 2
7	Servo generation hardware on axis 3

Controllers with features which can be enabled are fitted with a unique security code number when manufactured. This security code number can be found by typing **FEATURE_ENABLE** with no parameters:

Example 1:
```
>>feature_enable
Security code=17980000000028
Enabled features: 0 1
```
If you require additional features for a controller. These can be enabled by the entry of a password which is unique for each feature and controller security code. To obtain a feature enable code, the feature must be ordered and the security code FAXed to Trio or a distributor.

Example 2: In example one axes 0 and 1 are enabled for stepper operation. If axis 2 was required to operate as a stepper axis it would be necessary to obtain the password. For this card and this feature only the password is 5P0APT.

```
>>feature_enable(2)
Feature 2 Password=5P0APT
>>
>>feature_enable
Security code=17980000000028
Enabled features: 0 1 2
```

Note: When entering the passwords always enter the characters in upper case. Take care to check that 0 (zero) is not confused with O and 1 (one) is not confused with I.

FLASHVR

Type: System Function

Syntax: **FLASHVR([variable number])**

Description: Stores a single **VR()** global variable into permanent flash memory. VR variables stored in this way will have their value restored to the current value when the unit is powered up again. This feature is provided on controllers which do not feature battery backed ram **VR()** storage. Each **FLASHVR** command generates a write to a block of flash eprom. After 8000 block writes the flash sector will be erased. The controller writes into a second sector during the erase. Each sector can be erased over 1,000,000 times. It is therefore possible to use the FLASHVR() command many hundreds of millions of times. It does however have a finite life and cannot easily be replaced. Programmers MUST allow for this fact.

Note: The FLASHVR function is provided on controllers without batteries such as the MC202. However the FLASHVR(-1) and FLASHVR(-2) functions can be used with all *Motion Coordinator*'s. These functions write a whole block of data to flash memory and the programmer must ensure that they are only used occasionally.

The **FLASHVR** command can also be used to store the TABLE memory. By using the command **FLASHVR(-1)**, the entire contents of the TABLE memory will be written to the flash memory.

To revert to the standard power-up mode, i.e. not reading the values from the eprom, you should use the command **FLASHVR(-2)**

Parameters: **variable number:** The variable to be stored into flash

Example 1:
```
VR(25)=k
FLASHVR(25)
```

Example 2:
```
FOR v=1 to 10
  FLASHVR(v)
NEXT v
```

Example 3:
```
FLASHVR(-1)' Store TABLE memory to flash eprom
```

FRAME

Type: System Parameter

Description: Used to specify which "frame" to operate within when employing frame transformations. Frame transformations are used to allow movements to be specified in a multi-axis coordinate frame of reference which do not correspond one-to-one with the axes. An example is a SCARA robot arm with jointed axes. For the end tip of the robot arm to perform straight line movements in X-Y the motors need to move in a pattern determined by the robots geometry.

Frame transformations to perform functions such as these need to be compiled from "C" language source and loaded into the controller system software. Contact Trio if you need to do this.

A machine system can be specified with several different "frames". The currently active **FRAME** is specified with the **FRAME** system parameter.

The default **FRAME** is 0 which corresponds to a one-to-one transformation.

Example:
```
FRAME=1
```

FREE

Type: System Parameter (Read Only)

Description: Returns the amount of program memory available for user programs.

Note: Each line takes a minimum of 4 characters (bytes) in memory. This is for the length of this line, the length of the previous line, number of spaces at the beginning of the line and a single command token. Additional commands need one byte per token, most other data is held as ASCII.

The Coordinator compiles programs before they are run, this means that approximately twice the memory is required to be able to run a program.

Parameters: None

Example 1:
```
>>PRINT FREE
47104.0000
>>
```

Example 2:
```
VR(10)=IN AND 255
```
This line requires 21 bytes of storage in the uncompiled version and 19 in the compiled version:

Uncompiled:

Byte	Length	Value
0	1	PREVIOUS LINE LENGTH
1	1	THIS LINE LENGTH
2	1	PRECEDING BLANKS
3	1	VR TOKEN
4	1	(TOKEN
5	1	NUMBER TOKEN
6	2	Digits 1-0
8	1	END OF NUMBER TOKEN
9	1	) TOKEN
10	1	= TOKEN
11	1	IN TOKEN
12	1	SPACE TOKEN
13	1	AND TOKEN
14	1	SPACE TOKEN
15	1	NUMBER TOKEN
16	3	Digits 2-5-5
19	1	END OF NUMBER TOKEN
20	1	END OF LINE

Compiled version:

Byte	Length	Value
0	1	VR TOKEN
1	1	(TOKEN
2	1	NUMBER TOKEN
3	4	32 bit floating number
7	1	END OF EXPRESSION TOKEN
8	1	) TOKEN
9	1	ASSIGNMENT TOKEN
10	1	IN TOKEN
11	1	NUMBER TOKEN
12	4	32 bit floating point number
16	1	END OF NUMBER TOKEN
17	1	AND TOKEN
18	1	END OF LINE

HALT

Type: System Command

Description: Halts execution of all running programs. The **STOP** command will stop a specific program.

Example:
```
HALT ' Stop ALL programs
```
or
```
STOP "main"
' Stop only the program called 'MAIN'
```

Note: **HALT** does not stop any motion. Currently executing, or buffered moves will continue unless they are terminated with a **CANCEL** or **RAPIDSTOP** command.

INITIALISE

Type: System Command.

Description: Sets all axis, system and process parameters to their default values. The parameters are also reset each time the controller is powered up, or when an **EX** (software reset) command is performed. When using *Motion* Perfect a "Reset the controller.." under the "Controller" menu performs the equivalent of an **EX** command

LAST_AXIS

Type: System Parameter (Read Only)

Description: In order to maximise the processor time available to BASIC, the *Motion Coordinator* keeps a record of the highest axis number that is in use. This axis number is held in the system parameter **LAST_AXIS**. Axes higher than **LAST_AXIS** are not processed.

LAST_AXIS is set automatically by the system software when an axis command is used.

LIST

Type: System Command

Alternate Format: TYPE

Description: Prints the current SELECTed program or a specified program to channel 0.

Note: ***LIST is used as an immediate (command line) command only and should not be used in programs. Use of LIST in Motion Perfect is not recommended.***

LOADSYSTEM

Type: System Command

Description: Loads new version of system software:

On the *Motion Coordinator* family of controllers the system software is stored in FLASH EPROM. It is copied into RAM when the system is powered up so it can execute faster. The system software can be re-loaded through the serial port 0 into RAM using *Motion* Perfect. The command **STORE** is then used to transfer the updated copy of the system software into the FLASH EPROM for use on the next power up.

To re-load the system software you will need the system software on disk supplied by TRIO in COFF format. (Files have a.OUT suffix, for example C140.OUT)

The download sequence:

Run *Motion* Perfect in the usual way. Under the “Controller” menu select “Load system software...”. Select the version of system software to be loaded and follow the on screen instructions. The system file takes around 12 minutes to download. When the download is complete the system performs a checksum prior to asking the user to confirm that the file should be loaded into flash eprom. The storing process takes around 10 seconds and must NEVER be interrupted by the power being removed. If this final stage is interrupted the controller may have to be returned to Trio for re-initialisation.

Note 1: All *Motion Coordinator* models have different system software files. The file name indicates the controller type.

Controller Type	Filename
MC202	Fnnn.OUT
Euro205	Ennn.OUT
MC206	Jnnn.OUT
MC216	Cnnn.OUT

Updates can be obtained from Trio's website at WWW.TRIOMOTION.COM

Note 2: Application programs should be stored on disk prior to a system software load and MUST be reloaded following a system software load.

LOCK

Type: System Command

Syntax: LOCK(code)

Description: LOCK is designed to prevent programs from being viewed or modified by personnel unaware of the security code. The lock code number is stored in the flash eprom.

When a *Motion Coordinator* is locked, it is not possible to view, edit or save any programs and command line instructions are limited to those required to execute the program.

To unlock the *Motion Coordinator*, the UNLOCK command should be entered using the same lock code number which was used originally to LOCK it.

The lock code number may be any integer and is held in encoded form. Once LOCKed, the only way to gain full access to the *Motion Coordinator* is to UNLOCK it with the correct code. For best security the lock number should be 7 digits.

Parameters: code Any integer number

Example: >>LOCK(5619234)
The program cannot now be modified or viewed.

>>UNLOCK(5619234)
The system is now unlocked.

Note 1: LOCK and UNLOCK are available from the *Motion Coordinator* menu in *Motion* Perfect.

Note 2 ***If you forget the security code number, the Motion Coordinator may have to be returned to your supplier to be unlocked!***

Note 3 ***It is possible to compromise the security of the lock system. Users must consider if the level of security is sufficient to protect their programs.***

MOTION_ERROR

Type: System Parameter

Description: This system parameter returns the value 1 when a motion error has occured on at least one axis, (normally a following error, but see **ERRORMASK**), and the value 0 when none of the axes has had a motion error. When there is a motion error then the **ERROR_AXIS** contains the number of the first axis to have an error. When any axis has a motion error then the watchdog relay is opened. A motion error can be cleared by resetting the controller with an EX command ("Reset the controller.." under the "Controller" menu in *Motion* Perfect), or by using the **DATUM(0)** command.

MPE

Type: System Command

Description: Sets the type of channel handshaking to be performed on the serial port 0. This is normally only used by the *Motion* Perfect program, but can be used for user applications. There are 4 valid settings

Parameters **channel type:** Any valid Trio BASIC expression

0 No channel handshaking, XON/XOFF controlled by the port. When the current output channel is changed then nothing is sent to the serial port. When there is not enough space to store any more characters in the current input channel then XOFF is sent even though there may be enough space in a different channel buffer to receive more characters

1 Channel handshaking on, XON/XOFF controlled by the port. When the current output channel is changed, the channel change sequence is sent (<ESC><channel number>). When there is not enough space to store any more characters in the current input channel then XOFF is sent even though there may be enough space in a different channel buffer to receive more characters

2 Channel handshaking on, XON/XOFF controller by the channel. When the current output channel is changed, the channel change sequence is sent (<ESC><channel number>). When there is not enough space to store any more characters in the current input buffer, then XOFF is sent for this channel (<XOFF><channel number) and characters can still be received into a different channel.

Whatever the **MPE** state, if a channel change sequence is received on serial port A then the current input channel will be changed.

3 Channel handshaking on, XON/XOFF controller by the channel. In **MPE(3)** mode the system transmits and receives using a protected packet protocol using a 16 bit CRC.

Example1:
```
>> PRINT #5,"Hello"
Hello
```

Example2:
```
MPE(1)
>> PRINT #5,"Hello"
<ESC>5Hello
<ESC>0
>>
```

NAIO

Type: System Parameter (Read Only)

Description: This parameter returns the number of CAN analog input channels connected on the IO expansion CAN bus. For example an MC216 will return 8 if there is 1 x P325 CAN Module connected as it has 8 analog input channels.

NETSTAT

Type: System Parameter

Description: This parameter stores the network error status since the parameter was last cleared by writing to it. The error types reported are:

Bit Set	Error Type	Value
0	TX Timeout	1
1	TX Buffer Error	2
2	RX CRC Error	4
3	RX Frame Error	8

NEW

Type: System Command

Description Deletes all the program lines in the controller memory. It also may be used to delete the current **TABLE** entries.

Note:

NEW	Deletes the currently selected program
NEW progname	Deletes a particular program
NEW ALL	Deletes all programs in memory
NEW "TABLE"	Delete TABLE (In this case ONLY the program name “TABLE” must be in quotes)

NIO

Type: System Parameter (Read Only)

Description: This parameter returns the number of inputs/outputs fitted to the system, or connected on the IO expansion CAN bus.

Note: Depending on the particular controller type, there may be a number of channels which are input only. For example, on the MC216 the first 8 channels are inputs, the next 8 bi-directional. If an MC216 has 2 P315 CAN-16 I/O modules connected the NIO parameter will return 48.

All channels on the CAN-16 I/O modules are bi-directional.

PEEK

Type: System Command

Syntax: **PEEK(address<,mask>)**

Description: The PEEK command returns value of a memory location of the controller ANDed with an optional mask value.

POKE

Type: System Command

Syntax: **POKE(address,value)**

Description: The POKE command allows a value to be entered into a memory location of the controller. The POKE command can prevent normal operation of the controller and should only be used if instructed by Trio Motion Technology.

POWER_UP

Type: Flash EPROM stored System Parameter

Description: This parameter is used to determine whether or not programs should be read from Flash Eprom on power up or software reset (**EX**).

Two values are possible:

0 Use the programs in battery backed RAM

1 Copy programs from the controllers Flash Eprom or Memory Stick (if present) into RAM.

The EPROM request flag held on the Memory Stick itself controls if the programs loaded are to be written into the internal Flash Eprom after loading (See **EPROM**)

Programs are individually selected to be run at power up with the **RUNTYPE** command

Note: **POWER_UP** is always an immediate command and therefore cannot be included in programs.

This value is normally set by *Motion* Perfect.

When using the Memory Stick users should refer to the overview in the MC206 Hardware Overview for a complete description of the Memory Stick functionality.

PROCESS

Type: System Command

Description: Displays the running status and process number for each current process.

PROFIBUS

Type: System Command

Syntax: **PROFIBUS(slot,function<,register><,value>)**

Description: The command PROFIBUS provides access to the registers of the SPC3 ASIC used on the Profibus daughter board. Trio can supply sample programs using this command to setup and control a Profibus daughter board.

Parameters:

slot:	Specifies the slot on the controller to be used. Set 0 for the daughter board slot of an MC206/Euro205 or the slot number of an MC216.
function:	Specifies the function to be performed. 0: read register 1: write register
register:	The SPC3 register number to read or write
value:	The value to write into an SPC3 register

REMOTE

Type: System Command

Syntax: **REMOTE(slot)**

Description: Transfers control of a process to the remote computer via a USB interface and the Trio OCX control. The REMOTE command is normally inserted automatically on to a process by the system software. When a process is performing the REMOTE function execution of BASIC statements is suspended.

RENAME

Type: System Command

Description: Renames a program in the *Motion Coordinator* directory.

Example:
```
>>RENAME car voiture
```

Note: *Motion* Perfect users should use "Rename Program..." under the "Program" menu to perform a **RENAME** command.

RUN

Type: System Command

Description: Runs a program on the controller.

Parameter: A program name, and process number may optionally be specified.

Note: Execution continues until:

- There are no more lines to execute
- or **HALT** is typed at the command line. This stops all programs
- or **STOP** "name" is typed at the command line. This stops single program

RUN may be included in a program to run another program:

```
RUN "cycle"
```

Example: **RUN** - this will run currently selected program)

Example 2: **RUN** "sausage" - this will run the named program)

Example 3: **RUN** "sausage",3 - run the named program on a particular process)

RUNTYPE

Type: System Command

Syntax: **>>RUNTYPE progname,autorun[,process#]**

Description: Sets whether program is run automatically at power up, and which process it is to run on. The current status of each program's **RUNTYPE** is displayed when a **DIR** command is performed. For any program to run automatically on power-up ALL the programs on the controller must compile without errors.

Parameters:

program name	Can be in inverted commas or without autorun
autorun	1 to run automatically, 0 for manual running
<process number>	optional to force process number

Example: **>>RUNTYPE progname,1,10**
- Sets program "progeny" to run automatically on power up on process 10

>>RUNTYPE "progname",0
- Sets program "progname" to manual running

Note: To set the **RUNTYPE** using *Motion* Perfect select the "Set Power-up mode" option in the "Program" menu.

Note 2: The **RUNTYPE** information is stored into the flash EPROM only when an **EPROM** command is performed.

See Also: **POWER_UP**

SCOPE

Type: System Command

Description: The **SCOPE** command is used to program the system to automatically store up to 4 parameters every sample period. The sample period can be any multiple of the servo period. The data stored is put in the **TABLE** data structure. It may then be read back to a PC and displayed on the *Motion* Perfect Oscilloscope or stored to a file for further analysis using the "Save TABLE file" option under the "File" menu.

Motion Perfect uses the **SCOPE** command when running the Oscilloscope function.

Parameters:

ON/OFF control	Set ON or OFF to control the SCOPE function. OFF implies that the scope data is not ready. ON implies that the scope data is loaded correctly and is ready to run when the TRIGGER command is executed.
Period	The number of servo periods between data samples
Table start	Position to start to store the data in the table array
Table stop	End of table range to use
P0	first parameter to store
P1	optional second parameter to store
P2	optional third parameter to store
P3	optional fourth parameter to store

Example 1:
```
SCOPE(ON,10,0,1000,MPOS AXIS(5), DPOS AXIS(5))
```
This example programs the **SCOPE** facility to store away the **MPOS** axis 5 and **DPOS** axis 5 every 10 milliseconds. The **MPOS** will be stored in table values 0..499, the **DPOS** in table values 500 to 999. The sampling does not start until the **TRIGGER** command is executed.

Example 2:
```
SCOPE(OFF)
```

Note: The **SCOPE** facility is a “one-shot” and needs to be re-started by the **TRIGGER** command each time an update of the samples is required.

Note2: Data saved to the TABLE memory by the **SCOPE** command is not placed in battery backed memory so will be lost when power is removed.

SCOPE_POS

Type: System Parameter (Read Only)

Description: Returns the current index position at which the **SCOPE** function is currently storing its parameters.

SELECT

Type: System Command

Description: Selects the current active program for editing, running, listing etc. **SELECT** makes a new program if the name entered is not a current program.

When a program is **SELECT**ed the commands **EDIT**, **RUN**, **LIST**, **NEW** etc. assume that the **SELECT**ed program is the one to operate with unless a program is specified as in for example: **RUN progname**

When a program is selected any previously selected program is compiled.

Note: The **SELECT**ed program cannot be changed when programs are running.

Note 2: *Motion* Perfect automatically **SELECT**s programs when you click on their entry in the list in the control panel.

SLOT

Type: Slot Modifier

Description: Modifier specifies the slot number for a slot parameter such as COMMSTYPE.

SERVO_PERIOD

Type: System Parameter

Description: This parameter allows the controller servo period to be specified. On the MC202, MC216 and Euro205 controllers this period is specified in seconds and on the MC206, MC224, PCI208, and EURO205X controller it is specified in microseconds. It is recommended not to adjust the default 1msec servo period on the MC202 and Euro205 controllers or where the stepper daughter board is being used. Where the MC216 is being used for servo axes the servo period may be adjusted down to around 0.0005 seconds.

Note: The *Motion Coordinator* must be reset using the **EX** command before the new servo period will be applied.

On the MC206, MC224, PCI208, and EURO205X the servo period may be set to 1000,500 or 250 usec.

STEPLINE

Type: System Command

Syntax: **STEPLINE {Program name}{,Process number}**

Description: Steps one line in a program. This command is used by *Motion* Perfect to control program stepping. It can also be entered directly from the command line or as a line in a program with the following parameters.

Parameters:

Program name: This specifies the program to be stepped. All copies of this named program will step unless the process number is also specified. If the program is not running it will step to the first executable line on either the specified process or the next available process if the next parameter is omitted. If the program name is not supplied, either the SELECTed program will step (if command line entry) or the program with the STEPLINE in it will stop running and begin stepping.

Process number: This optional parameter determines which process number the program will use for stepping, or, if multiple copies of the same program exist, it is used to select the required copy for stepping.

Example 1:
```
>>STEPLINE "conveyor"
```

Example 2:
```
>>STEPLINE "maths",2
```

STICK_READ

Type: System Command

Syntax: **STICK_READ(sector, table start)**

Description: Copy one block of 128 values from a sector on the FlashStick to TABLE memory. The function returns TRUE (-1) if the **STICK_READ** was successful and FALSE (0) if the command failed, if for example the FlashStick is not present.

Parameters:

sector:	A number between 0 and 2047 that is used as a pointer to the sector to be read from the FlashStick.
table start:	The start point in the TABLE where the 128 values will be transferred to.

Example:
```
IF STICK_READ(25, 1000) THEN PRINT "Stick read OK"
```

STICK_WRITE

Type: System Command

Syntax: **STICK_WRITE(sector, table start)**

Description: Copy one block of 128 values from TABLE memory to a sector on the FlashStick. The function returns TRUE (-1) if the **STICK_WRITE** was successful and FALSE (0) if the command failed, if for example the FlashStick is not present.

Parameters:

sector:	A number between 0 and 2047 that is used as a pointer to the sector to be written to the FlashStick.
table start:	The start point in the TABLE where the 128 values will be transferred from.

Example:
```
check = STICK_WRITE(25, 1000)
IF check=TRUE THEN PRINT "Stick write OK"
```

STORE

Type: System Command

Description: Stores an update to the system software into FLASH EPROM. This should only be necessary following loading an update to the system software supplied by TRIO. See also **LOADSYSTEM**.

Warning: ***Removing the controller power during a STORE sequence can lead to the controller having to be returned to Trio for re-initialization.***

Note: Use of **STORE** and **LOADSYSTEM** is automated for *Motion* Perfect users by the "Load system software..." option in the "Controller" menu.

TABLE

Type: System Command

Syntax: **TABLE(address [, data1..data20])**

Description: The **TABLE** command is used to load and read back the internal cam table. This table has a fixed maximum table length of 16000 points on all *Motion Coordinator*s EXCEPT the MC202 which has an 8000 point table length, and the MC224 which has a 256000 point table. Issuing the **TABLE** command or running it as a program line must be done before table points are used by a **CAM** or **CAMBOX** command. The table values are floating point and can therefore be fractional.

The command has two forms:

(i)With 2 or more parameters the **TABLE** command defines a sequence of values, the first value is the first table position.

(ii) If a single parameter is specified the table value at that entry is returned. As the table can be written to and read from, it may be used to hold information as an alternative to variables.

The values in the table may only be read if a value of THAT NUMBER OR GREATER has been specified. For example, if the value of table position 1000 has been specified e.g. **TABLE(1000,1234)** then **TABLE(1001)** will produce an error message. The highest **TABLE** which has been loaded can be read using the **TSIZE** parameter.

Except in the MC202 the table entries are automatically battery backed. If FLASH Eprom storage is required it is recommended to set the values inside a program or use the FLASHVR(-1) function. It is not normally required to delete the table but if this necessary the **DEL** command can be used:

```
>>DEL "TABLE"
```

Parameters:

address:	location in the table at which to store a value or to read a value from if only this parameter is specified.
data1..data20:	the value to store in the given location and at subsequent locations if more than one data parameter is used.

Example 1:

```
TABLE(100,0,120,250,370,470,530)
```

This loads the internal table:

Table Entry:	Value:
100	0
101	120
102	250
103	370
104	470
105	530

Example 2:
```
>>PRINT TABLE(1000)
0.0000
>>
```

Note: The Oscilloscope function of *Motion* Perfect uses the table as a data area. The range used can be set in the scope "Options..." screen. Care should be taken not to use a data area in use be the Oscilloscope function.

TABLEVALUES

Type: System Command

Syntax: **TABLEVALUES(first table number, last required table number, format)**

Description: Returns a list of table points starting at the number specified. There is only one format supported at the moment, and that is comma delimited text.

Parameters

address:	Number of the first point to be returned
number of points:	Total number of points to be returned
format:	Format for the list

Note: **TABLEVALUES** is provided mainly for *Motion* Perfect to allow for fast access to banks of **TABLE** values.

TIME

Type: System Parameter (MC216/MC224 only)

Description: Returns the time from the real time clock. The time returned is the number of seconds since midnight 24:00 hours.

TIME$

Type: System Command (MC216/MC224 only)

Description: Prints the current time as defined by the real time clock as a string in 24hr format.

Example:
```
>>? TIME$
  14/39/02
>>
```

TRIGGER

Type: System Command

Description: Starts a previously set up **SCOPE** command

Note: *Motion* Perfect uses **TRIGGER** automatically for its oscilloscope function.

TROFF

Type: System Command

Description: Suspends the trace facility at the current line and resumes normal program execution. A program name can be specified or the selected program will be assumed.

Example:
```
>>TROFF "lines"
```

TRON

Type: System Command

Description: The trace on command suspends a programs execution at the current line. The program can then be single stepped, executing one line at a time, using the **STEPLINE** command.

Note: Program execution may be restarted without single stepping using **TROFF**. The trace mode may be halted by issuing a **STOP** or **HALT** command. *Motion* Perfect highlights lines containing **TRON** in its editor and debugger.

Example:
```
TRON
MOVE(0,10)
MOVE(10,0)
TROFF
MOVE(0,-10)
MOVE(-10,0)
```

TSIZE

Type: System Parameter

Description: Returns one more than the highest currently defined table value.

Example:
```
>>TABLE(1000,3400)
>>PRINT TSIZE
1001.0000
```

Note: **TSIZE** can be reset using **>>DEL "TABLE"**

UNLOCK

Type: System Command

Syntax: **UNLOCK(code)**

Description: Enables full access to a *Motion Coordinator* which has a security lock code applied via the **LOCK()** command.

When a *Motion Coordinator* is locked, it is not possible to view, edit or save any programs and command line instructions may be limited to those required to execute the program only.

To unlock the *Motion Coordinator*, the **UNLOCK** command should be entered using the same security code number which was used originally to **LOCK** it.

The security code number may be any integer and is held in encoded form. Once **LOCK**ed, the only way to gain full access to the *Motion Coordinator* is to **UNLOCK** it with the correct code.

Parameters:

code	Any integer number

Example:

```
>>LOCK(561234)
```
The program cannot now be modified or seen.

```
>>UNLOCK(561234)
```
The system is now unlocked.

Note 1: It is not normally necessary to use the **LOCK**/**UNLOCK** commands from the command line as the they are available directly from the Controller menu in *Motion* Perfect 2.

USB

Type: System Command

Syntax: **USB(slot,function<,register><,value>)**

Description: The command USB provides access to the registers of the USBN9602 USB controller chip used in the MC206 and USB daughter board. It is not required to use this command as the functions are included in the *Motion Coordinator* system software.

Parameters:

slot:	Specifies the slot on the controller to be used. Set 1 for the built-in USB of the MC206 or the slot number of an MC216/ Euro205.
function:	Specifies the function to be performed. 0: read register 1: write register

register: The register number to read or write

value: The value to write into a register

USB_STALL

Type: System Parameter

Description: This parameter returns TRUE if the USB controller chip has its “stalled” (unable to communicate) bit set.

VERSION

Type: System Parameter

Description: Returns the version number of the system software installed on the *Motion Coordinator*.

Example:
```
>>? VERSION
  1.6200
```

VIEW

Type: System Command

Description: Lists the currently selected program in tokenised and internal compiled format.

Example: For the following program:

```
VR(10)=IN AND 255
```
the view command will give the following output:
Source code:

```
from xxx to xxx
10725: 00 15 00 29 92 95 31 30 00 93 88 64 A2 95 32 35 35 00 9B
10746: 15 00 00 00
Object code: from yyy to yyy
```

```
10750: 01 00 29 92 95 00 20 03 91 93 9A 64 95 00 00 7F 07 8E 91 9B
10771:
```

VR

Type: System Command

Syntax: **VR(expression)**

Description: Recall or assign to a global numbered variable. The variables hold real numbers and can be easily used as an array or as a number of arrays. There are 251 variable locations which are accessed as variables 0 to 250, except on the MC206 which has 1024 VR's numbered from 0..1023.

The numbered variables are used for several purposes in Trio BASIC. If these requirements are not necessary it is better to use a named variable:

The numbered variables are BATTERY BACKED (except on MC202) and are not cleared between power ups.- The numbered variables are globally shared between programs and can be used for communication between programs. To avoid problems where two processes write unexpectedly to a global variable, the programs should be written so that only one program writes to the global variables.

The numbered variables can be changed by remote controllers on the TRIO Fibre Optic Network, or from a master via a MODBUS or other supported network.

The numbered variables can be used for the **LINPUT**, **READPACKET** and **CAN** commands.

Example 1: ' put value 1.2555 into VR() variable 15. Note local variable 'van' used to give name to global variable:

```
val=15
VR(val)=1.2555
```

Example 2: A transfer gantry has 10 put down positions in a row. Each position may at any time be FULL or EMPTY. **VR(101)** to **VR(110)** are used to hold an array of ten1's or 0's to signal that the positions are full (1) or EMPTY (0). The gantry puts the load down in the first free position. Part of the program to achieve this would be:

```
movep:
  MOVEABS(115)  ' MOVE TO FIRST PUT DOWN POSITION:
  FOR VR(0)=101 TO 110
    IF VR(VR(0))=0) THEN GOSUB load
```

```
      MOVE(200)' 200 IS SPACING BETWEEN POSITIONS
    NEXT VR(0)
    PRINT "All Positions Are Full"
    WAIT UNTIL IN(3)=ON
GOTO movep

load:
   'PUT LOAD IN POSITION AND MARK ARRAY
   OP(15,OFF)
   VR(VR(0))=1
RETURN
```

Note: The variables are battery-backed so the program here could be designed to store the state of the machine when the power is off. It would of course be necessary to provide a means of resetting completely following manual intervention.

Example 3:
```
'Assign VR(65) to VR(0) multiplied by Axis 1 measured position
VR(65)=VR(0)*MPOS AXIS(1)
PRINT VR(65)
```

WDOG

Type: System Parameter

Description: Controls the **WDOG** relay contact used for enabling external drives. The **WDOG=ON** command MUST be issued in a program prior to executing moves. It may then be switched ON and OFF under program control. If however a following error condition exists on any axis the system software will override the watchdog contact OFF.

Example:
```
WDOG=ON
```

Note: **WDOG=ON / WDOG=OFF** is issued automatically by *Motion* Perfect when the “Drives Enable” button is clicked on the control panel

Note 2: Analog outputs and step/direction outputs are also disabled when WDOG=OFF

:

Type: Special Character

Description: The colon character is used to terminate labels used as destinations for **GOTO** and **GOSUB** commands.

Labels may be character strings of any length. (The first 15 characters are significant) Alternatively line numbers can be used. Labels must be the first item on a line and should have no leading spaces.

Example:
```
start:
```

The colon is also used to separate Trio BASIC statements on a multi-statement line. The only limit to the number of statements on a line is the maximum of 79 characters per line.

Example:
```
PRINT "THIS LINE":GET low:PRINT "DOES THREE THINGS!"
```

Note: The colon separator must not be used after a **THEN** command in a multi-line **IF..THEN** construct. If a multi-statement line contains a **GOTO** the remaining statements will not be executed:

```
PRINT "Hello":GOTO Routine:PRINT "Goodbye"
```
Goodbye will not be printed.

Similarly with GOSUB because subroutine calls return to the following line.

’

Type: Special Character

Description: A single ' is used to mark a line as being a comment only with no execution significance.

Note: The REM command of other BASICs is replaced by '.

Like REM statements ' must be at the beginning of the line or statement or after the executable statement. Comments use memory space and so should be concise in very long programs. Comments have no effect on execution speed since they are not present in the compiled code.

Example:
```
‘PROGRAM TO ROTATE WHEEL
turns=10
‘turns contains the number of turns required
MOVE(turns)’ the movement occurs here
```

#

Type: Special Character

Description: The # symbol is used to specify a communications channel to be used for serial input/output commands.

Note: Communications Channels greater than 3 will only be used when the controller is running in *Motion* Perfect mode (See **MPE** command).

Example 1:
```
PRINT #3,"Membrane Keypad"
PRINT #2,"Port 2"
```

Example 2:
```
' Check membrane keypad on fibre-optic channel
IF KEY #3 THEN GET #3,k
```

$

Type: Special Character

Description: The $ symbol is used to specify that the number that follows is in hexadecimal format.

Note: Only the MC206,EURO205X, PCI208 and MC224 controllers support hexadecimal format.

Example 1:
```
VR(10)=$8F3B
OP($CC00)
```

Process Parameters and Commands

ERROR_LINE

Type: Process Parameter (Read Only)

Description: Stores the number of the line which caused the last Trio BASIC error. This value is only valid when the **BASICERROR** is **TRUE**. This parameter is held independently for each process.

Example:
```
>>PRINT ERROR_LINE PROC(14)
```

INDEVICE

Type: Process Parameter

Description: This parameter specifies the active input device. Specifying an **INDEVICE** for a process allows the channel number for a program to set for all subsequent **GET** and **KEY**, **INPUT** and **LINPUT** statements. (This command is not usually required - Use **GET #** and **KEY #** etc. instead)

Chan	Input device:-
0	Serial port A
1	Serial port B
2	RS485 Port
3	Fibre optic port (value returned defined by DEFKEY)
4	Fibre optic port (returns raw keycode of key pressed)
5	*Motion* Perfect user channel
6	*Motion* Perfect user channel
7	*Motion* Perfect user channel
8	Used for *Motion* Perfect internal operations
9	Used for *Motion* Perfect internal operations
10	Fibre optic network data

Example:
```
INDEVICE=5
  ' Get character on channel 5:
GET k
```

LOOKUP

Type: Process Command

Syntax: **LOOKUP(format,entry) <PROC(process#)>**

Description: The LOOKUP command allows *Motion* Perfect to access the local variables on an executing process. It is not normally required for BASIC programs.

Parameters:

format:	0: Prints (in binary) floating point value from an expression 1: Prints (in binary) integer value from an expression 2: Prints (in binary) local variable from a process 3: Returns to BASIC local variable from a process
entry:	Either an expression string (format=0 or 1) or the offset number of the local variable into the processes local variable list.

OUTDEVICE

Type: Process Parameter

Description: The value in this parameter determines the serial output device for the **PRINT** command for the process. The channel numbers are the same as described in **INDEVICE**.

PMOVE

Type: Process Parameter

Modifier: **PROC**

Description: Returns 1 if the process move buffer is occupied, and 0 it is empty. When one of the *Motion Coordinator* processes encounters a movement command the process loads the movement requirements into its “process move buffer”. This can hold one movement instruction for any group of axes. When the load into the process move buffer is complete the **PMOVE** parameter is set to 1. When the next servo interrupt occurs the motion generation program will load the movement into the “next move buffer” of the required axes if these are available. When this second transfer is complete the **PMOVE** parameter is cleared to 0. Each process has its own **PMOVE** parameter.

PROC

Type: Process Modifier

Description: Allows a process parameter from a particular process to be read or set.

Example:
```
WAIT UNTIL PMOVE PROC(14)=0
```

PROC_LINE

Type: Process Parameter (Read Only)

Description: Allows the current line number of another program to be obtained with the **PROC(x)** modifier.

Example:
```
PRINT PROC_LINE PROC(2)
```

PROCNUMBER

Type: Process Parameter

Description: Returns the process on which a Trio BASIC program is running. This is normally required when multiple copies of a program are running on different processes.

Example:
```
MOVE(length) AXIS(PROCNUMBER)
```

PROC_STATUS

Type: Process Parameter (Read Only)

Description: Returns the status of another process, referenced with the **PROC(x)** modifier.

Example:
```
WAIT UNTIL PROC_STATUS PROC(12)=0
```

Returns

0	Process Stopped
1	Process Running
2	Process Stepping
3	Process Paused

RESET

Type: Process Command

Description: Sets the value of all the local named variables of a Trio BASIC process to 0.

RUN_ERROR

Type: Process Parameter

Modifier: PROC

Description: Contains the number of the last program error that occurred on the specified process.

Example:
```
>>? RUN_ERROR PROC(5)
  9.0000
>>
```

TICKS

Type: Process Parameter

Description: The current count of the process clock ticks is stored in this parameter. The process parameter is a 32 bit counter which is DECREMENTED on each servo cycle. It can therefore be used to measure cycle times, add time delays, etc. The ticks parameter can be written to and read.

Example:
```
delay:
  TICKS=3000
  OP(9,ON)
test:
  IF TICKS<=0 THEN OP(9,OFF) ELSE GOTO test
```

Note: TICKS is held independently for each process.

Mathematical Operations and Commands

+ Add

Type: Arithmetic operation

Syntax **<expression1> + <expression2>**

Description: Adds two expressions

Parameters: **Expression1:** Any valid Trio BASIC expression

Expression2: Any valid Trio BASIC expression

Example:

```
result=10+(2.1*9)
```

Trio BASIC evaluates the parentheses first giving the value 18.9 and then adds the two expressions. Therefore result holds the value 28.9

- Subtract

Type: Arithmetic operation

Syntax **<expression1> - <expression2>**

Description: Subtracts expression2 from expression1

Parameters: **Expression1:** Any valid Trio BASIC expression

Expression2: Any valid Trio BASIC expression

Example:

```
VR(0)=10-(2.1*9)
```

Trio BASIC evaluates the parentheses first giving the value 18.9 and then subtracts this from 10. Therefore VR(0) holds the value -8.9

* Multiply

Type: Arithmetic operation

Syntax **<expression1> * <expression2>**

Description: Multiplies expression1 by expression2

Parameters: **Expression1:** Any valid Trio BASIC expression

Expression2: Any valid Trio BASIC expression

Example:

```
factor=10*(2.1+9)
```

Trio BASIC evaluates the brackets first giving the value 11.1 and then multiplies this by 10. Therefore factor holds the value 111

/ Divide

Type: Arithmetic operation

Syntax **<expression1> / <expression2>**

Description: Divides expression1 by expression2

Parameters: **Expression1:** Any valid Trio BASIC expression

Expression2: Any valid Trio BASIC expression

Example:

```
a=10/(2.1+9)
```

Trio BASIC evaluates the parentheses first giving the value 11.1 and then divides 10 by this number

Therefore a holds the value 0.9009

= Equals

Type: Arithmetic Comparison Operation

Syntax **<expression1> = <expression2>**

Description: Returns **TRUE** if expression1 is equal to expression2, otherwise returns false.

Note: **TRUE** is defined as -1, and **FALSE** as 0

Parameters: **Expression1:** Any valid Trio BASIC expression

Expression2: Any valid Trio BASIC expression

Example:
```
IF IN(7)=ON THEN GOTO label
```

If input 7 is ON then program execution will continue at line starting "**label:**"

<> Not Equal

Type: Arithmetic Comparison Operation

Syntax **<expression1> <> <expression2>**

Description: Returns **TRUE** if expression1 is not equal to expression2, otherwise returns false.

Note: **TRUE** is defined as -1, and **FALSE** as 0

Parameters: **Expression1:** Any valid Trio BASIC expression

Expression2: Any valid Trio BASIC expression

Example: IF MTYPE<>0 THEN GOTO scoop
If axis is not idle (MTYPE=0 indicates axis idle) then goto label "scoop"

> Greater Than

Type: Arithmetic Comparison Operation

Syntax **<expression1> > <expression2>**

Description: Returns **TRUE** if expression1 is greater than expression2, otherwise returns false.

Note: **TRUE** is defined as -1, and **FALSE** as 0

Parameters:
Expression1: Any valid Trio BASIC expression
Expression2: Any valid Trio BASIC expression

Example 1:
```
WAIT UNTIL MPOS>200
```
The program will wait until the measured position is greater than 200

Example 2:
```
VR(0)=1>0
```
1 is greater than 0 and therefore VR(0) holds the value -1

>= Greater Than or Equal

Type: Arithmetic Comparison Operation

Syntax **<expression1> >= <expression2>**

Description: Returns **TRUE** if expression1 is greater than or equal to expression2, otherwise returns false.

Note: **TRUE** is defined as -1, and **FALSE** as 0

Parameters:
Expression1: Any valid Trio BASIC expression
Expression2: Any valid Trio BASIC expression

Example:
```
IF target>=120 THEN MOVEABS(0)
```
If variable target holds a value greater than or equal to 120 then move to the absolute position of 0.

< Less Than

Type: Arithmetic Comparison Operation

Syntax **<expression1> < <expression2>**

Description: Returns **TRUE** if expression1 is less than expression2, otherwise returns false.

Note: **TRUE** is defined as -1, and **FALSE** as 0

Parameters:

Expression1: Any valid Trio BASIC expression

Expression2: Any valid Trio BASIC expression

Example:
```
IF AIN(1)<10 THEN GOSUB rollup
```
If the value returned from analog input 1 is less than 10 then execute subroutine "**rollup**"

<= Less Than or Equal

Type: Arithmetic Comparison Operation

Syntax **<expression1> <= <expression2>**

Description: Returns **TRUE** if expression1 is less than or equal to expression2, otherwise returns false.

Note: **TRUE** is defined as -1, and **FALSE** as 0

Parameters:

Expression1: Any valid Trio BASIC expression

Expression2: Any valid Trio BASIC expression

Example:
```
maybe=1<=0
```
1 is not less than or equal to 0 and therefore variable **maybe** holds the value 0

ABS

Type: Function

Syntax: **ABS(expression)**

Description: The **ABS** function converts a negative number into its positive equal. Positive numbers are unaltered.

Parameters: **Expression:** Any valid Trio BASIC expression

Example:
```
IF ABS(AIN(0))>100 THEN
  PRINT "Analog Input Outside +/-100"
ENDIF
```

ACOS

Type: Function

Syntax: **ACOS(expression)**

Description: The **ACOS** function returns the arc-cosine of a number which should be in the range 1 to -1. The result in radians is in the range 0..PI

Parameters: **Expression:** Any valid Trio BASIC expression.

Example:
```
>>PRINT ACOS(-1)
3.1416
```

AND

Type: Logical and bitwise operator

Syntax **<expression1> AND <expression2>**

Description: This performs an AND function between corresponding bits of the integer part of two valid Trio BASIC expressions.

The AND function between two bits is defined as follows:

Parameters: **Expression1:** Any valid Trio BASIC expression

Expression2: Any valid Trio BASIC expression

Example 1:
```
IF (IN(6)=ON) AND (DPOS>100) THEN tap=ON
```

Example 2:
```
VR(0)=10 AND (2.1*9)
```
Trio BASIC evaluates the parentheses first giving the value 18.9, but as was specified earlier, only the integer part of the number is used for the operation, therefore this expression is equivalent to:

```
VR(0)=10 AND 18
```
AND is a bitwise operator and so the binary action taking place is:

	0	1
0	0	0
1	0	1

```
    01010
AND 10010
    -----
    00010
```

Therefore **VR(0)** holds the value 2

Example 3:
```
IF MPOS AXIS(0)>0 AND MPOS AXIS(1)>0 THEN GOTO cyc1
```

ASIN

Type: Mathematical Function

Syntax: **ASIN(expression)**

Alternate Format: **ASN(expression)**

Description: The **ASIN** function returns the arc-sine of a number which should be in the range +/-1. The result in radians is in the range -PI/2.. +PI/2 (Numbers outside the +/-1 input range will return zero)

Parameters: **Expression:** Any valid Trio BASIC expression.

Example:
```
>>PRINT ASIN(-1)
-1.5708
```

ATAN

Type: Mathematical Function

Syntax: **ATAN(expression)**

Alternate Format: **ATN(expression)**

Description: The **ATAN** function returns the arc-tangent of a number. The result in radians is in the range -PI/2.. +PI/2

Parameters: **Expression:** Any valid Trio BASIC expression.

Example:
```
>>PRINT ATAN(1)
0.7854
```

ATAN2

Type: Mathematical Function

Syntax: **ATAN2(expression1,expression 2)**

Description: The **ATAN2** function returns the arc-tangent of the ratio expression1/expression 2. The result in radians is in the range -PI.. +PI

Parameters: **Expressions:** Any valid Trio BASIC expression.

Example:
```
>>PRINT ATAN2(0,1)
0.0000
```

CLEAR_BIT

Type: Command

Syntax: **CLEAR_BIT(bit#,vr#)**

Description: **CLEAR_BIT** can be used to clear the value of a single bit within a **VR()** variable.

Example: **CLEAR_BIT(6,23)**
Bit 6 of VR(23) will be cleared (set to 0).

Parameters:

bit #	Bit number within the VR. Valid range is 0 to 23
vr#	VR() number to use

See also **READ_BIT, SET_BIT**

CONSTANT

Type: System Command

Syntax: **CONSTANT "name", value**

Description: Declares the *name* as a constant for use both within the program containing the **CONSTANT** definition and all other programs in the Motion Coordinator project. MC206, EURO205X, PCI208 and MC224 only.

Parameters:

name:	Any user-defined name containing lower case alpha, numerical or underscore (_) characters.
value	The value assigned to *name*.

Example:

```
CONSTANT "nak",$15
CONSTANT "start_button",5

IF IN(start_button)=ON THEN OP(led1,ON)
IF key_char=nak THEN GOSUB no_ack_received
```

Note: The program containing the **CONSTANT** definition must be run before the name is used in other programs. For fast startup the program should also be the ONLY process running at power-up.

A maximum of 128 **CONSTANTs** can be declared.

COS

Type: Mathematical Function

Syntax: **COS(expression)**

Description: Returns the **COSINE** of an expression. Will work for any value. Input values are in radians.

Parameters: **Expression:** Any valid Trio BASIC expression.

Example:
```
>>PRINT COS(0)[3]
1.000
>>
```

EXP

Type: Mathematical Function

Syntax: **EXP(expression)**

Description: Returns the exponential value of the expression.

FRAC

Type: Mathematical Function

Syntax: **FRAC(expression)**

Description: Returns the fractional part of the expression.

Example:
```
>>PRINT FRAC(1.234)
0.2340
```

GLOBAL

Type: System Command

Syntax: **GLOBAL "name", vr_number**

Description: Declares the *name* as a reference to one of the global VR variables. The name can then be used both within the program containing the **GLOBAL** definition and all other programs in the Motion Coordinator project. MC206, EURO205X, PCI208 and MC224 only.

Parameters:

name:	Any user-defined name containing lower case alpha, numerical or underscore (_) characters.
vr_number	The number of the VR to be associated with *name*.

Example:

```
GLOBAL "srew_pitch",12
GLOBAL "ratio1",534

ratio1 = 3.56
screw_pitch = 23.0
PRINT screw_pitch, ratio1
```

Note: The program containing the **GLOBAL** definition must be run before the name is used in other programs. For fast startup the program should also be the ONLY process running at power-up.

In programs that use the defined **GLOBAL**, **name** has the same meaning as **VR(vr_number)**. Do not use the syntax: VR(name).

A maximum of 128 **GLOBALs** can be declared.

IEEE_IN

Type: Mathematical Function

Syntax: **IEEE_IN(byte0,byte1,byte2,byte3)**

Description: The **IEEE_IN** function returns the floating point number represented by 4 bytes which typically have been received over a communications link such as Modbus. MC206, EURO205X, PCI208 and MC224 only.

Parameters: **byte0 - 3:** Any combination of 8 bit values that represents a valid IEEE floating point number.

Example:
```
VR(20) = IEEE_IN(b0,b1,b2,b3)
```

Note: Byte 0 is the high byte of the 32 bit floating point format.

IEEE_OUT

Type: Mathematical Function

Syntax: **byte_n = IEEE_OUT(value, n)**

Description: The **IEEE_OUT** function returns a single byte in IEEE format extracted from the floating point value for transmission over a bus sytem. The function will typically be called 4 times to extract each byte in turn. MC206, EURO205X, PCI208 and MC224 only.

Parameters: **value:** Any TrioBASIC floating point variable or parameter.

n: The byte number (0 - 3) to be extracted.

Example:
```
byte0 = IEEE_OUT(MPOS AXIS(2), 0)
byte1 = IEEE_OUT(MPOS AXIS(2), 1)
byte2 = IEEE_OUT(MPOS AXIS(2), 2)
byte3 = IEEE_OUT(MPOS AXIS(2), 3)
```

Note: Byte 0 is the high byte of the 32 bit IEEE floating point format.

INT

Type: Mathematical Function

Syntax: **INT(expression)**

Description: The **INT** function returns the integer part of a number.

Parameters: **expression:** Any valid Trio BASIC expression.

Example:
```
>>PRINT INT(1.79)
1.0000
>>
```

Note: To round a positive number to the nearest integer value take the **INT** function of the (number + 0.5)

LN

Type: Mathematical Function

Syntax: **LN(expression)**

Description: Returns the natural logarithm of the expression.

Parameter: Any valid Trio BASIC expression.

MOD

Type: Mathematical Function

Syntax: **MOD(expression)**

Description: Returns the integer modulus of an expression.

Example:
```
>>PRINT 122 MOD(13)
5.0000
>>
```

NOT

Type: Mathematical Function

Description: The **NOT** function truncates the number and inverts all the bits of the integer remaining.

Parameter: **expression:** Any valid Trio BASIC expression.

Example:
```
PRINT 7 AND NOT(1.5)
  6.0000
>>
```

OR

Type: Logical and bitwise operator

Description: This performs an OR function between corresponding bits of the integer part of two valid Trio BASIC expressions. The OR function between two bits is defined as follows:

OR	0	1
0	0	1
1	1	1

Parameters:

Expression1:	Any valid Trio BASIC expression
Expression2:	Any valid Trio BASIC expression

Example 1:
```
IF KEY OR IN(0)=ON THEN GOTO label
```

Example 2:
```
result=10 OR (2.1*9)
```

Trio BASIC evaluates the parentheses first giving the value 18.9, but as was specified earlier, only the integer part of the number is used for the operation, therefore this expression is equivalent to:

```
result=10 OR 18
```

The OR is a bitwise operator and so the binary action taking place is:

```
    01010
OR  10010
----------
    11010
```

Therefore result holds the value 26

READ_BIT

Type: Command

Syntax: **READ_BIT(bit#,vr#)**

Description: **READ_BIT** can be used to test the value of a single bit within a **VR()** variable.

Example: **res=READ_BIT(4,13)**

Parameters:

bit #	Bit number within the VR. Valid range is 0 to 23
vr#	VR() number to use

See also **SET_BIT, CLEAR_BIT**

SET_BIT

Type: Command

Syntax: **SET_BIT(bit#,vr#)**

Description: **SET_BIT** can be used to set the value of a single bit within a **VR()** variable. All other bits are unchanged.

Parameters:

bit #	Bit number within the VR. Valid range is 0 to 23
vr#	**VR()** number to use

Example:
```
SET_BIT(3,7)
Will set bit 3 of VR(7) to 1.
```

See also **READ_BIT, CLEAR_BIT**

SGN

Type: Mathematical Function

Syntax: **SGN(expression)**

Description: The **SGN** function returns the SIGN of a number.

1	Positive non-zero
0	Zero
-1	Negative

Parameters: **expression:** Any valid Trio BASIC expression.

Example:
```
>>PRINT SGN(-1.2)
-1.0000
>>
```

SIN

Type: Mathematical Function

Syntax: **SIN(expression)**

Description: Returns the SINE of an expression. This is valid for any value in expressed in radians.

Parameters: **expression:** Any valid Trio BASIC expression.

Example:
```
>>PRINT SIN(0)
  0.0000
```

SQR

Type: Mathematical Function

Syntax: **SQR(number)**

Description: Returns the square root of a number.

Parameters: **number:** Any valid Trio BASIC number or variable.

Example:
```
>>PRINT SQR(4)
2.0000
```

```
>>
```

TAN

Type: Mathematical Function

Syntax: `TAN(expression)`

Description: Returns the TANGENT of an expression. This is valid for any value expressed in radians.

Parameters: `Expression:` Any valid Trio BASIC expression.

Example:
```
>>PRINT TAN(0.5)
  0.5463
```

XOR

Type: Logical and bitwise operator

Description: This performs and XOR function between corresponding bits of the integer part of two valid Trio BASIC expressions. It may therefore be used as either a bitwise or logical condition.

The `XOR` function between two bits is defined as follows:

Parameters: `Expression1:` Any valid Trio BASIC expression

`Expression2:` Any valid Trio BASIC expression

Example:
```
a = 10 XOR (2.1*9)
```

Trio BASIC evaluates the parentheses first giving the value 18.9, but as was specified earlier, only the integer part of the number is used for the operation, therefore this expression is equivalent to: `a=10 XOR 18`. The `XOR` is a bitwise operator and so the binary action taking place is:

```
    01010
XOR 10010
----------
    11000
```

The result is therefore 24.

Constants

FALSE

Type: Constant

Description: The constant **FALSE** takes the numerical value of 0.

Example:
```
test:
res=IN(0) OR IN(2)
  IF res=FALSE THEN PRINT "Inputs are off"
ENDIF
```

OFF

Type: Constant

Description: **OFF** returns the value 0

Example:
```
IF IN(56)=OFF THEN GOTO label
'Branch if input 56 is off.
```

ON

Type: Constant

Description: **ON** returns the value 1.

Example:
```
OP(lever,ON)'  This sets the output named lever to ON.
```

TRUE

Type: Constant

Description: The constant **TRUE** takes the numerical value of -1.

Example:
```
t=IN(0)=ON AND IN(2)=ON
IF t=TRUE THEN
  PRINT "Inputs are on"
ENDIF
```

PI

Type: Constant

Description: **PI** is the circumference/diameter constant of approximately 3.14159

Example:
```
circum=100
PRINT "Radius=";circum/(2*PI)
```

Axis Parameters

ACCEL

Type: Axis Parameter

Syntax: **ACCEL=value**

Description: The **ACCEL** axis parameter may be used to set or read back the acceleration rate of each axis fitted. The acceleration rate is in units/sec/sec.

Example:
```
ACCEL=130:' Set acceleration rate
PRINT " Accel rate:";ACCEL;" mm/sec/sec"
```

ADDAX_AXIS

Type: Axis Parameter (Read Only)

Syntax: **ADDAX_AXIS**

Description: Returns the axis currently linked to with the **ADDAX** command, if none the parameter returns -1.

ATYPE

Type: Axis Parameter

Description: The **ATYPE** axis parameter may be used to test the type of each axis fitted. It will return the values:

ATYPE	Description
0	No axis daughter board fitted
1	Stepper Axis
2	Servo Axis
3	Encoder Reference Axis
4	Stepper with position verification/Differential stepper

ATYPE	Description
5	Resolver Axis
6	Voltage output
7	Absolute SSI Servo
8	CAN daughter board
9	Remote CAN axis
10	PSWITCH Axis
11	Remote DriveLink axis
12	Reserved
13	Embedded axis
14	Encoder Output
15	Reserved
16	Remote SERCOS speed axis
17	Remote SERCOS position axis
18	Remote CanOpen position axis
19	Remote CanOpen speed axis
20	Reserved
21	Remote User Specific CAN axis

The **ATYPE** axis parameter is set by the system software at power up.

Controllers such as the Euro205, MC206 and MC202 an return ATYPE of the equivalent function of the controller. On the MC202 the **ATYPE** parameter can be set to either 1 or 2 to select the axis function. On the Euro205, EURO205X, PCI208 and MC206 the ATYPE can be altered by entering Feature Enable Codes.

Example:
```
>>PRINT ATYPE AXIS(2)
1.0000
```
This would show that an stepper daughter board is fitted in this axis slot.

AXISSTATUS

Type: Axis Parameter (Read Only)

Description: The **AXISSTATUS** axis parameter may be used to check various status bits held for each axis fitted:

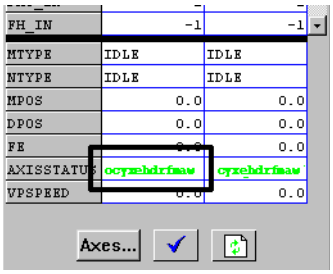
Bit	Description	Value	char
0	Unused	1	
1	Following error warning range	2	**w**
2	Communications error to remote drive	4	**a**
3	Remote drive error	8	**m**
4	In forward limit	16	**f**
5	In reverse limit	32	**r**
6	Datuming	64	**d**
7	Feedhold	128	**h**
8	Following error exceeds limit	256	**e**
9	In forward software limit	512	**x**
10	In reverse software limit	1024	**y**
11	Cancelling move	2048	**c**
12	Encoder power supply overload (MC206)	4096	**o**
13	Set on SSI axis after initialisation	8192	

The **AXISSTATUS** axis parameter is set by the system software is read-only..

Example:
```
IF (AXISSTATUS AND 16)>0 THEN
  PRINT "In forward limit"
ENDIF
```

Note: In the *Motion* Perfect parameter screen the **AXISSTATUS** parameter is displayed as a series of characters, **ocyxehdrfmaw**, as listed in the table above.

These characters are displayed in green lowercase letters normally, or red uppercase when set.



See Also: **ERRORMASK**

BOOST

Type: Axis Parameter

Syntax: **BOOST=ON / BOOST=OFF**

Description: Sets the boost output on a stepper daughter board. The boost output is a dedicated open collector output on the stepper and stepper encoder daughter boards. The open collector can be switched on or off for each axis using this command.

Example: **BOOST AXIS(5)=ON**

CAN_ADDRESS

Type: Axis Parameter

Description: The **CAN_ADDRESS** axis parameter is used when control is being made of remote servo drives with CAN communications. The **CAN_ADDRESS** holds the address of the remote servo drive.

CAN_ENABLE

Type: Axis Parameter

Description: The **CAN_ENABLE** axis parameter is used when control is being made of the remote servo drives with CAN communications. The **CAN_ENABLE** is used to control the enable on the remote servo drive.

CLOSE_WIN

Type: Axis Parameter

Alternate Format: **CW**

Description: By writing to this parameter the end of the window in which a registration mark is expected can be defined. The value is in user units.

Example: **CLOSE_WIN=10.5**

CLUTCH_RATE

Type: Axis Parameter

Description: This affects operation of **CONNECT** by changing the connection ratio at the specified rate/second.

Default **CLUTCH_RATE** is set very high to ensure compatibility with earlier versions.

Example:
```
CLUTCH_RATE=5
```

CREEP

Type: Axis Parameter

Description: Sets the creep speed on the current base axis. The creep speed is used for the slow part of a DATUM sequence. The creep speed must always be a positive value. When given a **DATUM** move the axis will move at the programmed **SPEED** until the datum input **DATUM_IN** goes low. The axis will then ramp the speed down and start a move in the reversed direction at the **CREEP** speed until the datum input goes high.

The creep speed is entered in units/sec programmed using the unit conversion factor. For example, if the unit conversion factor is set to the number of encoder edges/inch the speed is programmed in INCHES/SEC.

Example:
```
BASE(2)
CREEP=10
SPEED=500
DATUM(4)
CREEP AXIS(1)=10
SPEED AXIS(1)=500
DATUM(4) AXIS(1)
```

DAC

Type: Axis Parameter

Description: Writing to this axis parameter when **SERVO=OFF** allows the user to force a specified voltage on a servo axis. The range of values that a 12 bit DAC can take is:

DAC=-2048 corresponds to a voltage of 10V

to

DAC=2047 corresponds to a voltage of -10v

The range of values that a 16 bit DAC (MC206 built-in axes only) can take is:

DAC=32767 corresponds to a voltage of 10V

to

DAC=-32768 corresponds to a voltage of -10v

Note: The SERVO DAUGHTER BOARD/MC202 hardware inverts the signal compared to the number.

Example: To force a square wave of amplitude +/-5V and period of approximately 500ms on axis 0.

```
WDOG=ON
SERVO AXIS(0)=OFF
square:
  DAC AXIS(0)=1024
  WA(250)
  DAC AXIS(0)=-1024
  WA(250)
GOTO square
```

DAC_OUT

Type: Axis Parameter (Read Only)

Description: The axis DAC is the electronics hardware used to output +/-10volts to the servo drive when using a servo daughter board. The **DAC_OUT** parameter allows the value being used to be read back. The value put on the DAC comes from 2 potential sources:

If the axis parameter **SERVO** is set **OFF** then the axis parameter DAC is written to the axis hardware. If the **SERVO** parameter is **ON** then a value calculated using the servo algorithm is placed on the DAC. Either case can be read back using **DAC_OUT**. Values returned will be in the range -2048 to 2047.

Example:
```
>>PRINT DAC_OUT AXIS(8)
288.0000
>>
```

DAC_SCALE

Type: Axis Parameter

Description: The **DAC_SCALE** axis parameter is only available on the MC206, MC224, EURO205X and PCI208. The parameter has 2 purposes: It is set to value 16 on power up on the built-in axes of the MC206. This scales the values applied to the higher resolution DAC so that the gains required on the axis are similar to those required on the other controllers. **DAC_SCALE** may be set negative (-16) to reverse the polarity of the DAC output signal. When the servo is off the magnitude of **DAC_SCALE** is not important as the voltage applied is controlled by the DAC parameter. The polarity is still reversed however by **DAC_SCALE**. When a Servo Daughter Board is used in an MC206 the default **DAC_SCALE** parameter for that axis will be 1.

Example:

```
>>DAC_SCALE AXIS(3)=-16
>>
```

DATUM_IN

Type: Axis Parameter

Alternate Format: **DAT_IN**

Description: This parameter holds a digital input channel to be used as a datum input. The input can be in the range 0..31. If **DATUM_IN** is set to -1 (default) then no input is used as a datum.

The same input may also be used as a limit input if required.

Example:
```
DATUM_IN AXIS(0)=28
```

Note: Feedhold, forward, reverse, datum and jog inputs are ACTIVE LOW.

DECEL

Type: Axis Parameter

Syntax: **DECEL=value**

Description: The **DECEL** axis parameter may be used to set or read back the deceleration rate of each axis fitted. The deceleration rate will be returned in units/sec/sec.

Example:
```
DECEL=100' Set deceleration rate
PRINT " Decel is ";DECEL;" mm/sec/sec"
```

DEMAND_EDGES

Type: Axis Parameter (Read Only)

Description: Allows the user to read back the current **DPOS** in encoder edges.

Example:
```
>>PRINT DEMAND_EDGES AXIS(4)
```

DPOS

Type: Axis Parameter (Read Only)

Description: The demand position **DPOS** is the demanded axis position generated by the move commands. Its value may also be adjusted without doing a move by using the **DEFPOS()** or **OFFPOS** commands. It is reset to 0 on power up or software reset. The demand position must never be written to directly although a value can be forced to create a step change in position by writing to the **ENDMOVE** parameter if no moves are currently in progress on the axis.

Example:
```
>>? DPOS AXIS(10)
```
This will return the demand position in user units.

DRIVE_STATUS

Type: Axis Parameter

Alternate Format: **AMP_STATUS**

Description: Returns the status register of a drive with digital communications capability connected to the *Motion Coordinator*.

In the case of an SLM axis it returns the SLM and drive status:
Bits 0..7 return bits 0..7 of register 0x8000 on the drive. Bits 8..23 return register 0xD000 on the SLM.

Example:
```
>>PRINT DRIVE_STATUS AXIS(8)
0.0000
>>
```

D_GAIN

Type: Axis Parameter

Syntax: **D_GAIN=value**

Description: The derivative gain is a constant which is multiplied by the change in following error.

Adding derivative gain to a system is likely to produce a smoother response and allow the use of a higher proportional gain than could otherwise be used.

High values may lead to oscillation. For a derivative term *Kd* and a change in following error *Đe* the contribution to the output signal is:

$$Od=Kd \times Đe$$

Example: `D_GAIN=0.25`

Note: Servo gains have no effect in stepper motor axes.

ENCODER

Type: Axis Parameter (Read Only)

Description: The **ENCODER** axis parameter holds a raw copy of the encoder or resolver hardware register. On Servo daughter boards, for example, this is a 12 bit (Modulo 4096) number. On SSI absolute daughter boards the **ENCODER** register holds a value of the numbers of bits programmed with **SSI_BITS**. The **MPOS** axis measured position is calculated from the **ENCODER** value automatically allowing for overflows and offsets. On MC202 and the built-in axes of a Euro205 or MC206 the **ENCODER** register is a 14 bit register.

ENDMOVE

Type: Axis Parameter

Description: This parameter holds the position of the end of the current move in user units. It is normally only read back although may be written to if required provided that **SERVO=ON** and no move is in progress. This will produce a step change in **DPOS**. Making step changes in **DPOS** can easily lead to “Following error exceeds limit” errors unless the steps are small or the **FE_LIMIT** is high.

ERRORMASK

Type: Axis Parameter

Description: The value held in this parameter is bitwise **AND**ed with the **AXISSTATUS** parameter by every axis on every servo cycle to determine if a runtime error should switch off the enable (**WDOG**) relay. If the result of the **AND** operation is not zero the enable relay is switched OFF.

On the MC202, Euro 205 and MC216 the default setting is 256. This will trip the enable relay only if a following error condition occurs.

For the MC206, the default value is 268 which is set to also trap critical errors with digital drive communications.

See Also: **AXISSTATUS**

FAST_JOG

Type: Axis Parameter

Description: This parameter holds the input number to be used as the fast jog input. The input can be in the range 0..31. If **FAST_JOG** is set to -1 (default) then no input is used for the fast jog. If the **FAST_JOG** is asserted then the jog inputs use the axis **SPEED** for the jog functions, otherwise the **JOGSPEED** will be used.

Note: Feedhold, forward, reverse, datum and jog inputs are ACTIVE LOW.

FASTDEC

Type: Axis Parameter

Description: The **FASTDEC** axis parameter is a reserved word, but is not currently used in the motion generator program.

FE

Type: Axis Parameter (Read Only)

Description: This parameter is the position error, which is equal to the demand position(**DPOS**)-measured position(**MPOS**). The parameter is returned in user units.

FE_LIMIT

Type: Axis Parameter

Alternate Format: **FELIMIT**

Syntax: **FE_LIMIT=value**

Description: This is the maximum allowable following error. When exceeded the controller will generate a run time error and always resets the enable (**WDOG**) relay thus disabling further motor operation. This limit may be used to guard against fault conditions such as mechanical lock-up, loss of encoder feedback, etc. It is returned in USER UNITS.

The default value is 2000 encoder edges.

FERANGE

Type: Axis Parameter

Syntax: **FERANGE=value**

Description: Following error report range. When the following error exceeds this value on a servo axis the axis has bit 1 in the **AXISSTATUS** axis parameter set.

FEGRAD

Type: Axis Parameter

Syntax: **FEGRAD=value**

Description: Following error limit gradient. Specifies the allowable increase in following error per unit increase in velocity profile speed. The parameter is not currently used in the motion generator program.

FEMIN

Type: Axis Parameter

Syntax: **FEMIN=value**

Description: Following error limit at zero speed. The parameter is not currently used in the motion generator program.

FHOLD_IN

Type: Axis Parameter

Alternate Format: **FH_IN**

Syntax: **FHOLD_IN=value**

Description: This parameter holds the input number to be used as a feedhold input. The input can be in the range 0..31. If **FHOLD_IN** is set to -1 (default) then no input is used as a feedhold. When the feedhold input is set motion on the specified axis has its speed overridden to the Feedhold speed (**FHSPEED**) WITHOUT CANCELLING THE MOVE IN PROGRESS. This speed is usually zero. When the input is reset any move in progress when the input was set will go back to the programmed speed. Moves which are not speed controlled E.G. **CONNECT**, **CAMBOX**, **MOVELINK** are not affected.

Note: Feedhold, forward, reverse, datum and jog inputs are ACTIVE LOW.

FHSPEED

Type: Axis Parameter

Syntax: **FHSPEED=value**

Description: When the feedhold input is set motion is usually ramped down to zero speed as the feedhold speed is set to its default zero value. In some cases it may be desirable for the axis to ramp to a known constant speed when the feedhold input is set. To do this the **FHSPEED** parameter is set to a non zero value. The value is in user units/sec.

FS_LIMIT

Type: Axis Parameter

Alternate Format: **FSLIMIT**

Description: An end of travel limit may be set up in software thus allowing the program control of the working envelope of the machine. This parameter holds the absolute position of the forward travel limit in user units. When the limit is hit the controller will ramp down the speed to zero then cancel the move. Bit 9 of the **AXISSTATUS** register is set when the axis position is greater than the **FS_LIMIT**.

FWD_IN

Type: Axis Parameter

Description: This parameter holds the input number to be used as a forward limit input. The input can be in the range 0..31. If **FWD_IN** is set to -1 (default) then no input is used as a forward limit. When the forward limit input is asserted any forward motion on that axis is stopped.

Example: **FWD_IN=19**

Note: Feedhold, jog forward, reverse and datum inputs are ACTIVE LOW.

FWD_JOG

Type: Axis Parameter

Description: This parameter holds the input number to be used as a jog forward input. The input can be in the range 0..31. If **FWD_JOG** is set to -1 (default) then no input is used as a forward jog.

Example: **FWD_JOG=7**

Note: Feedhold, forward, reverse, datum and jog inputs are ACTIVE LOW.

INVERT_STEP

Type: Axis Parameter

Description: **INVERT_STEP** is used to switch a hardware inverter into the stepper pulse output circuit. This can be necessary in for connecting to some stepper drives. The electronic logic inside the *Motion Coordinator* stepper pulse generation assumes that the FALLING edge of the step output is the active edge which results in motor movement. This is suitable for the majority of stepper drives. Setting **INVERT_STEP=ON** effectively makes the RISING edge of the step signal the active edge. **INVERT_STEP** should be set if required prior to enabling the controller with **WDOG=ON**. Default=OFF.

Note: If the setting is incorrect. A stepper motor may lose position by one step when changing direction.

I_GAIN

Type: Axis Parameter

Description: The integral gain is a constant which is multiplied by the sum of following errors of all the previous samples. This term may often be set to 0 (Default). Adding integral gain to a servo system reduces position error when at rest or moving steadily but it will produce or increase overshoot and may lead to oscillation.

For an integral gain *Ki* and a sum of position errors $\oplus e$, the contribution to the output signal is:

$$Oi = Ki \times \oplus e$$

Note: Servo gains have no effect on stepper motor axes.

JOGSPEED

Type: Axis Parameter

Description: Sets the slow jog speed in user units for an axis to run at when performing a slow jog. A slow jog will be performed when a jog input for an axis has been declared and that input is low. The jog will be at the **JOGSPEED** provided the **FAST_JOG** input has not be declared and is set low. Two separate jog inputs are available for each axis **FWD_JOG** and **REV_JOG**.

LINKAX

Type: Axis Parameter (Read Only)

Description: Returns the axis number that the axis is linked to during any linked moves. Linked moves are where the demand position is a function of another axis. E.G. **CONNECT**, **CAMBOX**, **MOVELINK**

MARK

Type: Axis Parameter (Read Only)

Description: Returns **TRUE** when a registration event has occurred. This is set to **FALSE** by the **REGIST** command and set to true when the registration event occurs. When **TRUE** the **REG_POS** is valid.

Example:
```
loop:
  WAIT UNTIL IN(punch_clr)=ON
  MOVE(index_length)
  REGIST(3)' rising edge of R
  WAIT UNTIL MARK  MOVEMODIFY(REG_POS + offset)
  WAIT IDLE
GOTO loop
```

MARKB

Type: Axis Parameter (Read Only)

Description: Only usable on MC206 built-in axes. MARKB returns **TRUE** when the Z registration position has been latched. This is set to **FALSE** by the **REGIST** command and set to true when the registration event occurs. When **TRUE** the **REG_POSB** is valid. See REGIST() and REG_POSB.

MERGE

Type: Axis Parameter

Syntax: **MERGE=ON / MERGE=OFF**

Description: This is a software switch which can be used to enable or disable the merging of consecutive moves. With merging enabled, if the next move is already in the buffer the axis will not ramp down to zero speed but load up the following move allowing them to be seamlessly merged. Note that it is up to the programmer to ensure that the merging is sensible. For example merging a forward move with a reverse move will cause an attempted instantaneous change of direction.

MERGE will only function if:

1) The next move is loaded
2) Axis group does not change on multi-axis moves
3) Velocity profiled moves (**MOVE**, **MOVEABS**, **MOVECIRC**, **MHELICAL**, **REVERSE**, **FORWARD**) cannot be merged with linked moves (**CONNECT**,**MOVELINK**,**CAMBOX**)

Note: When merging multi-axis moves only the base axis **MERGE** flag needs to be set.

If the moves are short a high deceleration rate must be set to avoid the controller ramping the speed down in anticipation of the end of the buffered move

Example:
```
MERGE=OFF'    Decelerate at the end of each move
MERGE=ON'     Moves will be merged if possible
```

MICROSTEP

Type: Axis Parameter

Description: Sets microstepping mode when using a stepper daughter board, MC202 or Euro205. On these controllers the stepper pulse circuit contains a circuit which places the step pulses more evenly in time by dividing the pulse rate by 2 or 16:

MICROSTEP=OFF (DEFAULT)	62.5 kHz Maximum
MICROSTEP=ON	500 kHz Maximum

(On the MC206, PCI208 and EURO205X a different pulse generation circuit is used which always divides the pulse rate by 16 and is NOT affected by the MICROSTEP parameter. This circuit can generate pulses up to 2Mhz) The stepper daughter board can generate pulses at up to 62500 Hz with **MICROSTEP=OFF** (This is the default setting and should be used when the pulse rate does not exceed 62500 Hz even if the motor is microstepping) With **MICROSTEP=ON** the stepper board can generate pulses at up to 500,000 Hz although the pulses are not so evenly spaced in time.

With **MICROSTEP=OFF** the **UNITS** parameter should be set to 16 times the number of pulses in a distance parameter. With **MICROSTEP=ON** the **UNITS** should be set to 2 times the number.

Example:
```
UNITS AXIS(2)=180*2' 180 pulses/rev * 2
MICROSTEP AXIS(2)=ON
```

MPOS

Type: Axis Parameter (Read Only)

Description: This parameter is the position of the axis as measured by the encoder or resolver. It is reset to 0 (unless a resolver is fitted) on power up or software reset. The value is adjusted using the **DEFPOS()** command or **OFFPOS** axis parameter to shift the datum position or when the **REP_DIST** is in operation. The position is reported in user units.

Example:
```
WAIT UNTIL MPOS>=1250
SPEED=2.5
```

MSPEED

Type: Axis Parameter (Read Only)

Description: The **MSPEED** represents the change in measured position in user units (per second) in the last servo period. The **SERVO_PERIOD** defaults to 1mSec. It therefore can be used to represent the speed measured. This value represents a snapshot of the speed and significant fluctuations can occur, particularly at low speeds. It can be worthwhile to average several readings if a stable value is required at low speeds.

MTYPE

Type: Axis Parameter (Read Only)

Description: This parameter holds the type of move currently being executed.

MTYPE	Move Type
0	Idle (No move)
1	MOVE
2	MOVEABS
3	MHELICAL
4	MOVECIRC
5	MOVEMODIFY
10	FORWARD
11	REVERSE

MTYPE	Move Type
12	DATUMING
13	CAM
14	Forward Jog
15	Reverse Jog
20	CAMBOX
21	CONNECT
22	MOVELINK

This parameter may be interrogated to determine whether a move has finished or if a transition from one move type to another has taken place.

A non-idle move type does not necessarily mean that the axis is actually moving. It may be at zero speed part way along a move or interpolating with another axis without moving itself.

Note that testing MTYPE=0 is not equivilent to WAIT IDLE. WAIT IDLE is equivilent to WAIT UNTIL (MTYPE=0) AND (NTYPE=0) AND (PMOVE=0).

Example: Load a move and print its type. Note how because the MOVE becomes active on the next servo cycle the programmer MUST wait for the move to become active, prior to printing the type:

```
MOVE(1000)
WAIT LOADED
PRINT MTYPE
```

NTYPE

Type: Axis Parameter (Read Only)

Description: This parameter holds the type of the next buffered move. The values held are as for **MTYPE**. If no move is buffered zero will be returned. The **NTYPE** parameter is read only but the **NTYPE** can be cleared using **CANCEL(1)**

OFFPOS

Type: Axis Parameter

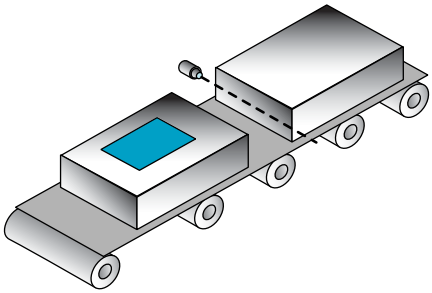
Description: The **OFFPOS** parameter allows the axis position to be offset by any value without affecting motion. **OFFPOS** can therefore be used to effectively datum a system at full speed.Values loaded into the **OFFPOS** axis parameter are reset to 0 by the system software as the axis position is changed.

Example: Define the current demand position as zero:

```
OFFPOS=-DPOS
WAIT UNTIL OFFPOS=0'   wait until applied
```

This is equivalent to **DEFPOS(0)**

Example 2: A conveyor is used to transport boxes onto which labels must be applied.



Using the **REGIST()** function, we can capture the position at which the leading edge of the box is seen, then by using **OFFPOS** we can adjust the measured position of the axis to be zero at that point. Therefore, after the registration event has occurred, the measured position (seen in **MPOS**) will actually reflect the absolute distance from the start of the box, the mechanism which applies the label can take advantage of the absolute position start mode of the **MOVELINK** or **CAMBOX** commands to apply the label.

```
BASE(conv)
REGIST(3)
WAIT UNTIL MARK
OFFPOS = -REG_POS ' Leading edge of box is now zero
```

Note: The **OFFPOS** adjustment is executed on the next servo period. Several Trio BASIC instructions may occur prior to the next servo period. Care must be taken to ensure these instructions do not assume the position shift has occurred.

OPEN_WIN

Type: Axis Parameter

Alternate Format: **OW**

Description: This parameter defines the position before which registration marks will be ignored if windowing is specified by the **REGIST()** command.

OUTLIMIT

Type: Axis Parameter

Description: The output limit restricts the voltage output from a servo axis to a lower value than the maximum. The value required varies depending on whether the axis has a 12 bit or 16 bit DAC. If the voltage output is generated by a 12 bit DAC values an OUTLIMIT of 2047 will produce the full +/-10v range. If the voltage output is generated by a 16 bit DAC values an OUTLIMIT of 32767 will produce the full +/-10v range. Only the MC206 internal axes have 16 bit DAC's

Example: `OUTLIMIT AXIS(0)=1023`

The above will limit the voltage output to a ±5V output range on a servo daughter board axis. This will apply to the **DAC** command if **SERVO=OFF** or to the voltage output by the servo if **SERVO=ON**.

OV_GAIN

Type: Axis Parameter

Description: The output velocity gain is a gain constant which is multiplied by the change in measured position. The result is summed with all the other gain terms and applied to the servo DAC. Default value is 0. Adding NEGATIVE output velocity gain to a system is mechanically equivalent to adding damping. It is likely to produce a smoother response and allow the use of a higher proportional gain than could otherwise be used, but at the expense of higher following errors. High values may lead to oscillation and produce high following errors. For an output velocity term Kov and change in position ΔPm, the contribution to the output signal is:

$$Oov = Kov \times \Delta Pm$$

Note: Negative values are normally required. Servo gains have no effect on stepper motor axes.

PP_STEP

Type: Axis parameter

Description: This parameter allows the incoming raw encoder counts to be multiplied by an integer value in the range -1024 to 1023. This can be used to match encoders to high resolution microstepping motors for position verification or for moving along circular arcs on machines where the number of encoder edges/distance do not match on the axes. Using a negative number will reverse the encoder count.

Example: A microstepping motor has 20000 steps/rev. The *Motion Coordinator* is working in **MICROSTEP=ON** mode so will internally process 40000 counts/rev. A 2500 pulse encoder is to be connected. This will generate 10000 edge counts/rev. A multiplication factor of 4 is therefore is required to convert the 10000 counts/rev to match the 40000 counts/rev of the motor.

```
PP_STEP AXIS(3)=4
```

Example 2: An X-Y machine has encoders which give 50 edges/mm in the X axis (Axis 0) and 75 edges/mm in the Y axis (Axis 1). Circular arc interpolation is required between the axes. This requires that the interpolating axes have the same number of encoder counts/distance. It is not possible to multiply the X axis counts by 1.5 as the **PP_STEP** parameter must be an integer. Both X and Y axes must therefore be set to give 150 edges/mm:

```
PP_STEP AXIS(0)=3
PP_STEP AXIS(1)=2
UNITS AXIS(0)=150
UNITS AXIS(1)=150
```

Note: In a servo loop, increasing PP_STEP will require a proportionate decrease of loop gains.

P_GAIN

Type: Axis Parameter

Description: The proportional gain sets the 'stiffness' of the servo response. Values that are too high will produce oscillation. Values that are too low will produce large following errors.
For a proportional gain *Kp* and position error *e*, its contribution to the output signal is:

$$Op=Kp \times e$$

Note: **P_GAIN** may be fractional values. The default value is 1.0. Servo gains have no effect on stepper motor axes.

Example:
```
P_GAIN AXIS(11)=0.25
```

REG_MATCH

Type: Axis Parameter (Read Only)

Description: Indicates to a programmer the quality of the fit of a **RECORD** / **MATCH** sequence. A value of 1 is returned if the fit is 100%.

Note: See the **MATCH** command for an example of a complete recognition sequence.

REG_POS

Type: Axis Parameter (Read Only)

Alternate Format: **RPOS**

Description: Stores the position at which a registration mark was seen on each axis in user units. See **REGIST()** for more details.

Example: A paper cutting machine uses a **CAM** profile shape to quickly draw paper through servo driven rollers then stop it whilst it is cut. The paper is printed with a registration mark. This mark is detected and the length of the next sheet is adjusted by scaling the CAM profile with the third parameter of the **CAM** command:

```
'  Example Registration Program using CAM stretching:
' Set window open and close:
  length=200
  OPEN_WIN=10
  CLOSE_WIN=length-10
  GOSUB Initial
Loop:
  TICKS=0' Set millisecond counter to 0
  IF MARK THEN
    offset=REG_POS
  '  This next line makes offset -ve if at end of sheet:
    IF ABS(offset-length)<offset THEN offset=offset-length
```

```
      PRINT "Mark seen at:"offset[5.1]
  ELSE
      offset=0
    PRINT "Mark not seen"
    ENDIF

  ' Reset registration prior to each move:
  DEFPOS(0)
    REGIST(3+768)' Allow mark at first 10mm/last 10mm of sheet
    CAM(0,50,(length+offset*0.5)*cf,1000)
  WAIT UNTIL TICKS<-500
  GOTO Loop
```

(variable “cf” is a constant which would be calculated depending on the machine draw length per encoder edge)

REG_POSB

Type: Axis Parameter (Read Only)

Description: Useable only on MC206 built-in axes. REG_POSB returns the position at which a registration Z mark was seen on an axis. See **REGIST()** for more details.

REMAIN

Type: Axis Parameter (Read Only)

Description: This is the distance remaining to the end of the current move. It may be tested to see what amount of the move has been completed. The units are user distance units.

Example: To change the speed to a slower value 5mm from the end of a move.

```
start:
  SPEED=10
  MOVE(45)
  WAIT UNTIL REMAIN<5
  SPEED=1
  WAIT IDLE
```

REMOTE_ERROR

Type: Axis Parameter

Description: Returns the number of errors on a drive's digital communication link.

Example:
```
>>PRINT REMOTE_ERROR
1.0000
>>
```

REPDIST

Type: Axis Parameter

Description: The repeat distance contains the allowable range of movement for an axis before the position count overflows or underflows. For example, when an axis executes a **FORWARD** move the demand and measured position will continually increase. When the measured position reaches the **REPDIST** twice that distance is subtracted to ensure that the axis always stays in the range **-REPEAT DISTANCE** to **+REPEAT DISTANCE** (Assuming **REP_OPTION=OFF**). The *Motion Coordinator* will adjust its absolute position without affecting the move in progress or the servo algorithm.

REP_OPTION

Type: Axis Parameter

Description: Bit 0 of the **REP_OPTION** parameter controls the way the REPDIST is applied. In the default setting (**REP_OPTION bit 0=0**) REPDIST operation is selected in the range **-REPEAT DISTANCE** to **+REPEAT DISTANCE**. In some circumstances it more convenient for the axis positions to be specified from 0 to **+REPEAT DISTANCE**. (**REP_OPTION bit 0=1**)

REP_OPTION bit 1 is set **ON** to switch OFF the automatic repeat option of the **CAMBOX** or MOVELINK function. When the system software has set the option OFF it automatically clears bit 1 of **REP_OPTION**.

REV_IN

Type: Axis Parameter

Description: This parameter holds the input number to be used as a reverse limit input. The input should be in the range 0..31. If **REV_IN** is set to -1 (default) then no input is used as a reverse limit. When the reverse limit input is asserted moves going in the reverse direction will be cancelled. The axis status bit 5 will also be set.

Note: Feedhold, forward, reverse and datum inputs are ACTIVE LOW.

REV_JOG

Type: Axis Parameter

Description: This parameter holds the input number to be used as a reverse jog input. The input should be in the range 0..31. If **REV_JOG** is set to -1 (default) then no input is used as a reverse jog. When the input is asserted then the axis is moved forward at the **JOGSPEED** or axis **SPEED** depending on the status of the **FAST_JOG** input.

Note: Feedhold, forward, reverse and datum inputs are ACTIVE LOW.

RS_LIMIT

Type: Axis Parameter

Alternate Format: **RSLIMIT**

Description: An end of travel software limit may be set up in software thus allowing the program control of the working envelope of the machine. This parameter holds the absolute position of the reverse travel limit in user units. When the limit is hit the controller will ramp down the speed to zero then cancel the move. Bit 10 in the axis status parameter is set when the axis is in the **RS_LIMIT**

SERVO

Type: Axis Parameter

Description: On a servo axis this parameter determines whether the axis runs under servo control or open loop. When **SERVO=OFF** the axis hardware will output a voltage dependent on the DAC parameter. When **SERVO=ON** the axis hardware will output a voltage dependent on the gain settings and the following error.

SERVO is also used on stepper axes with position verification. If **SERVO=ON** the system software will compare the difference between the **DPOS** and **MPOS (FE)** on the axis with the **FE_LIMIT**. If the difference exceeds the limit the following error bit is set in the **AXISSTATUS** register, the enable relay is forced OFF and the servo is set OFF. If the **SERVO=OFF** on a stepper verification axis the **FE** is not compared with the **FE_LIMIT**.

Example:
```
SERVO AXIS(0)=ON'   Axis 0 is under servo control
SERVO AXIS(1)=OFF'  Axis 1 is run open loop
```

Note: Stepper axes with position verification need consideration also of **VERIFY** and **PP_STEP**.

SP

Type: Axis Command - Use the **SPEED** axis parameter for new applications.

Syntax: **SP(speed)**

Description: This format is only provided to simplify compatibility with earlier controllers. Sets demand speed of the current or base axis.

SPEED

Type: Axis Parameter

Description: The **SPEED** axis parameter can be used to set/read back the demand speed axis parameter. The speed is returned in units/s. The demand speed is the speed ramped up to during the movement commands **MOVE**, **MOVEABS**, **MOVECIRC**, **FORWARD**, **REVERSE**, **MHELICAL and MOVEMODIFY**.

Example:
```
SPEED=1000
PRINT "Speed Set=";SPEED
```

SRAMP

Type: Axis Parameter

Description: This parameter stores the s-ramp factor. This controls the amount of rounding applied to trapezoidal profiles. 0 sets no rounding. 10 maximum rounding. Using S ramps increases the time required for the movement to complete. **SRAMP** can be used with **MOVE**, **MOVEABS**, **MOVECIRC**, **MHELICAL**, **FORWARD**, **REVERSE** and **MOVEMODIFY** move types.

Note: The **SRAMP** factor should not be changed while a move is in progress.

SSI_BITS

Type: Axis Parameter

Description: This parameter is only used with the SSI Absolute daughter board. It is used to set the number of data bits to be clocked out of the encoder by the axis hardware. The maximum permitted value is 24. The default value is 0 which will cause no data to be clocked from the SSI encoder, users MUST therefore set a value depending on the encoder type.

If the number of SSI_BITS is to be changed, the parameter must first be set to zero before entering the new value.

Example:
```
SSI_BITS AXIS(3)=12
SSI_BITS AXIS(7)=21
```

Example2:
```
're-initialise MPOS using absolute value from encoder
SERVO=OFF
SSI_BITS =0
SSI_BITS =databits
```

TRANS_DPOS

Type: Axis Parameter (Read Only)

Description: Axis demand position at output of frame transformation. **TRANS_DPOS** is normally equal to **DPOS** on each axis. The frame transformation is therefore equivalent to 1:1 for each axis. For some machinery configurations it can be useful to install a frame transformation which is not 1:1, these are typically machines such as robotic arms or machines with parasitic motions on the axes. Frame transformations have to be specially written in the "C" language and downloaded into the controller. It is essential to contact Trio if you want to install frame transformations.

Note: See also **FRAME**

TRANSITIONS

Type: Axis Parameter

Description: Records the number of register input transitions in a **REGIST** sequence

See also **REGIST**, **RECORD**, **MATCH**

UNITS

Type: Axis Parameter

Description: The unit conversion factor sets the number of encoder edges/stepper pulses in a user unit. The motion commands to set speeds, acceleration and moves use the **UNITS** parameter to allow values to be entered in more convenient units e.g.: mm for a move or mm/sec for a speed.

Note: Units may be any positive value but it is recommended to design systems with an integer number of encoder pulses/user unit.

Example: A leadscrew arrangement has a 5mm pitch and a 1000 pulse/rev encoder. The units should be set to allow moves to be specified in mm. The 1000 pulses/rev will generate 1000 x 4=4000 edges/rev. One rev is equal to 5mm therefore there are 4000/5=800 edges/mm so:

```
>>UNITS=1000*4/5
```

Example 2: A stepper motor has 180 pulses/rev and is being used with **MICROSTEP=OFF**

To program in revolutions the unit conversion factor will be:

```
>>UNITS=180*16
```

Note: Users with stepper axes should also read the **MICROSTEP** command when choosing **UNITS**.

VERIFY

Type: Axis Parameter

Description: The verify axis parameter is used to select different modes of operation on a stepper encoder axis.

VERIFY=OFF
Encoder count circuit is connected to the **STEP** and **DIRECTION** hardware signals so that these are counted as if they were encoder signals. This is particularly useful for registration as the registration circuit can therefore function on a stepper axis.

VERIFY=ON
Encoder circuit is connected to external A,B, Z signal

Note: On the MC202 and the Euro205 when **VERIFY=OFF**, the encoder counting circuit is configured to accept **STEP** and **DIRECTION** signals hard wired to the encoder A and B inputs. If **VERIFY=ON**, the encoder circuit is configured for the usual quadrature input.

Take care that the encoder inputs do not exceed 5 volts.

Example:
```
VERIFY AXIS(3)=ON
```

VFF_GAIN

Type: Axis Parameter

Description: The velocity feed forward gain is a constant which is multiplied by the change in demand position. Adding velocity feed forward gain to a system decreases the following error during a move by increasing the output proportionally with the speed. For a velocity feed forward term *Kvff* and change in position ΔPd, the contribution to the output signal is:

$$Ovff = Kvff \times \Delta Pd$$

Note: Servo gains have no effect on stepper motor axes.

VP_SPEED

Type: Axis Parameter (Read Only)

Alternate Format: `VPSPEED`

Description: The velocity profile speed is an internal speed which is ramped up and down as the movement is velocity profiled. It is reported in user units/sec.

Example: Wait until command speed is achieved:

```
MOVE(100)
WAIT UNTIL SPEED=VP_SPEED
```