CHAPTER

8

TRIO BASIC COMMANDS

Motion and Axis Commands 13
ACC 13
ADD_DAC 14
ADDAX 16
AXIS 19
BACKLASH 20
BASE 21
CAM 22
CAMBOX 27
CANCEL 36
CONNECT 38
DATUM 40
DEC 44
DEFPOS 45
DISABLE_GROUP 47
ENCODER_RATIO 50
FORWARD 52
MHELICAL 54
MHELICALSP 57
MOVE 57
MOVEABS 59
MOVEABSSP 62
MOVECIRC 63
MOVECIRCSP 65
MOVELINK 66
MOVEMODIFY 71
MOVESP 75
MSPHERICAL 75
MOVETANG 77
RAPIDSTOP 80
REGIST 83
REGIST_SPEED 87
REVERSE 88
STEP_RATIO 90

Input / Output Commands 92
AIN 92
AIN0..3 / AINBI0..3 93
AOUT0...3 93
CHANNEL_READ 93
CHANNEL_WRITE 94
CHR 94
CLOSE 95
CURSOR 95

DEFKEY 95
ENABLE_OP 96
FILE 96
FLAG 98
FLAGS 99
GET 99
GET# 100
HEX 101
IN()/IN 101
INPUT 102
INPUTS0 / INPUTS1 103
INVERT_IN 103
KEY 103
LINPUT 104
OP 105
OPEN 107
PRINT 108
PRINT# 109
PSWITCH 110
READ_OP() 112
READPACKET 112
SEND 113
SETCOM 114
TIMER 116

Program Loops and Structures 118
BASICERROR 118
ELSE 118
ELSEIF 118
ENDIF 119
FOR..TO.. STEP..NEXT 120
GOSUB 121
GOTO 122
NEXT 122
ON.. GOSUB 122
ON.. GOTO 123
REPEAT.. UNTIL 123
RETURN 124
THEN 124
TO 124
UNTIL 125
WA 125
WAIT IDLE 126
WAIT LOADED 126

WAIT UNTIL 126
WEND 127
WHILE 127

System Parameters and Commands 128
ADDRESS 128
APPENDPROG 128
AUTORUN 128
AXISVALUES 129
BATTERY_LOW 129
BOOT_LOADER 130
BREAK_ADD 130
BREAK_DELETE 130
BREAK_LIST 131
BREAK_RESET 131
CAN 131
CANIO_ADDRESS 133
CANIO_ENABLE 134
CANIO_STATUS. 134
CANOPEN_OP_RATE 135
CHECKSUM 135
CLEAR 135
CLEAR_PARAMS 136
COMMSERROR 136
COMMSTYPE 137
COMPILE 137
CONTROL 138
COPY 138
DATE 139
DATE$ 140
DAY 140
DAY$ 140
DEL 140
DEVICENET 141
DIR 142
DISPLAY 142
DLINK 143
EDIT 147
EDPROG 147
EPROM 148
ERROR_AXIS 149
ETHERNET 149
ETHERNET_IP 151
EX 151

EXECUTE 152
FB_SET 152
FB_STATUS 153
FEATURE_ENABLE 153
FLASHVR 154
FRAME 155
FREE 156
HALT 156
HLM_COMMAND 157
HLM_READ. 159
HLM_STATUS 160
HLM_TIMEOUT 160
HLM_WRITE 161
HLS_MODEL 162
HLS_NODE 162
INCLUDE 163
INITIALISE 163
LAST_AXIS 164
LIST 164
LIST_GLOBAL 164
LOAD_PROJECT 165
LOADSYSTEM 165
LOCK 166
MC_TABLE 167
MC_VR 167
MOTION_ERROR 167
MPE 167
N_ANA_OUT 168
NAIO 169
NETSTAT 169
NEW 169
NIO 170
PEEK 170
POKE 170
PORT 170
POWER_UP 171
PROCESS 171
PROFIBUS 171
PROTOCOL 172
REMOTE 172
RENAME 172
RS232_SPEED_MODE 173
RUN 173
RUNTYPE 174

SCOPE 175
SCOPE_POS 176
SELECT 176
SERCOS 176
SERCOS_PHASE 181
SERIAL_NUMBER 182
SERVO_PERIOD 182
SLOT 182
STEP 183
STEPLINE 183
STOP 184
STICK_READ 184
STICK_WRITE 185
STORE 186
SYNC_TIMER 187
TABLE 187
TABLEVALUES 188
TIME 189
TIME$ 189
TRIGGER 189
TROFF 190
TRON 190
TSIZE 190
UNLOCK 191
USB 191
USB_HEARTBEAT 192
USB_STALL 193
VERSION 193
VIEW 193
VR 194
VRSTRING 195
WDOG 195
WDOGB 196
: 196
' 197
. 197
$ 198
BITREV8 198
ERROR_LINE 198
INDEVICE 199
LOOKUP 199
OUTDEVICE 200
PMOVE 200
PROC 200

PROC_LINE 201
PROC_MODE 201
PROC_STATUS 201
PROCNUMBER 202
RESET 202
RUN_ERROR 202
SHIFTR 202
STRTOD 203
TABLE_POINTER 204
TICKS 206

Mathematical Operations and Commands 207

+ Add 207
- Subtract 207
* Multiply 208
/ Divide 208
^ Power 209
= Equals 209
<> Not Equal 209
> Greater Than 210
>= Greater Than or Equal 210
< Less Than 211
<= Less Than or Equal 211
ABS 212
ACOS 212
AND 212
ASIN 213
ATAN 214
ATAN2 214
B_SPLINE 214
CLEAR_BIT 217
CONSTANT 217
COS 218
CRC16 218
EXP 219
FRAC 219
GLOBAL 219
IEEE_IN 220
IEEE_OUT 220
INT 221
INTEGER_READ/INTEGER_WRITE 222
LN 222
MOD 223
NOT 223

OR 223
READ_BIT 224
SET_BIT 225
SGN 225
SIN 225
SQR 227
TAN 227
XOR 227

Constants 229

OFF 229
ON 229
FALSE 229
PI 229
TRUE 230

Axis Parameters 231

ACCEL 231
ADDAX_AXIS 231
AFF_GAIN 231
ATYPE 232
AXIS_ADDRESS 234
AXIS_ENABLE 234
AXIS_MODE 235
AXISSTATUS 235
BACKLASH_DIST 236
BOOST 237
CAN_ENABLE 237
CLOSE_WIN 237
CLUTCH_RATE 237
CREEP 238
D_GAIN 238
D_ZONE_MIN 239
D_ZONE_MAX 239
DAC 240
DAC_OUT 240
DAC_SCALE 241
DATUM_IN 241
DECEL 242
DEMAND_EDGES 242
DEMAND_SPEED 242
DPOS 242
DRIVE_CLEAR 243
DRIVE_CONTROL 243

DRIVE_ENABLE 244
DRIVE_EPROM 244
DRIVE_HOME 244
DRIVE_INPUTS 244
DRIVE_INTERFACE 245
DRIVE_MODE 245
DRIVE_MONITOR 245
DRIVE_READ 246
DRIVE_RESET 246
DRIVE_STATUS 246
DRIVE_WRITE 247
ENCODER 247
ENCODER_BITS 247
ENCODER_CONTROL 248
ENCODER_ID 249
ENCODER_READ 249
ENCODER_STATUS 249
ENCODER_TURNS 249
ENCODER_WRITE 250
ENDMOVE 250
ENDMOVE_BUFFER2. 50
ENDMOVE_SPEED 251
ERRORMASK 251
FAST_JOG 252
FASTDEC 252
FE 252
FE_LATCH 252
FE_LIMIT 253
FE_LIMIT_MODE 253
FE_RANGE 253
FEGRAD 254
FEMIN 254
FHOLD_IN 254
FHSPEED 255
FORCE_SPEED 255
FS_LIMIT 256
FULL_SP_RADIUS 256
FWD_IN 257
FWD_JOG 257
I_GAIN 257
INVERT_STEP 258
JOGSPEED 258
LIMIT_BUFFERED 258
LINKAX 259

MARK 259
MARKB 259
MERGE 260
MICROSTEP 260
MOVES_BUFFERED 261
MPOS 261
MSPEED 262
MTYPE 262
NTYPE 263
OFFPOS 263
OPEN_WIN 264
OUTLIMIT 264
OV_GAIN 265
P_GAIN 265
PP_STEP 266
REG_POS 266
REMAIN 267
REP_DIST 268
REP_OPTION 268
REV_IN 268
REV_JOG 269
RS_LIMIT 269
SERVO 269
SPEED 270
SPHERE_CENTRE 270
SRAMP 270
TANG_DIRECTION 271
TRANS_DPOS 271
UNITS 272
VECTOR_BUFFERED 272
VERIFY 273
VFF_GAIN 274
VP_SPEED 274

Motion and Axis Commands

ACC

Type: Axis Command

Syntax: **ACC(rate)**

Description: Sets both the acceleration and deceleration rate simultaneously.

This command is provided to aid compatibility with older Trio controllers. Use the **ACCEL** and **DECEL** axis parameters in new programs.

Parameters:

rate:	The acceleration rate in UNITS/SEC/SEC.

Example 1: Move an axis at a given speed and using the same rates for both acceleration and deceleration.

```
ACC(120)    'set accel and decel to 120 units/sec/sec
SPEED=14.5  'set programmed speed to 14.5 units/sec
MOVE(200)   'start a relative move with distance of 200
```

Example 2: Changing the ACC whilst motion is in progress.

```
SPEED=100000              'set required target speed (units/sec)
ACC(1000)                 'set initial acc rate
FORWARD
WAIT UNTIL VP_SPEED>5000  'wait for acutal speed to exceed 5000
ACC(100000)               'change to high acc rate
WAIT UNTIL SPEED=VP_SPEED 'wait until final speed is reached
WAIT UNTIL IN(2)=OFF
CANCEL
```

ADD_DAC

Type: Axis Command

Syntax: **ADD_DAC(axis)**

Description: Adds the output from the 5-term servo control block of a secondary axis to the output of the base axis. The resulting **DAC_OUT** is then the sum of the two control loop outputs.

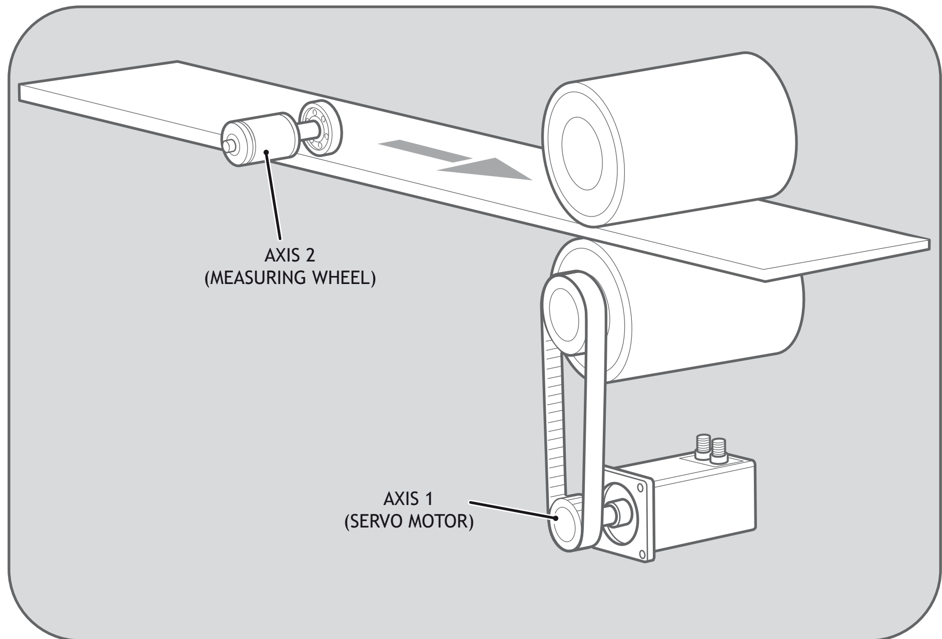
The **ADD_DAC** command is provided to allow a secondary encoder to be used on a servo axis to implement dual feedback control. This would typically be used in applications such as a roll-feed where you need a secondary encoder to compensate for slippage.

Parameters:

axis:	Number of the second axis, who's output will be added to the current axis. -1 will terminate the ADD_DAC link.

Example 1: Use **ADD_DAC** to add the output of a measuring wheel to the servo motor axis controlling a roll-feed. Set up the servo motor axis as usual with encoder feedback from the motor drive. The measuring wheel axis must also be set up as a servo by setting the **AYTPE** to 2. This is so that the software will perform the servo control calcualtions on that axis.

It is necessary for the two axes to be controlled by a common demand position. Typically this would be achieved by running the moves on a virtual axis and using **ADDAX** to produce a matching **DPOS** on BOTH axes. The servo gains are then set up on BOTH axes, and the output summed on to one physical output using **ADD_DAC**. If the required demand positions on both axes are not identical due to a difference in resolution between the 2 feedback devices, **ENCODER_RATIO** can be used on one axis to produce matching **UNITS**.




```
BASE(1)
'match the encoder counts per linear distance of the 2 axes
ENCODER_RATIO(counts_per_mm2, counts_per_mm1)
UNITS AXIS(1) = counts_per_mm1
UNITS AXIS(2) = counts_per_mm1 ‘ units MUST be the same
ADD_DAC(2)          'Combine axis(2) DAC_OUT with axis(1)
ADDAX(1) AXIS(2)    'Superimpose axis 1 demand on axis 2
                    'the axes are now set up and ready to move
MOVE(1200)
WAIT IDLE
```

ADDAX

Type: Axis 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.

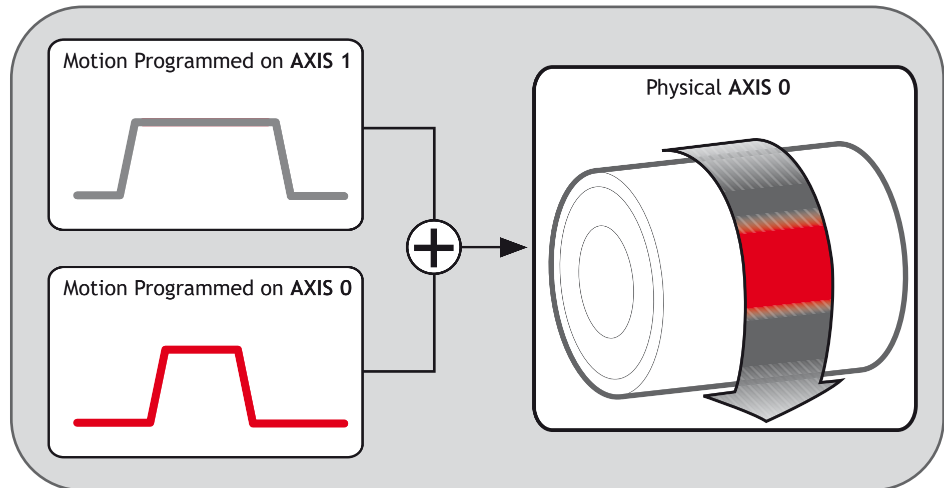
The **ADDAX** command therefore allows an axis to perform the moves specified on TWO axes added together. When the axis parameter **SERVO** is set to **OFF** on an axis with an encoder interface the measured position **MPOS** is copied into the demand position **DPOS**. This allows **ADDAX** to be used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

Parameter: **axis:** Axis to superimpose.
-1 breaks the link with the other axis.

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

Example 1:

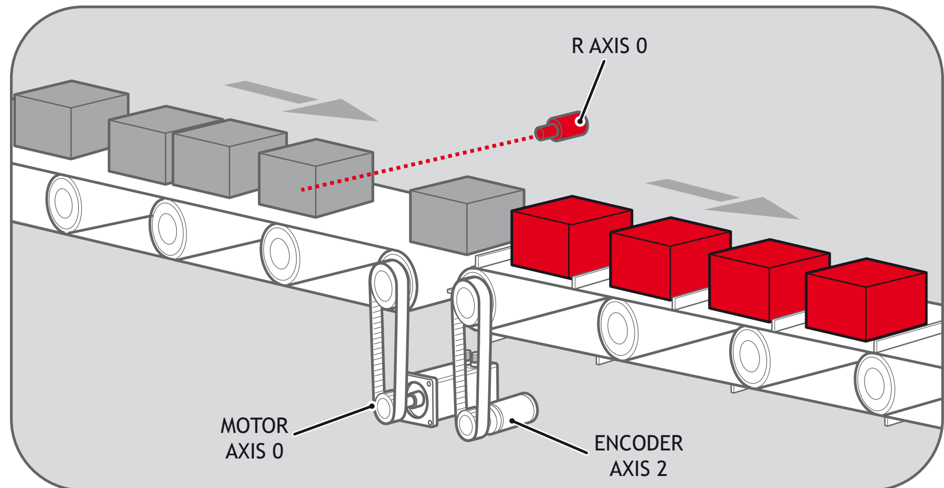
```
UNITS AXIS(0)=1000
UNITS AXIS(1)=20
'Superimpose axis 1 on axis 0
ADDAX(1) AXIS(0)
MOVE(1) AXIS(0)
MOVE(2) AXIS(1)
'Axis 0 will move 1*1000+2*20=1040 edges
```



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

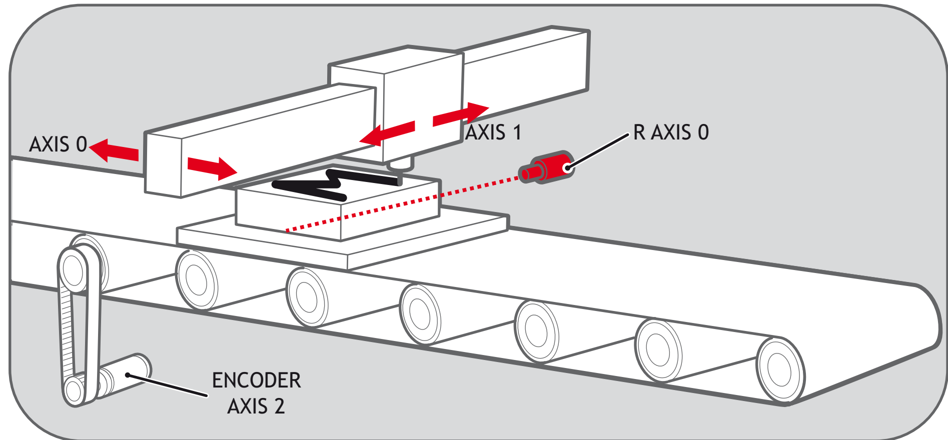
```
expected=2000 'sets expected position
BASE(0)
ADDAX(1)
CONNECT(1,2)    'continuous geared connection to flighted belt
REPEAT
  GOSUB getoffset         'get offset to apply
  MOVE(offset) AXIS(1)    'make correcting move on virtual axis
UNTIL IN(2)=OFF           'repeat until stop signal on input 2
RAPIDSTOP
ADDAX(-1)                 'clear ADDAX connection
STOP

getoffset:      'sub routine to register the position of the
                'piece and calculate the offset
BASE(0)
REGIST(3)
WAIT UNTIL MARK
seenat=REG_POS
offset=expected-seenat
RETURN
```

Axis 0 in this example is connected to the second conveyor's encoder and a superimposed **MOVE** on axis 1 is used to apply offsets



Example 3: An XY marking machine must mark boxes as they move along a conveyor. Using **CONNECT** enables the X marking axis to follow the conveyor. A virtual axis is used to program the marking absolute positions; this is then superimposed onto the X axis using **ADDAX**.

```
  ATYPE AXIS(3)=0      'set axis 3 as virtual axis
  SERVO AXIS(3)=ON
  DEFPOS(0) AXIS(3)
  ADDAX (3)AXIS(0)     'connect axis 3 requirement to axis 0
  WHILE IN(2)=ON
    REGIST(3)  'registration input detects a box on the conveyor
    WAIT UNTIL MARK OR IN(2)=OFF
    IF MARK THEN
      CONNECT(1,2) AXIS(0)'connect axis 0 to the moving belt
      BASE(3,1) 'set the drawing motion to axis 3 and 1
      'Draw the M
      MOVEABS(1200,0)'move A > B
      MOVEABS(600,1500)'move B > C
      MOVEABS(1200,3000)' move C > D
      MOVEABS(0,0)'move D > E
      WAIT IDLE
      BASE(0)
      CANCEL           'stop axis 0 from folowing the belt
      WAIT IDLE
      MOVEABS(0)       'move axis 0 to home position
    ENDIF
  WEND
CANCEL
```



AXIS

Type: Modifier

Syntax: **AXIS(expression)**

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

If it is required to change the axis used in every subsequent command, the **BASE** command should be used instead.

Parameters: **Expression:** Any valid Trio BASIC expression. The result of the expression should be a valid integer axis number.

Example 1: The command line has a default base axis of 0. To print the measured position of axis 3 to the terminal in *Motion* Perfect, you must add the axis number after the parameter name.

```
>>PRINT MPOS AXIS(3)
```

Example 2: The base axis is 0, but it is required to start moves on other axes as well as the base axis.

```
MOVE(450)                'Start a move on the base axis (axis 0)
MOVE(300) AXIS(2)        'Start a move on axis 2
MOVEABS(120) AXIS(5)     'Start an absolute move on axis 5
```

Example 3: Set up the repeat distance and repeat option on axis 3, then return to using the base axis for all later commands.

```
REP_DIST AXIS(3)=100
REP_OPTION AXIS(3)=1
SPEED=2.30 'set speed accel and decel on the BASE axis
ACCEL=5.35
DECEL=8.55
```

See Also: **BASE()**

BACKLASH

Type: Motion Command

Syntax: **BACKLASH(on/off, distance, speed, accel)**

Description: This axis function allows the parameters for the backlash compensation to be loaded. The backlash compensation is achieved by applying an offset move when the motor demand is in one direction, then reversing the offset move when the motor demand is in the opposite direction. These moves are superimposed on the commanded axis movements.

Parameters:

on/off: Control flag: ON to enable backlash.

distance: The distance to be offset in user units.

speed: The speed at which is the compensation move is applied in user units.

accel: The accel/decel rate at which is compensation move is applied in user units.

The backlash compensation is applied after a reversal of the direction of change of the **DPOS** parameter.
The backlash compensation can be seen in the TRANS_DPOS axis parameter. This is effectively **DPOS** + backlash compensation.

Example 1:

```
'Apply backlash compensation on axes 0 and 1:

BACKLASH(ON,0.5,10,50) AXIS(0)
BACKLASH(ON,0.4,8,50) AXIS(1)
```

BASE

Type: Motion Command

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

Alternate Format: **BA(...)**

Description: The **BASE** command is used to direct all 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 BASE 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.

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: Set up calibration units, speed and acceleration factors for axes 1 and 2.

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

Example 2: Set up an interpolated move to run on axes; 0 (x), 6 (y) and 4 (z). Axis 0 will move 100 units, axis 0 will move -23.1 and axis 4 will move 1250 units. The axes will move along the resultant path at the speed and acceleration set for axis 0.

```
BASE(0,6,4)
SPEED=120
ACCEL=2000
DECEL=2500
```

```
MOVE(100,-23.1,1250)
```

Note 1: 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:

```
'Set BASE array on a 16 axis MC224 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 2: Command line process ONLY; the **BASE** array may be seen by typing **BASE** with no parameters. For example on an MC206X with 8 axes:

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

See Also: **AXIS()**
The **AXIS()** command also redirects commands to different axes but applies to just a single command, and to a single axis.

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 3 up to the maximum table size available. 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 parameter relates the speed of the axis to the time taken to complete the cam profile. The time taken can be calculated using the current axis speed and this distance parameter (which are in user units).

For example the system is being programmed in mm and the speed is set to 10mm/sec. If it is required to take 10 seconds to complete the profile a distance of 100mm should be specified. 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).

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

Example1: Motion is required to follow the POSITION equation:

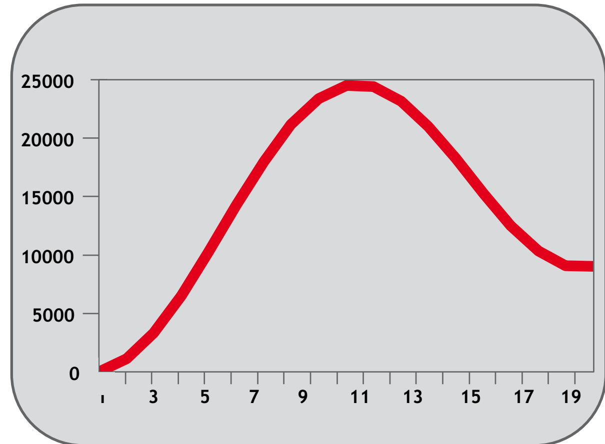
t(x) = x*25 + 10000(1-cos(x))

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:

```
FOR deg=0 TO 360 STEP 20   'loop to fill in the table
  rad = deg * 2 * PI/360   'convert degrees to radians
  x = deg * 25 + 10000 * (1-COS(rad))
  TABLE(deg/20,x)          'place value of x in table
NEXT deg

WHILE IN(2)=ON             'repeat cam motion while input 2 is on
  CAM(0,18,1,200)
  WAIT IDLE
WEND
```

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



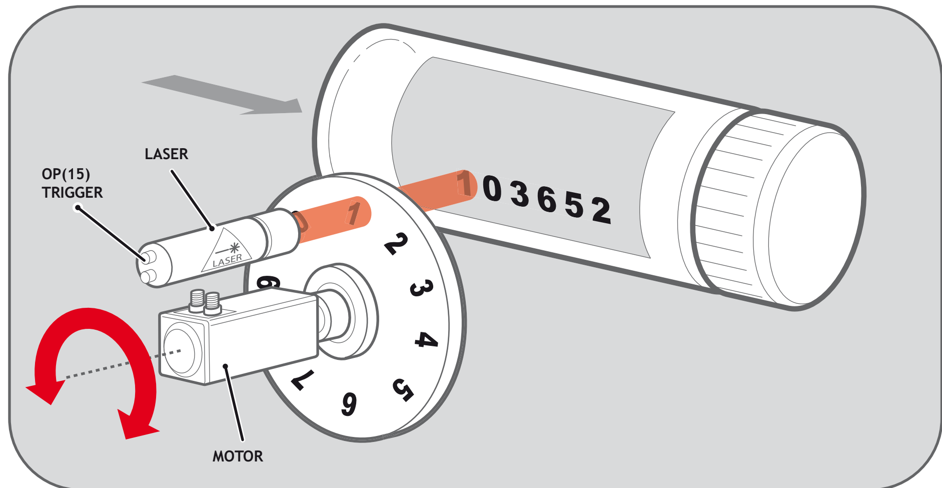
Example 2: A masked wheel is used to create a stencil for a laser to shine through for use in a printing system for the ten numerical digits. The required digits are transmitted through port 1 serial port to the controller as ASCII text.

The encoder used has 4000 edges per revolution and so must move 400 between each position. The cam table goes from 0 to 1, which means that the CAM multiplier needs to be a multiple of 400 to move between the positions.

The wheel is required to move to the pre-set positions every 0.25 seconds. The speed is set to 10000 edges/second, and we want the profile to be complete in 0.25 seconds. So multiplying the axis speed by the required completion time (10000 x 0.25) gives the distance parameter equals 2500.

```
GOSUB profile_gen
 WHILE IN(2)=ON
   WAIT UNTIL KEY#1              'Waits for character on port 1
   GET#1,k
   IF k>47 AND k<58 THEN         'check for valid ASCII character
```

```
        position=(k-48)*400        'convert to absolute position
        multiplier=position-offset 'calculate relative movement
        'check if it is shorter to move in reverse direction
        IF multiplier>2000 THEN
          multiplier=multiplier-4000
        ELSEIF multiplier<-2000 THEN
          multiplier=multiplier+4000
        ENDIF
        CAM(0,200,multiplier,2500)  'set the CAM movment
        WAIT IDLE
        OP(15,ON)                   'trigger the laser flash
        WA(20)
        OP(15,OFF)
        offset=(k-48)*400      'calculates current absolute position
      ENDIF
    WEND

profile_gen:
  num_p=201
  scale=1.0
  FOR p=0 TO num_p-1
    TABLE(p,((-SIN(PI*2*p/num_p)/(PI*2))+p/num_p)*scale)
  NEXT p
  RETURN
```



Example 3: A suction pick and place system must vary its speed depending on the load carried. The mechanism has a load cell which inputs to the controller on the analogue channel **(AIN)**.

The move profile is fixed, but the time taken to complete this move must be varied depending on the AIN. The **AIN** value varies from 100 to 800, which has to result in a move time of 1 to 8 seconds. If the speed is set to 10000 units per second and the required time is 1 to 8 seconds, then the distance parameter must range from 10000 to 80000. (distance = speed x time)

The return trip can be completed in 0.5 seconds and so the distance value of 5000 is fixed for the return movement. The Multiplier is set to -1 to reverse the motion.

```
GOSUB profile_gen         'loads the cam profile into the table
SPEED=10000:ACCEL=SPEED*1000:DECEL=SPEED*1000
WHILE IN(2)=ON
    OP(15,ON)             'turn on suction
    load=AIN(0)           'capture load value
    distance = 100*load   'calculate the distance parameter
    CAM(0,200,50,distance) 'move 50mm forward in time calculated
    WAIT IDLE
    OP(15,OFF)            'turn off suction
    WA(100)
    CAM(0,200,-50,5000)    'move back to pick up position
WEND

profile_gen:
  num_p=201
  scale=400               'set scale so that multiplier is in mm
  FOR p=0 TO num_p-1
    TABLE(p,((-SIN(PI*2*p/num_p)/(PI*2))+p/num_p)*scale)
  NEXT p
  RETURN
```

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 3 up to the maximum table size available. 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 positions relative to the start of the motion and are specified in encoder edges or steps. 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.

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 **link pos**)

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.

32 - Link is only active during a positive move on the link axis.

Note:

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

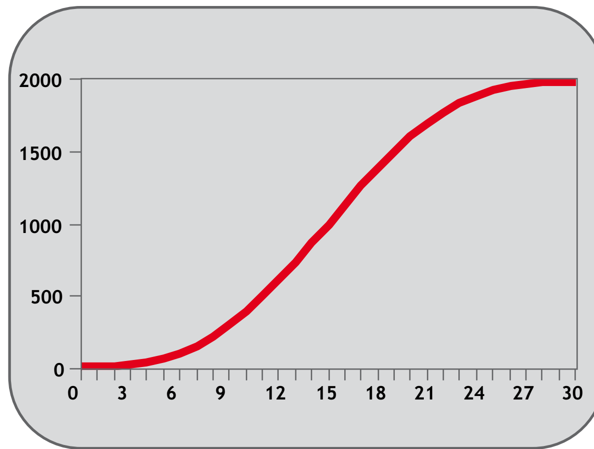
link pos: This parameter is the absolute position where the **CAMBOX** link is to be started when parameter 6 is set to 2.
Link pos cannot be at or within one servo_period's worth of movement of the **REP_DIST** position.

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.

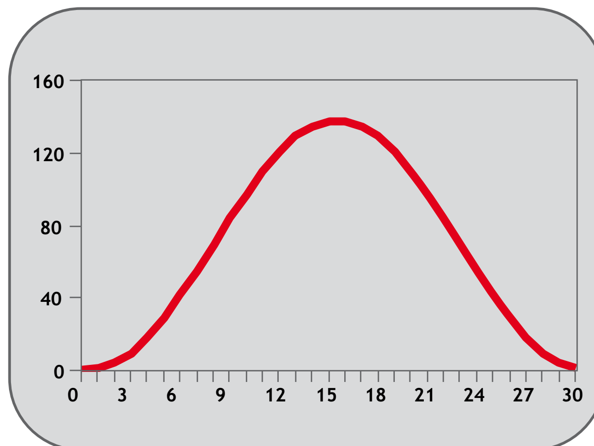
Example 1:

```
  ' 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
profile_gen:
  num_p=30
  scale=2000
  FOR p=0 TO num_p
    TABLE(p,((-SIN(PI*2*p/num_p)/(PI*2))+p/num_p)*scale)
  NEXT p
  RETURN
```



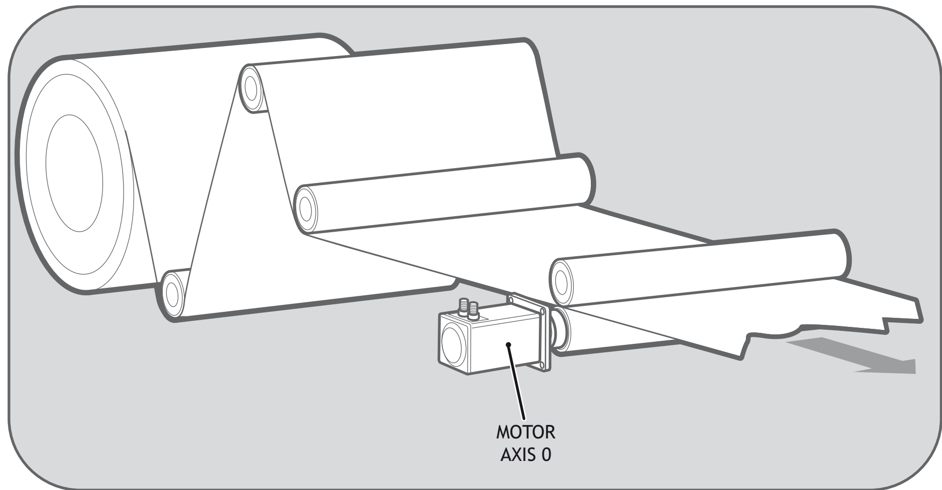
This graph plots **TABLE** contents against table array position. This corresponds to motor POSITION against link POSITION when called using **CAMBOX**. The **SPEED** of the motor will correspond to the derivative of the position curve above:

Speed Curve



Example 2: A pair of rollers feeds plastic film into a machine. The feed is syncronised to a master encoder and is activated when the master reaches a position held in the variable "start". This example uses the table points 0...30 generated in Example 1:

```
start=1000
FORWARD AXIS(1)
WHILE IN(2)=OFF
  CAMBOX(0,30,800,80,15,2,start)
  WA(10)
  WAIT UNTIL MTYPE=0 OR IN(2)=ON
WEND
CANCEL
CANCEL AXIS(1)
WAIT IDLE
```



Note:

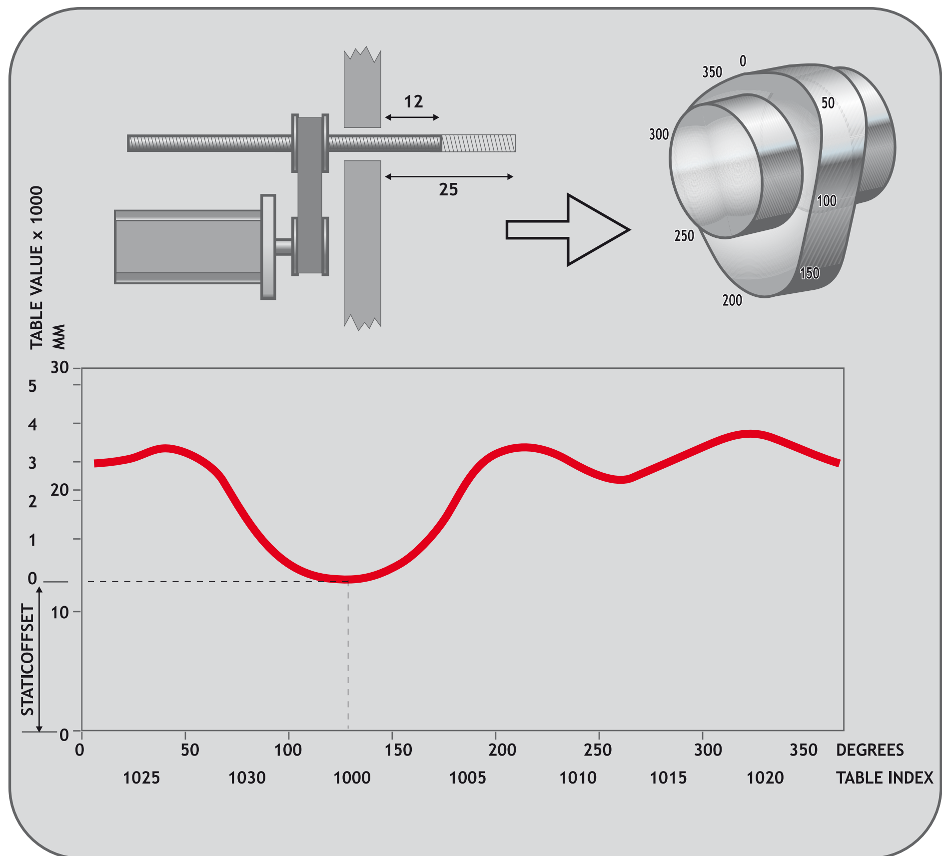
`0`	The start of the profile shape in the **TABLE**
`30`	The end of the profile shape in the **TABLE**
`800`	This scales the **TABLE** values. Each **CAMBOX** motion would therefore total 800*2000 encoder edges steps.
`80`	The distance on the product conveyor to link the motion to. The units for this 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..1035.

The table values represent the mechanical cam but are scaled to range from 0-4000

```
TABLE(1000,0,0,167,500,999,1665,2664,3330,3497,3497)
TABLE(1010,3164,2914,2830,2831,2997,3164,3596,3830,3996,3996)
TABLE(1020,3830,3497,3330,3164,3164,3164,3330,3467,3467,3164)
TABLE(1030,2831,1998,1166,666,333,0)

BASE(3)
MOVEABS(130)
WAIT IDLE
'start the continuously repeating cambox
CAMBOX(1000,1035,1,360,3,4) AXIS(0)
FORWARD             'start camshaft axis
WAIT UNTIL IN(2)=OFF
REP_OPTION = 2      'cancel repeating mode by setting bit 1
WAIT IDLE AXIS(0)   'waits for cam cycle to finish
CANCEL              'stop camshaft axis
WAIT IDLE
```

Note: The system software resets bit 1 of **REP_OPTION** after the repeating mode has been cancelled.



CAMBOX Pattern Mode:

Description: 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, next page, 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 4: A quilt stitching machine runs a feed cycle which stiches a plain pattern before starting a patterned stitch. The plain pattern should run for 1000 cycles prior to running a pattern continuously until requested to stop at the end of the pattern. The cam profile controls the motion of the needle bar between moves and the pattern table controls the distance of the move to make the pattern.

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

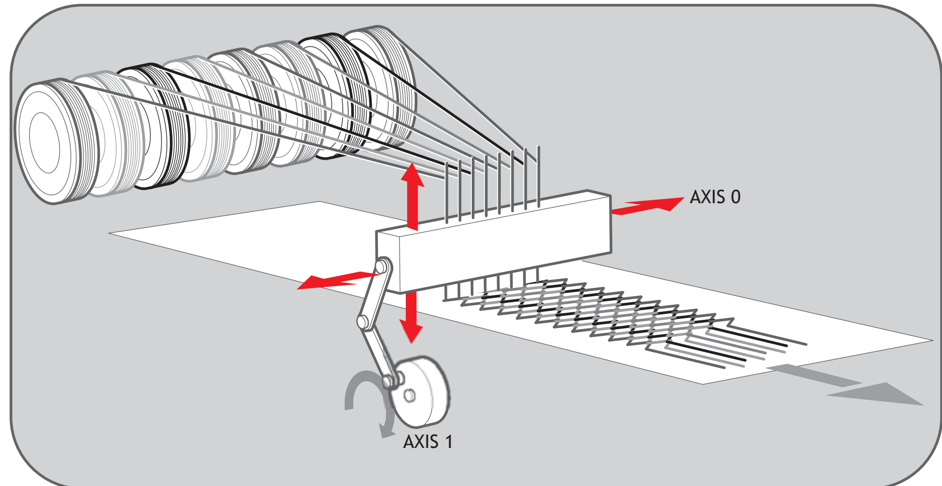
The running pattern sequence is held in **TABLE** values 1000..4999

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

Syntax: **CANCEL / CANCEL(1)**

Alternate Format: **CA**

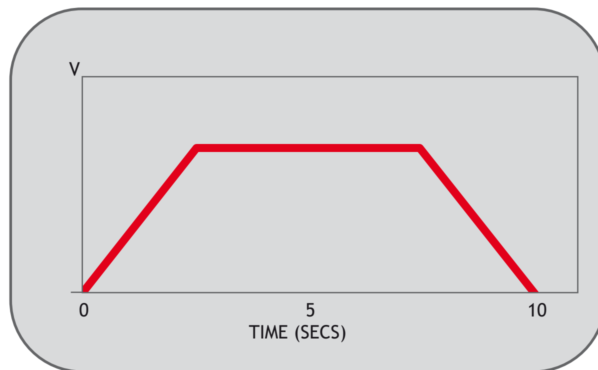
Description: Cancels a move on an axis or an interpolating axis group. Velocity profiled moves, for example; **FORWARD, REVERSE, MOVE, MOVEABS, MOVECIRC, MHELICAL, MOVE-MODIFY**, will be ramped down at the programmed deceleration rate then terminated. Other move types will be terminated immediately.

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 and the axis will not stop.

Example 1: Move the base axis forward at the progammed SPEED, wait for 10 seconds, then slow down and stop the axis at the programmed DECEL rate.

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

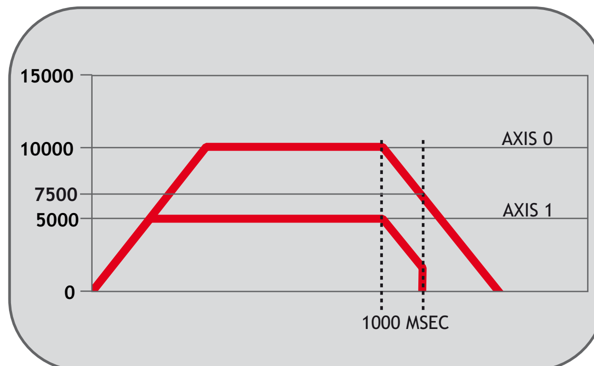


Example 2: A flying shear uses a sequence of **MOVELINKs** to make the base axis follow a reference encoder on axis 4. When the shear returns to the top position an input is triggered, this removes the buffered **MOVELINK** and replace with a decelrating **MOVELINK** to ramp down the slave (base) axis.

```
ref_axis = 4
REPEAT
  MOVELINK(100,100,0,0,ref_axis)
  WAIT LOADED   'make sure the NTYPE buffer is empty each time
UNTIL IN(5)=ON
CANCEL(1)       'cancel the movelink in the NTYPE buffer
MOVELINK(100,200,0,200,ref_axis) ' deceleration ramp
CANCEL         'cancel the main movelink, this starts the decel
```

Example 3: Two axes are connected with a ratio of 1:2. Axis 0 is cancelled after 1 second, then axis 1 is cancelled when the speed drops to a specified level. Following the first cancel axis 1 will decelerate at the **DECEL** rate. When axis 1's **CONNECT** is cancelled it will stop instantly.

```
BASE(0)
SPEED=10000
FORWARD
CONNECT(0.5,0) AXIS(1)
WA(1000)
CANCEL
WAIT UNTIL VP_SPEED<=7500
CANCEL AXIS(1)
```



See also: RAPIDSTOP.

CONNECT

Type: **Axis Command**

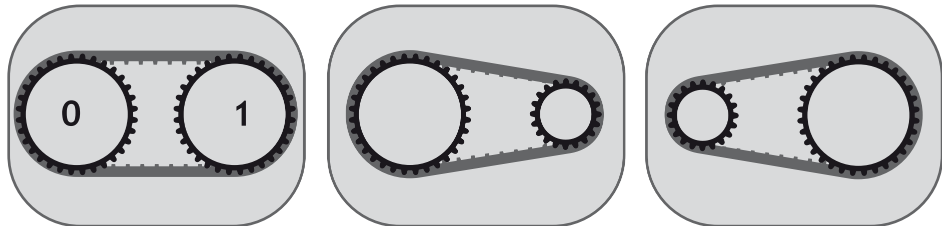
Syntax: **CONNECT(ratio , driving axis)**

Alternate Format: **CO(...)**

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

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.



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

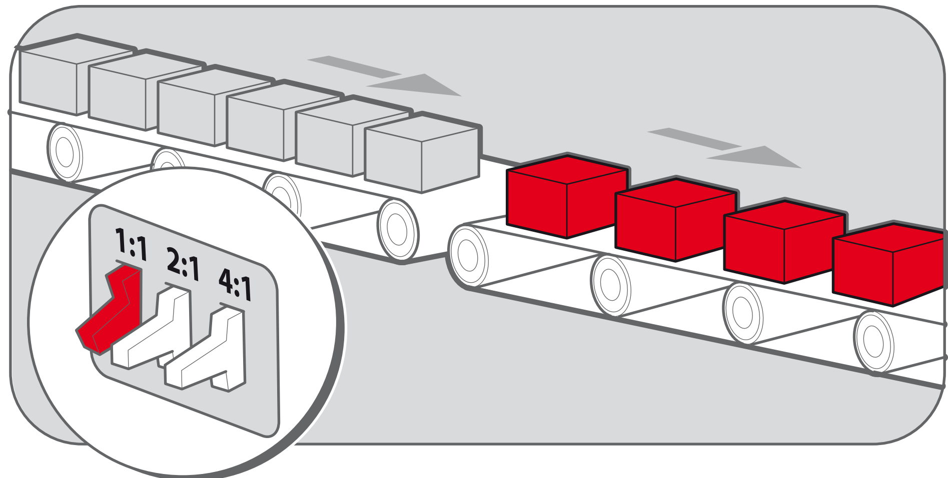
Note: To achieve an exact connection of fractional ratio's of values such as 1024/3072. The **MOVELINK** command can be used with the continuous repeat link option set to ON.

Example 1: 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. The reference encoder is connected to axis 1.

```
BASE(0)
SERVO=ON
CONNECT(0.25,1)
```

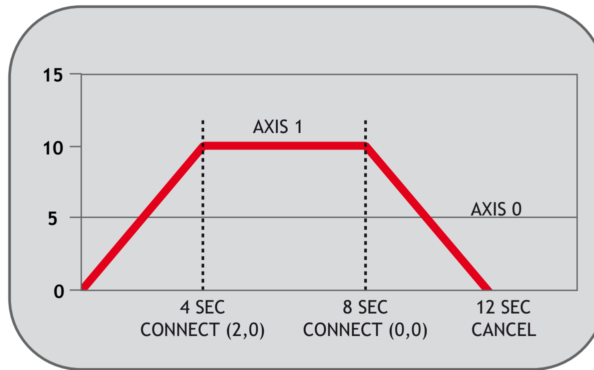
Example 2: A machine has an automatic feed on axis 1 which must move at a set ratio to axis 0. This ratio is selected using inputs 0-2 to select a particular "gear", this ratio can be updated every 100msec. Combinations of inputs will select intermediate gear ratios. For example 1 ON and 2 ON gives a ratio of 6:1.

```
BASE(1)
FORWARD AXIS(0)
WHILE IN(3)=ON
  WA(100)
  gear = IN(0,2)
  CONNECT(gear,0)
WEND
RAPIDSTOP        'cancel the FORWARD and the CONNECT
```



Example 3: Axis 0 is required to run a continuous forward, axis 1 must connect to this but without the step change in speed that would be caused by simply calling the **CONNECT**. **CLUTCH_RATE** is used along with an initial and final connect ratio of zero to get the required motion.

```
FORWARD AXIS(0)
BASE(1)
CONNECT(0,0)      'set intitial ratio to zero
CLUTCH_RATE=0.5   'set clutch rate
CONNECT(2,0)      'apply the required connect ratio
WA(8000)
CONNECT(0,0)      'apply zero ratio to disconnect
WA(4000)          'wait for deceleration to complete
CANCEL            'cancel connect
```



DATUM

Type: Command

Syntax: **DATUM(sequence no)**

Description: Performs one of 6 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.

DATUM(0) is a special case used for resetting the system after an axis critical error. It leaves the positions unchanged.

Parameter:

Seq.	Description
0	**DATUM(0)** clears the following error exceeded **FE_LIMIT** condition for ALL axes by setting these bits in **AXISSTATUS** to zero: BIT 1 Following Error Warning BIT 2 Remote Drive Comms Error BIT 3 Remote Drive Error BIT 8 Following Error Limit Exceeded BIT 11 Cancelling Move For stepper axes with position verification, the current measured position of ALL axes are set as demand position. FE is therefore set to zero. **DATUM(0)** must only be used after the **WDOG** is set to OFF, otherwise there will be unpredictable effects on the motion.
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.

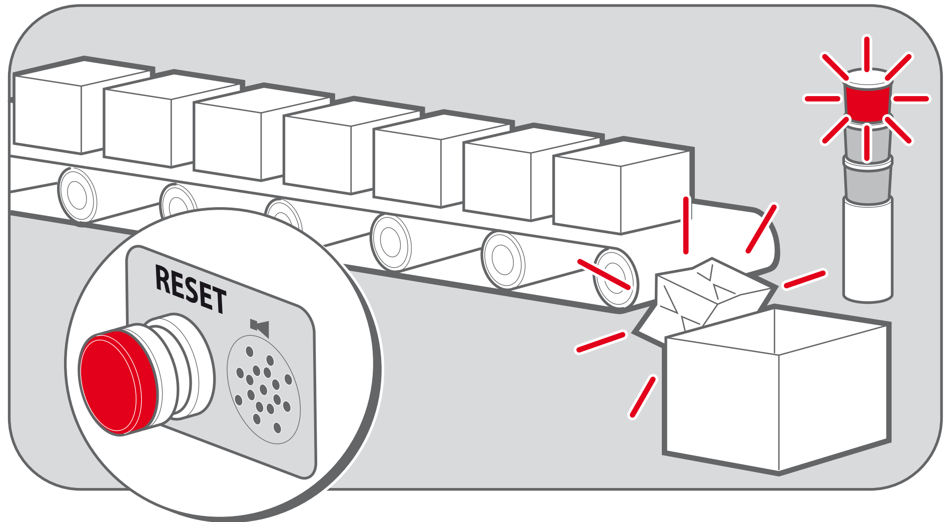
Seq.	Description
2	The axis moves at creep speed in reverse till the Z marker is encountered. The Demand position is then reset to zero and the Measured position corrected so as to maintain the following error.
3	The axis moves at the programmed speed forward until the datum switch is reached. The axis 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.
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 at creep speed 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 at creep speed 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.
7	Clear **AXISSTATUS** error bits for the **BASE** axis only. Otherwise the action is the same as **DATUM(0)**.

Note: The datuming input set with the **DATUM_IN** which 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 1: A production line is forced to stop if something jams the product belt, this causes a motion error. The obstacle has to be removed, then a reset switch is pressed to restart the line.

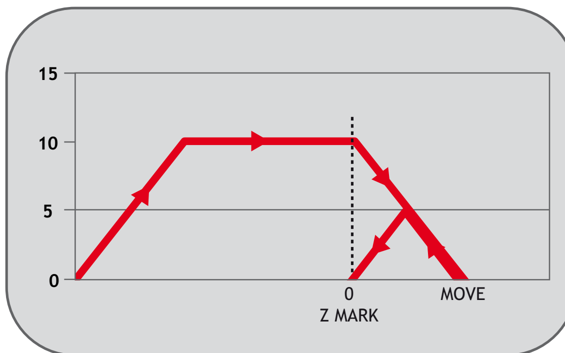
```
  FORWARD                'start production line
  WHILE IN(2)=ON
    IF MOTION_ERROR=0 THEN
       OP(8,ON)          'green light on; line is in motion
          ELSE
       OP(8, OFF)
      GOSUB error_correct
    ENDIF
  WEND
  CANCEL
  STOP

error_correct:
   REPEAT
```

```
    OP(10,ON)
    WA(250)
    OP(10,OFF)         'flash red light to show crash
    WA(250)
   UNTIL IN(1)=OFF
  DATUM(0)             'reset axis status errors
  SERVO=ON             'turn the servo back on
  WDOG=ON              'turn on the watchdog
  OP(9,ON)             'sound siren that line will restart
  WA(1000)
  OP(9,OFF)
  FORWARD              'restart motion
RETURN
```



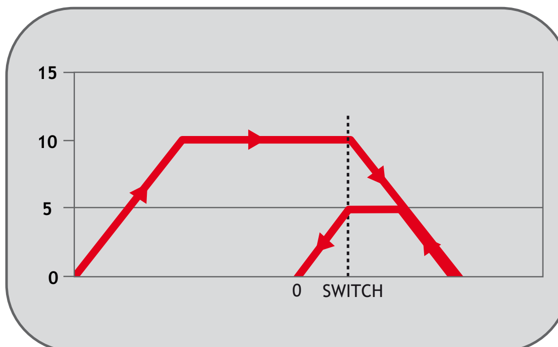
Example 2: An axis requires its position to be defined by the Z marker. This position should be set to zero and then the axis should move to this position. Using the datum 1 the zero point is set on the Z mark, but the axis starts to decelerate at this point so stops after the mark. A move is then used to bring it back to the Z position.

```
  SERVO=ON
  WDOG=ON
  CREEP=1000    'set the search speed
  SPEED=5000    'set the return speed
  DATUM(1)      'register on Z mark and sets this to datum
  WAIT IDLE
MOVEABS (0)    'moves to datum position
```



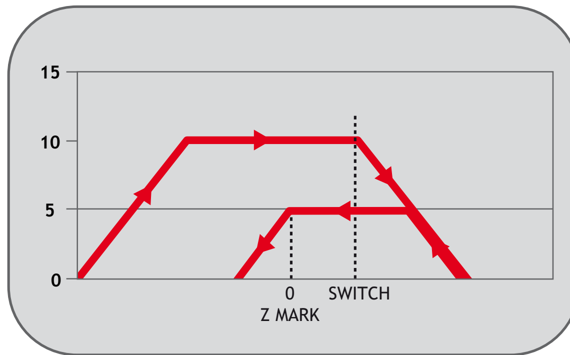
Example 3: A machine must home to its limit switch which is found at the rear of the travel before operation. This can be achieved through using **DATUM(4)** which moves in reverse to find the switch.

```
SERVO=ON
WDOG=ON
REV_IN=-1      'temporarily turn off the limit switch function
DATUM_IN=5     'sets input 5 for registration
SPEED=5000     'set speed, for quick location of limit switch
CREEP=500      'set creep speed for slow move to find edge of switch
DATUM(4)       'find "edge" at creep speed and stop
WAIT IDLE
DATUM_IN=-1
REV_IN=5       'restore input 5 as a limit switch again
```



Example 4: A similar machine to Example 3 must locate a home switch, which is at the forward end of travel, and then move backwards to the next Z marker and set this as the datum. This is done using **DATUM(5)** which moves forwards at speed to locate the switch, then reverses at creep to the Z marker. A final move is then needed, if required, as in Example 2 to move to the datum Z marker.

```
SERVO=ON
WDOG=ON
DATUM_IN=7   'sets input 7 as home switch
SPEED=5000   'set speed, for quick location of switch
CREEP=500    'set creep speed for slow move to find edge of switch
DATUM(5)     'start the homing sequence
WAIT IDLE
```



DEC

Type: Axis Command

Syntax: **DEC(rate)**

Description: Sets the deceleration rate for an axis. Different rates may be set for each axis. The **DEC** command is included to maintain compatibility with older controllers. Axis Parameter **DECEL** provides the same functionality and is the preferred method for setting the decleration rate.

Parameters: **rate:** The deceleration rate in UNITS/SEC/SEC.

Note: **ACC** sets both the acceleration and the deceleration rates to the same value. As **DEC** sets only the deceleration rate, you must use **DEC** after the **ACC** command in the program in order to make acceleration and deceleration rates different.

See Also: **ACCEL** and **DECEL** axis parameters.

Example 1: Initialising an axis to use different rates for acceleration and deceleration, then processing a move.

```
ACC(120)    'set accel and decel to 120 units/sec/sec
DEC(90)     'set deccel to 90 units/sec/sec
SPEED=14.5  'set programmed speed to 14.5 units/sec
MOVE(500)   'start a relative move with distance of 500
```

DEFPOS

Type: Function

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

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

Description: Defines the current position(s) as a new absolute value. The value pos# is placed in **DPOS**, while **MPOS** is adjusted to maintain the FE value. This function is completed after the next servo-cycle. **OFFPOS** is set non-zero when the **DEFPOS** begins execution and **OFFPOS** returns to 0 when the **DPOS** and **MPOS** have been updated. **DEFPOS** may be used at any time, even whilst a move is in progress, but its normal function is to set the position values of a group of axes which are stationary.

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.

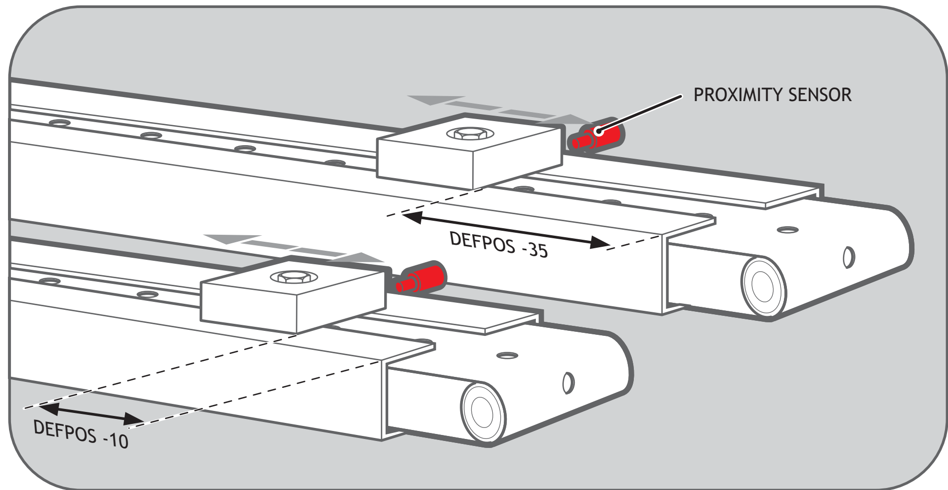
pos3: Abs. position to set on the next axis in **BASE** array in user units.

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

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

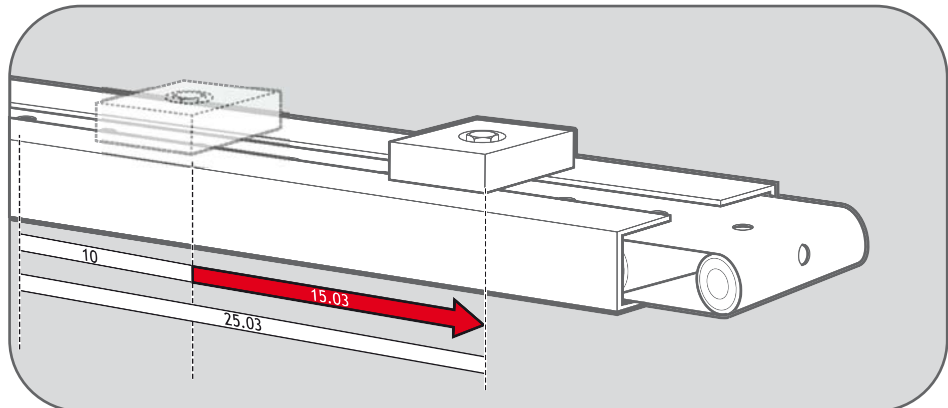
Example 1: After homing 2 axes, it is required to change the **DPOS** values so that the “home” positions are not zero, but some defined positions instead.

```
DATUM(5) AXIS(1)  'home both axes.  At the end of the DATUM
DATUM(4) AXIS(3)  'procedure, the positions will be 0,0.
WAIT IDLE AXIS(1)
WAIT IDLE AXIS(3)
BASE(1,3)         'set up the BASE array
DEFPOS(-10,-35)   'define positions of the axes to be -10 and -35
```



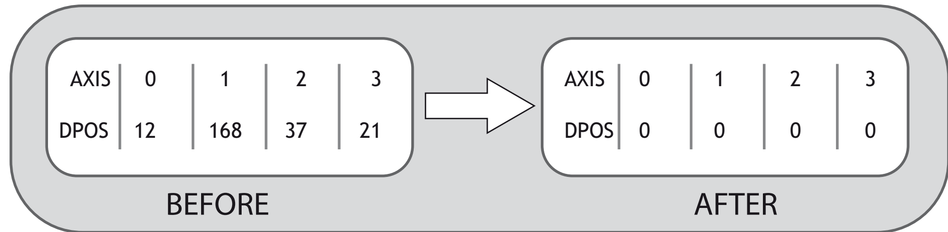
Example 2: Define the axis position to be 10, then start an absolute move, but make sure the axis has updated the position before loading the **MOVEABS**.

```
DEFPOS(10.0)
WAIT UNTIL OFFPOS=0' Ensures DEFPOS is complete before next line
MOVEABS(25.03)
```



Example 3: From the Motion Perfect terminal, quickly set the **DPOS** values of the first four axes to 0.

```
>>BASE(0)
>>DP(0,0,0,0)
>>
```



DISABLE_GROUP

Type: Function

Syntax: **DISABLE_GROUP(axis1 [,axis2[, axis3[, axis4.....]]])**

Description: Used to create a group of axes which will be disabled if there is a motion error in one or more of the group. After the group is created, when an error occurs all the axes in the group will have their **AXIS_ENABLE** set to OFF and SERVO set to OFF. Multiple groups can be made, although one axis cannot belong to more than one group.

All groupings will be cleared with the command **DISABLE_GROUP(-1)**.

Parameters:

axis1: Axis number of first axis in group.

axis2: Axis number of second axis in group.

axisN: Axis number of Nth axis in group.

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

Example 1: A machine has 2 functionally separate parts, which have their own emergency stop and operator protection guarding. If there is an error on one part of the machine, the other part can remain running while the cause of the error is removed and the axis group re-started. We need to set up 2 separate axis groupings.

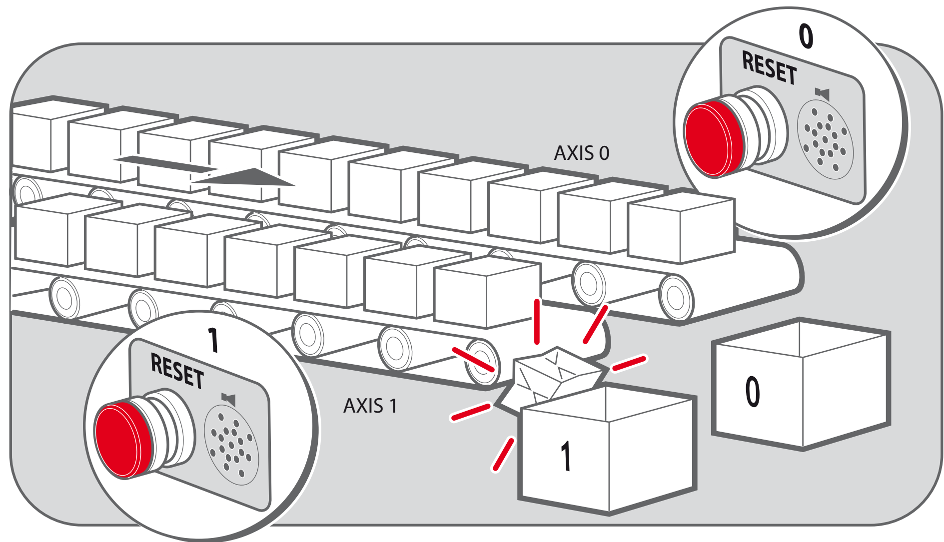
```
DISABLE_GROUP(-1)        'remove any previous axis groupings
DISABLE_GROUP(0,1,2,6)   'group axes 0 to 2 and 6
DISABLE_GROUP(3,4,5,7)   'group axes 3 to 5 and 7
```

```
WDOG=ON    'turn on the enable relay and the remote drive enable

FOR ax=0 TO 7
  AXIS_ENABLE AXIS(ax)=ON 'enable the 8 axes
  SERVO AXIS(ax)=ON     'start position loop servo for each axis
NEXT ax
```

Example 2: Two conveyors operated by the same *Motion Coordinator* are required to run independently so that if one has a “jam” it will not stop the second conveyor.

```
DISABLE_GROUP(0) 'put axis 0 in its own group
DISABLE_GROUP(1) 'put axis 1 in another group

GOSUB group_enable0
GOSUB group_enable1
WDOG=ON

FORWARD AXIS(0)
FORWARD AXIS(1)

WHILE TRUE
  IF AXIS_ENABLE AXIS(0)=0 THEN
    PRINT "motion error axis 0"
    reset_0_flag=1
  ENDIF
  IF AXIS_ENABLE AXIS(1)=0 THEN
    PRINT "motion error axis 1"
    reset_1_flag=1
  ENDIF
  IF reset_0_flag=1 AND IN(0)=ON THEN
    GOSUB group_enable0
    FORWARD AXIS(0)
    reset_0_flag=0
  ENDIF
  IF reset_1_flag=1 AND IN(1)=ON THEN
    GOSUB group_enable1
    FORWARD AXIS(1)
    reset_1_flag=0
  ENDIF
WEND

group_enable0:
  BASE(0)
  DATUM(7) ' clear motion error on axis 0
  WA(10)
```

```
  AXIS_ENABLE=ON
  SERVO=ON
RETURN
group_enable1:
  BASE(1)
  DATUM(7) ' clear motion error on axis 0
  WA(10)
  AXIS_ENABLE=ON
  SERVO=ON
RETURN
```



Example 3: One group of axes in a machine require resetting, without affecting the remaining axes, if a motion error occurs. This should be done manually by clearing the cause of the error, pressing a button to clear the controllers' error flags and re-enabling the motion.

```
  DISABLE_GROUP(-1)          'remove any previous axis groupings
  DISABLE_GROUP(0,1,2)       'group axes 0 to 2
  GOSUB group_enable         'enable the axes and clear errors
  WDOG=ON

  SPEED=1000
  FORWARD
```

```
WHILE IN(2)=ON
    'check axis 0, but all axes in the group will disable together
    IF AXIS_ENABLE =0 THEN
      PRINT "Motion error in group 0"
      PRINT "Press input 0 to reset"
      IF IN(0)=0 THEN       'checks if reset button is pressed
        GOSUB group_enable  'clear errors and enable axis
        FORWARD             'restarts the motion
      ENDIF
    ENDIF
  WEND
  STOP                      'stop program running into sub routine

group_enable:               'Clear group errors and enable axes
  DATUM(0)                  'clear any motion errors
  WA(10)
  FOR axis_no=0 TO 2
    AXIS_ENABLE AXIS(axis_no)=ON  'enable axes
    SERVO AXIS(axis_no)=ON        'start position loop servo
  NEXT axis_no
  RETURN
```

See Also: **AXIS_ENABLE** for enabling remote axes.

Note: For use with SERCOS and MECHATROLINK only.

ENCODER_RATIO

Type: Function

Syntax: **ENCODER_RATIO(mpos_count, input_count)**

Description: This command allows the incoming encoder count to be scaled by a non integer ratio, using the following ratio;

MPOS = (mpos_count / input_count) x encoder_edges_input

ENCODER_RATIO affects the number of edges within the servo loop at a low level and it will be necessary to change the position loop gains to maintain perfomance and stablilty. Unlike the **UNITS** parameter, which only affects the scaling seen by the user programs, **ENCODER_RATIO** affects all motion commands including **MOVECIRC** and **CAMBOX**.

Parameters:

mpos_count :	A number between 0 and 16777215 which defines the numerator of the above function.
input_count:	A number between 0 and 16777215 which defines the denominator of the above function.

Note 1: Large ratios should be avoided as they will lead to either loss of resolution or much reduced smoothness in the motion. The actual physical encoder count is the basic resolution of the axis and use of this command may reduce the ability of the Motion Coordinator to accurately achieve all positions.

Note 2: **ENCODER_RATIO** does not replace **UNITS**. Only use **ENCODER_RATIO** where absolutely necessary. **PP_STEP** and **ENCODER_RATIO** cannot be used at the same time on the same axis.

Example 1: A rotary table has a servo motor connected directly to its centre of rotation. An encoder is mounted to the rear of the servo motor and returns a value of 8192 counts per rev. The application requires the table to be calibrated in degrees so that each degree is an integer number of counts.

```
‘ 7200 is a value close to the encoder resolution, but can be
divided
‘ by an integer to give degrees.  (7200 / 20 = 360)
ENCODER_RATIO(7200,8192)
UNITS = 20 ‘ axis calibrated in degrees, resolution = 0.05 deg.
```

Example 2: An X-Y system has 2 different gearboxes on its vertical and horizontal axes. The software needs to use interpolated moves, including **MOVECIRC** and MUST therefore have **UNITS** on the 2 axes set the same. Axis 3 (X) is 409 counts per mm and axis 4 (Y) has 560 counts per mm. So as to use the maximum resolution available, set both axes to be 560 counts per mm with the **ENCODER_RATIO** command.

```
ENCODER_RATIO(560,409) AXIS(3) 'axis 3 is now 560 counts/mm
UNITS AXIS(3) = 56 'X axis calibrated in mm x 10
UNTIS AXIS(4) = 56 'Y axis calibrated in mm x 10
MOVECIRC(200,100,100,0,1) 'move axes in a semicircle
```

FORWARD

Type: Axis Command

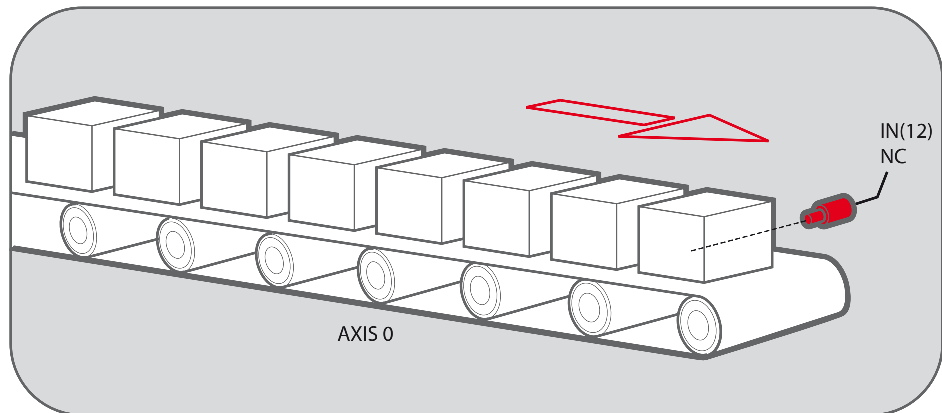
Syntax: **FORWARD**

Alternate Format: **FO**

Description: Sets continuous forward movement. The axis accelerates at the programmed **ACCEL** rate and continues moving at the **SPEED** value until either a **CANEL** or **RAPIDSTOP** command are encountered. It then declerates to a stop at the programmed **DECEL** rate.
If the axis reaches either the forward limit switch or forward soft limit, the **FORWARD** will be cancelled and the axis will decelerate to a stop.

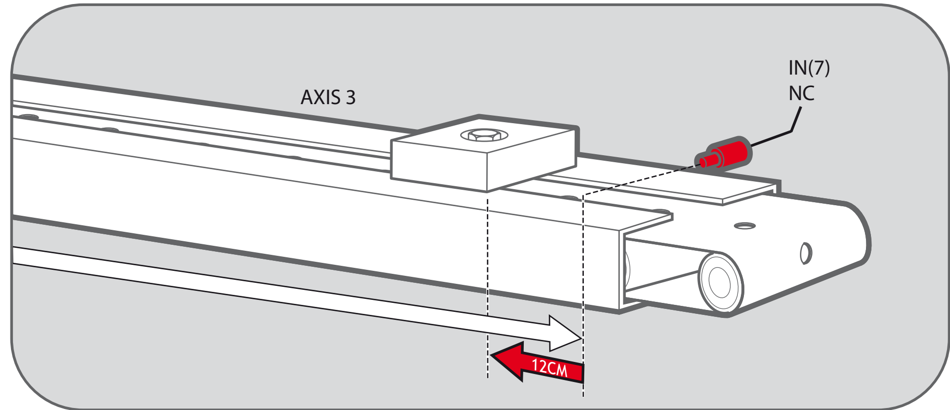
Example 1: Run an axis forwards. When an input signal is detected on input 12, bring the axis to a stop.

```
FORWARD
' wait for stop signal
WAIT UNTIL IN(12)=ON
CANCEL
WAIT IDLE
```



Example 2: Move an axis forwards until it hits the end limit switch, then move it in the reverse direction for 25 cm.

```
BASE(3)
FWD_IN=7 'limit switch connected to input 7
FORWARD
WAIT IDLE ' wait for motion to stop on the switch
MOVE(-25.0)
WAIT IDLE
```



Example 3: A machine that applies lids to cartons uses a simulated line shaft. This example sets up a virtual axis running forward, this is to simulate the line shaft. Axis 0 is then CONNECTed to this to run the conveyor. Axis 1 controls a vacuum roller that feeds the lids on to the cartons using the **MOVELINK** control.

```
BASE(4)
ATYPE=0          'Set axis 4 to virtual axis
REP_OPTION=1
SERVO=ON
FORWARD          'starts line shaft
BASE(0)
CONNECT(-1,4)    'Connects base 0 to virtual axis in reverse
WHILE IN(2)=ON
  BASE(1)
  'Links axis 1 to the shaft in reverse direction
  MOVELINK(-4000,2000,0,0,4,2,1000)
  WAIT IDLE
WEND
RAPIDSTOP
```

MHELICAL

Type: Motion Command.

Syntax: **MHELICAL(end1,end2,centre1,centre2,direction,distance3,[mode])**

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. End1 and centre1 are on the current **BASE** axis. End2 and centre2 are on the second axis. The first 4 distance parameters are scaled according to the current unit conversion factor for the **BASE** axis. The sixth parameter uses its own axis units.

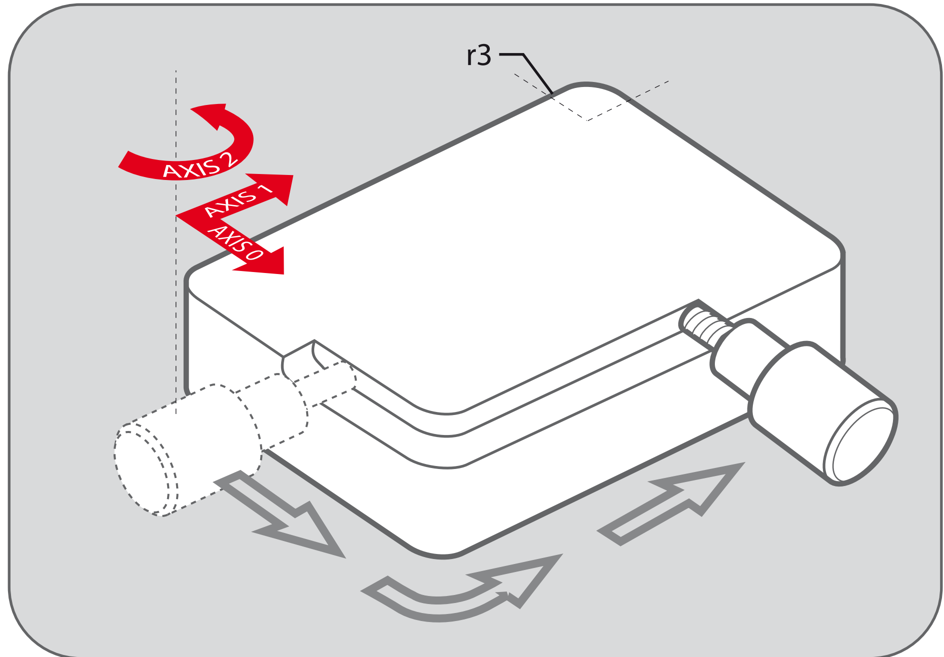
Parameters:

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 1 or 0. See **MOVECIRC**.
distance3:	The distance to move on the third axis in the **BASE** array axis in user units
mode:	0 = Interpolate the 3rd axis with the main 2 axes when calcualting path speed. (True helical path) 1 = Interpolate only the first 2 axes for path speed, but move the 3rd axis in coordination with the other 2 axes. (Circular path with following 3rd axis)

Example1: The command sequence follows a rounded rectangle path with axis 1 and 2. Axis 3 is the tool rotation so that the tool is always perpendicular to the product. The UNITS for axis 3 are set such that the axis is calibrated in degrees.

```
REP_DIST AXIS(3)=360
REP_OPTION AXIS(3)=ON
                   'all 3 axes must be homed before starting
MERGE=ON
MOVEABS(360) AXIS(3) 'point axis 3 in correct starting direction
WAIT IDLE AXIS(3)
MOVE(0,12)
```

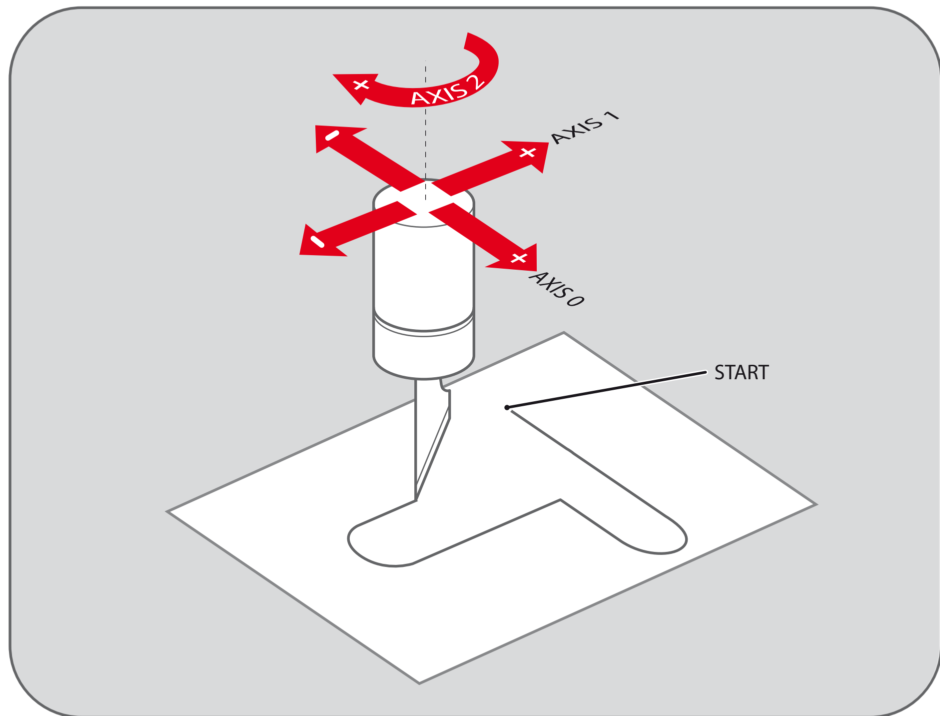
```
MHELICAL(3,3,3,0,1,90)
MOVE(16,0)
MHELICAL(3,-3,0,-3,1,90)
MOVE(0,-6)
MHELICAL(-3,-3,-3,0,1,90)
MOVE(-2,0)
MHELICAL(-3,3,0,3,1,90)
```



Exapmle 2: A PVC cutter uses 2 axis similar to a xy plotter, a third axis is used to control the cutting angle of the knife. To keep the resultant cutting speed for the x and y axis the same when cutting curves, mode 1 is applied to the helical command.

```
BASE(0,1,2) : MERGE=ON 'merge moves into one continuous movement
MOVE(50,0)
MHELICAL(0,-6,0,-3,1,180,1)
MOVE(-22,0)
WAIT IDLE
MOVE(-90) AXIS(2)    'rotate the knife after stopping at corner
WAIT IDLE AXIS(2)
```

```
MOVE(0,-50)
MHELICAL(-6,0,-3,0,1,180,1)
MOVE(0,50)
WAIT IDLE           'pause again to rotate the knife
MOVE(-90) AXIS(2)
WAIT IDLE AXIS(2)
MOVE(-22,0)
MHELICAL(0,6,0,3,1,180,1)
WAIT IDLE
```



MHELICALSP

Type: Motion Command.

Only available in system software versions where "LookAhead" is enabled.

Syntax: **MHELICALSP(end1,end2,centre1,centre2,direction,distance3,[mode])**

Description: Performs a helical move the same as **MHELICAL** and additionally allows vector speed to be changed when using multiple moves in the look-ahead buffer. Uses additional axis parameters **FORCE_SPEED** and **ENDMOVE_SPEED**.

Example: In a series of buffered moves using the look ahead buffer with **MERGE=ON** a helical move is required where the incoming vector speed is 40 units/second and the finishing vector speed is 20 units/second.

```
FORCE_SPEED=40
ENDMOVE_SPEED=20
MHELICALSP(100,100,0,100,1,100)
```

For more information see **MHELICAL**.

MOVE

Type: Motion Command

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

Alternate Format: **MO()**

Description: Incremental move. One axis or multiple axes move at the programmed speed and acceleration for a distance specified as an increment from the end of the last specified move. The first parameter in the list is sent to the **BASE** axis, the second to the next axis in the **BASE** array, and so on.

In the multi-axis form, the speed and acceleration employed for the movement are taken from the first axis in the **BASE** group. The speeds of each axis are controlled so as to make the resulting vector of the movement run at the **SPEED** setting.

Uninterpolated, unsynchronised multi-axis motion can be achieved by simply placing **MOVE** commands on each axis independently. If needed, the target axis for an individual **MOVE** can be specified using the **AXIS()** command. This overrides the **BASE** axis setting for one **MOVE** only.

The distance values specified are scaled using the unit conversion factor axis param-

eter; **UNITS**. Therefore if, for example, an axis has 400 encoder edges/mm and **UNITS** for that axis are 400, the command **MOVE(12.5)** would move 12.5 mm. When **MERGE** is set to **ON**, individual moves in the same axis group are merged together to make a continuous path movement.

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 maximum number of parameters is the number of axes on the controller

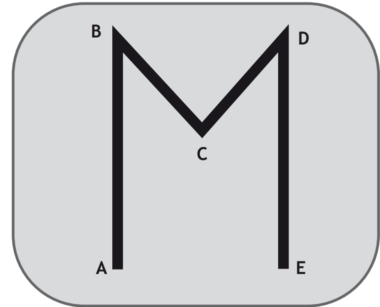
Example 1: A system is working with a unit conversion factor of 1 and has a 1000 line encoder. Note that a 1000 line encoder gives 4000 edges/turn.

```
MOVE(40000) ' move 10 turns on the motor.
```

Example 2: Axes 3, 4 and 5 are to move independently (without interpolation). Each axis will move at its own programmed **SPEED, ACCEL and DECEL** etc.

```
'setup axis speed and enable
BASE(3)
SPEED=5000
ACCEL=100000
DECEL=150000
SERVO=ON
BASE(4)
SPEED=5000
ACCEL=150000
DECEL=560000
SERVO=ON
BASE(5)
SPEED=2000
ACCEL=320000
DECEL=352000
SERVO=ON
WDOG=ON
MOVE(10) AXIS(5)     'start moves
MOVE(10) AXIS(4)
MOVE(10) AXIS(3)
WAIT IDLE AXIS(5)    'wait for moves to finish
WAIT IDLE AXIS(4)
WAIT IDLE AXIS(3)
```

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 regardless of the plot origin. The command subroutine for the letter 'M' might be:




```
write_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
  RETURN
```

MOVEABS

Type: Motion Command.

Syntax: **MOVEABS(position1[, position2[, position3[, position4...]]])**

Alternate Format: **MA()**

Description: Absolute position move. Move one axis or multiple axes to position(s) referenced with respect to the zero (home) 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, and so on.

In the multi-axis form, the speed, acceleration and deceleration employed for the movement are taken from the first axis in the **BASE** group. The speeds of each axis are controlled so as to make the resulting vector of the movement run at the **SPEED** setting.

Uninterpolated, unsynchronised multi-axis motion can be achieved by simply placing **MOVEABS** commands on each axis independently. If needed, the target axis for an individual **MOVEABS** can be specified using the **AXIS()** command. This overrides the **BASE** axis setting for one **MOVEABS** only.

The values specified are scaled using the unit conversion factor axis parameter; **UNITS**. Therefore if, for example, an axis has 400 encoder edges/mm the **UNITS** for that axis is 400. The command **MOVEABS(6)**would then move to a position 6 mm from the zero position. When **MERGE** is set to ON, absolute and relative moves are merged together to make a continuous path movement.

Parameters:

position1: position to move to on base axis.

position2: position to move to on next axis in BASE array.

position3: position to move to on next axis in BASE array.

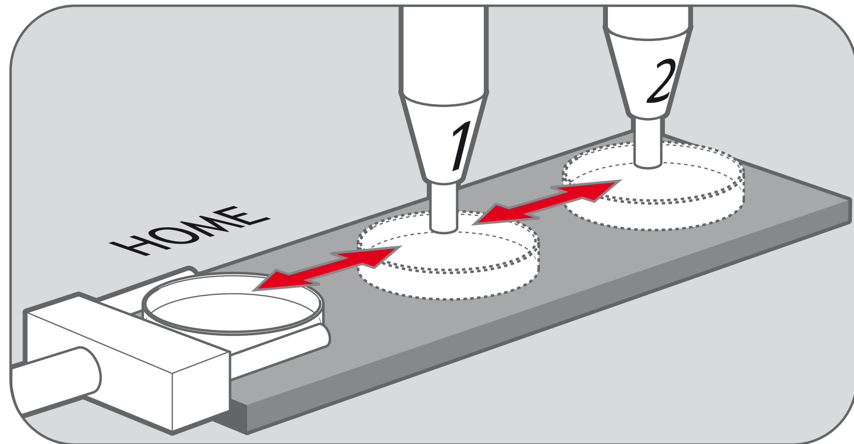
position4: position to move to on next axis in BASE array

Note1: The **MOVEABS** command can interpolate up to the full number of axes available on the controller.

Note2: The position of the axes' zero(home) positions can be changed by the commands: **OFFPOS, DEFPOS, REP_DIST, REP_OPTION**, and **DATUM**.

Example 1: A machine must move to one of 3 positions depending on the selection made by 2 switches. The options are home, position 1 and position 2 where both switches are off, first switch on and second switch on respectively. Position 2 has priority over position 1.

```
'define absolute positions
home=1000
position_1=2000
position_2=3000

WHILE IN(run_switch)=ON
  IF IN(6)=ON THEN          'switch 6 selects position 2
    MOVEABS(position_2)
    WAIT IDLE
  ELSEIF IN(7)=ON THEN      'switch 7 selects position 1
    MOVEABS(position_1)
    WAIT IDLE
  ELSE
    MOVEABS(home)
    WAIT IDLE
  ENDIF
WEND
```

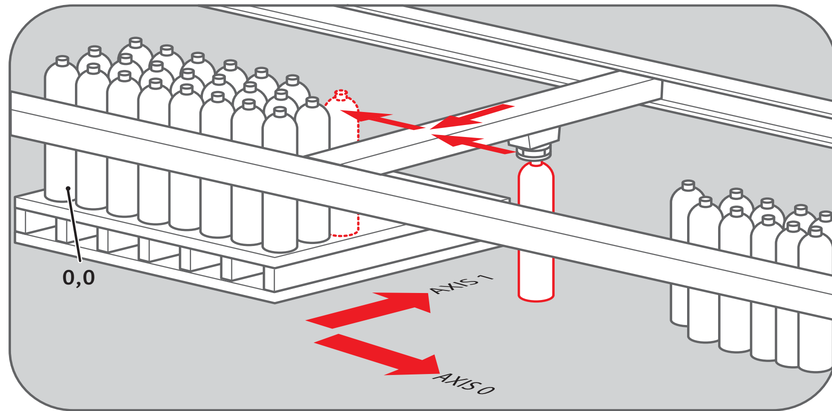


Example 2: An X-Y plotter has a pen carousel whose position is fixed relative to the plotter absolute zero position. To change pen an absolute move to the carousel position will find the target irrespective of the plot position when commanded.

```
MOVEABS(28.5,350) 'move to just outside the pen holder area
WAIT IDLE
SPEED = pen_pickup_speed
MOVEABS(20.5,350) 'move in to pick up the pen
```

Example 3: A pallet consists of a 6 by 8 grid in which gas canisters are inserted 185mm 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
    MOVEABS(-340,-516.5)          'move to pick-up point
    WAIT IDLE
    GOSUB pick                    'call pick up subroutine
    PRINT "Move to Position: ";x*6+y+1
    MOVEABS(x*185,y*185)          'move to position in grid
    WAIT IDLE
    GOSUB place                   'call place down subroutine
    NEXT y
NEXT x
```



MOVEABSSP

Type: Motion Command.

Only available in system software versions where “LookAhead” is enabled.

Syntax: **MOVEABSSP(position1[, position2[, position3[, position4]]])**

Description: Works as **MOVEABS** and additionally allows vector speed to be changed when using multiple moves in the look ahead buffer when **MERGE=ON**, using additional parameters **FORCE_SPEED** and **ENDMOVE_SPEED**.

Parameters:

position1:	position to move to on base axis.
position2:	position to move to on next axis in BASE array.
position3:	position to move to on next axis in BASE array.
position4:	position to move to on next axis in BASE array

Note: *Absolute moves are converted to incremental moves as they enter the buffer. This is essential as the vector length is required to calculate the start of deceleration. It should be noted that if any move in the buffer is cancelled by the programmer, the absolute position will not be achieved.*

Example 1: In a series of buffered moves using the look ahead buffer with **MERGE=ON**, an absolute move is required where the incoming vector speed is 40units/second and the finishing vector speed is 20 units/second.

```
FORCE_SPEED=40
ENDMOVE_SPEED=20
MOVEABSSP(100,100)
```

Only Available in Look-Ahead mode.
For more information see **MOVEABS**.

MOVECIRC

Type: Motion Command.

Syntax: **MOVECIRC(end1, end2, centre1, centre2, direction)**

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 relative to 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 next axis in the **BASE** array. The first 4 distance parameters are scaled according to the current unit conversion factor for the **BASE** axis.

Parameters:

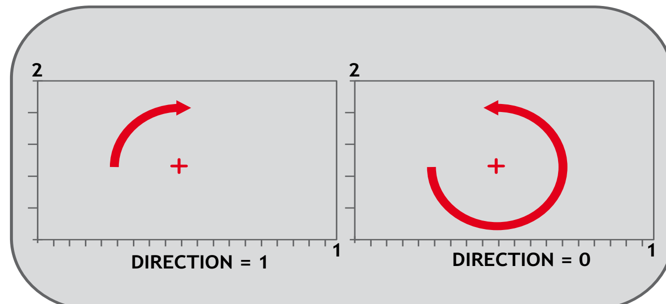
end1: position on BASE axis to finish at.

end2: position on next axis in BASE array to finish at.

centre1: position on BASE about which to move.

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

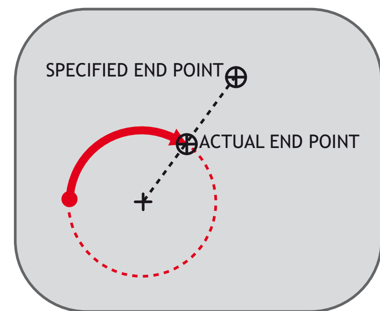
direction: The “direction” is a software switch which determines whether the arc is interpolated in a clockwise or anti- clockwise direction.



Note 1: *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 using* **ENCODER_RATIO** *or* **STEP_RATIO.**

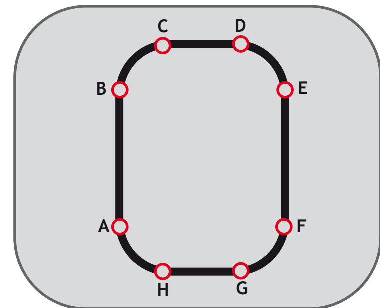
Note 2: *If the end point specified is not on the circular arc. The arc will end at the angle specified by a line between the centre and the end point.*

Note 3: *Neither axis may cross the set absolute repeat distance (***REP_DIST***) during a* **MOVECIRC***. Doing so may cause one or both axes to jump or for their* **FE** *value to exceed* **FE_LIMIT***.*



Example 1: The command sequence to plot the letter '0' might be:

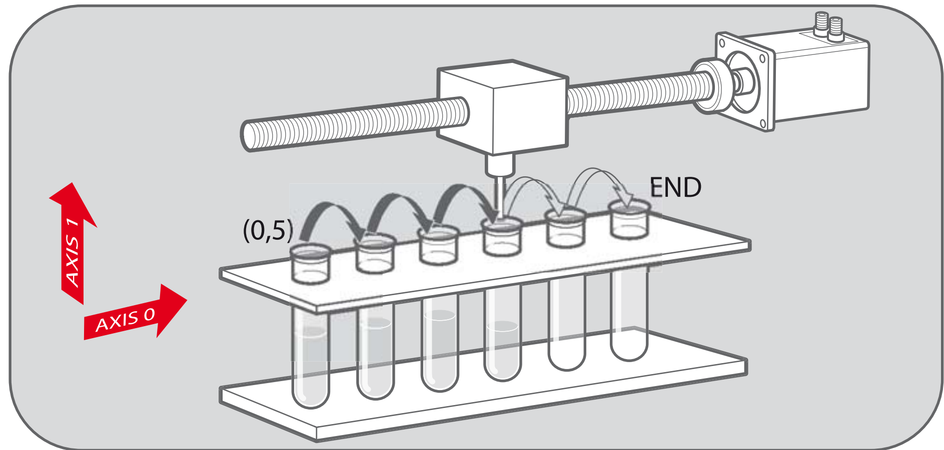
```
MOVE(0,6)              'move A -> B
MOVECIRC(3,3,3,0,1)    ' move B -> C
MOVE(2,0)              'move C -> D
MOVECIRC(3,-3,0,-3,1)  ' move D -> E
MOVE(0,-6)             'move E -> F
MOVECIRC(-3,-3,-3,0,1)' move F -> G
MOVE(-2,0)             'move G -> H
MOVECIRC(-3,3,0,3,1)   ' move H -> A
```



Example 2: A machine is required to drop chemicals into test tubes. The nozzle can move up and down as well as along its rail. The most efficient motion is for the nozzle to move in an arc between the test tubes.

```
BASE(0,1)
MOVEABS(0,5)                'move to position above first tube
MOVEABS(0,0)                'lower for first drop
WAIT IDLE
OP(15,ON)                   'apply dropper
```

```
WA(20)
OP(15,OFF)
FOR x=0 TO 5
  MOVECIRC(5,0,2.5,0,1)  'arc between the test tubes
  WAIT IDLE
  OP(15,ON)              'Apply dropper
  WA(20)
  OP(15,OFF)
NEXT x
MOVECIRC(5,5,5,0,1)     'move to rest position)
```



MOVECIRCSP

Type: Motion Command.

Only available in system software versions where "LookAhead" is enabled.

Syntax: **MOVECIRCSP(end1, end2, centre1, centre2, direction)**

Description: Works as **MOVECIRC** and additionally allows vector speed to be changed when using multiple moves in the look ahead buffer when **MERGE=ON**, using additional parameters **FORCE_SPEED** and **ENDMOVE_SPEED**.

Example 1: In a series of buffered moves using the look ahead buffer with **MERGE=ON**, a circular move is required where the incoming vector speed is 40units/second and the finishing vector speed is 20 units/second.

```
FORCE_SPEED=40
ENDMOVE_SPEED=20
```

```
MOVECIRCSP(100,100,0,100,1)
```

Note: Only available in Look-Ahead version.
For more information see **MOVECIRC**.

MOVELINK

Type: Motion Command.

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

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 a value between 0 and the 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 start`** parameter)

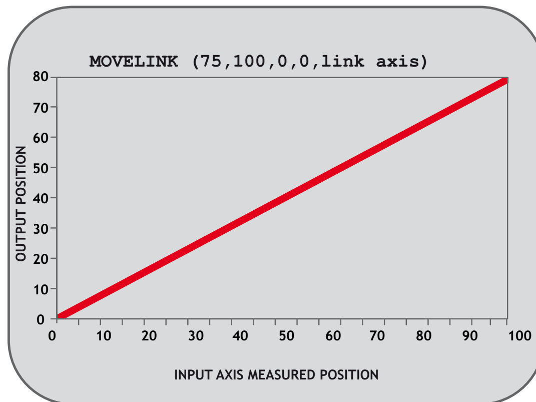
4 **`MOVELINK`** repeats automatically and bi-directional when this bit is set. (This mode can be cleared by setting bit 1 of the **`REP_OPTION`** axis parameter)

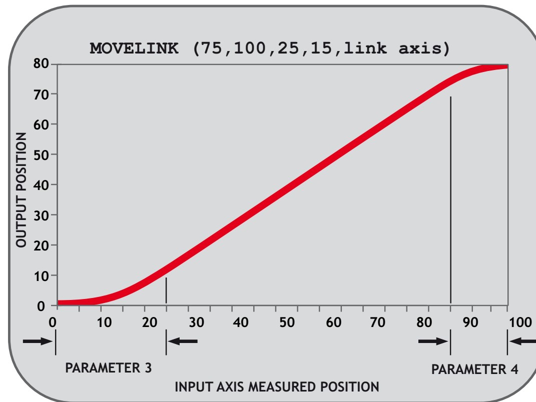
32 Link is only active during positive moves on the link axis.

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

Note 1: The command uses the **`BASE()`** and **`AXIS()`**, and unit conversion factors in a similar way to other move commands.

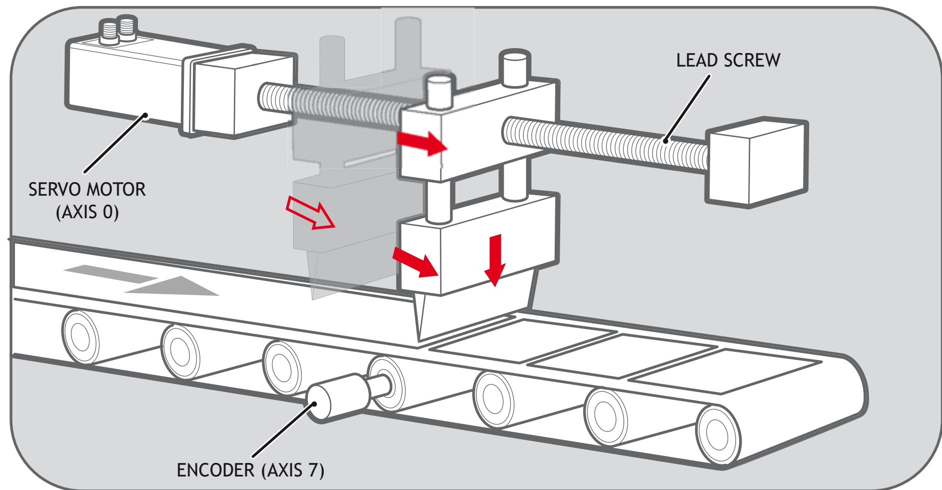
Note 2: The "link" axis may move in either direction to drive the output motion. The link distances specified are always positive.





Example 1: A flying shear cuts a long sheet of paper into cards every 160 m whilst moving at the speed of the material. 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)

```
WHILE IN(2)=ON
  MOVELINK(0,150,0,0,7) ' dwell (no movement) for 150m
  MOVELINK(0.3,0.6,0.6,0,7) ' accelerate to paper speed
  MOVELINK(0.7,1.0,0,0.6,7) ' track the paper then decelerate
  WAIT LOADED ' wait until acceleration movelink is finished
  OP(8,ON)    ' activate cutter
  MOVELINK(-1.0,8.4,0.5,0.5,7) ' retract cutter back to start
  WAIT LOADED
  OP(8,OFF)      ' deactivate cutter at end of outward stroke
WEND
```



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, moves at the same speed then decelerates to a stop within the 1m stroke. This movement is specified using two separate **MOVELINK** commands. This allows the program to wait for the next move buffer to be clear, **NTYPE=0**, which 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(0.3,0.6,0.6,0,1)' 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(0.4,0.4,0,0,1)' all constant speed
```

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

```
MOVELINK(0.3,0.6,0,0.6,1)' all decel
```

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

```
MOVELINK(1,1.6,0.6,0.6,1)' Same as 3 moves above
```

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

```
MOVELINK(0.3,0.6,0.6,0,1)
MOVELINK(0.7,1.0,0,0.6,1)
```

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

Example 2: Exact Ratio Gearbox

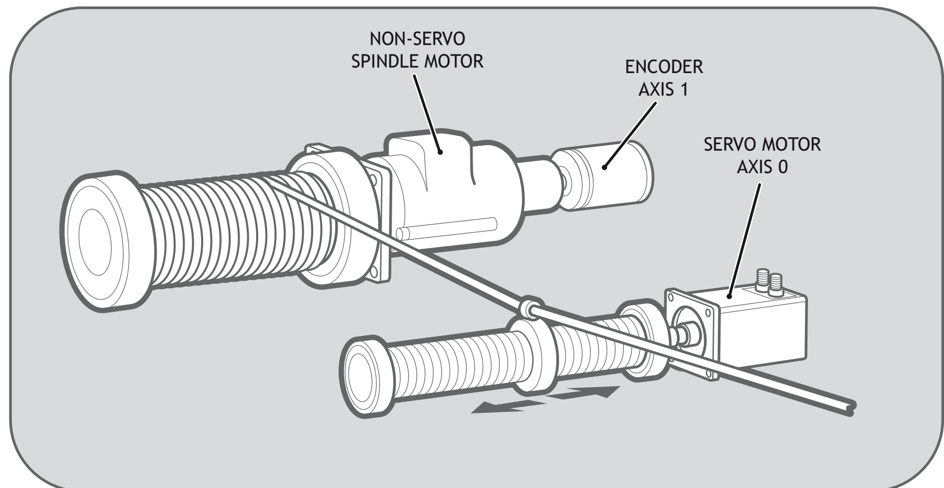
MOVELINK can be used to create an exact ratio gearbox between two axes. Suppose it is required to create gearbox link of 4000/3072. This ratio is inexact (1.30208333) and if entered into a **CONNECT** command the axes will slowly creep out of synchronisation. Setting the "link option" to 4 allows a continuously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

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

Example 3: 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(50,25,0,0,linkax)**. 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,linkax)**.

```
OP(motor,ON)   '- Switch spindle motor on
FOR layer=1 TO 10
  MOVELINK(50,25,0,0,1)
  MOVELINK(-50,25,0,0,1)
NEXT layer
WAIT IDLE
OP(motor,OFF)
```



MOVEMODIFY

Type: Axis Command.

Syntax: **MOVEMODIFY(absolute position)**

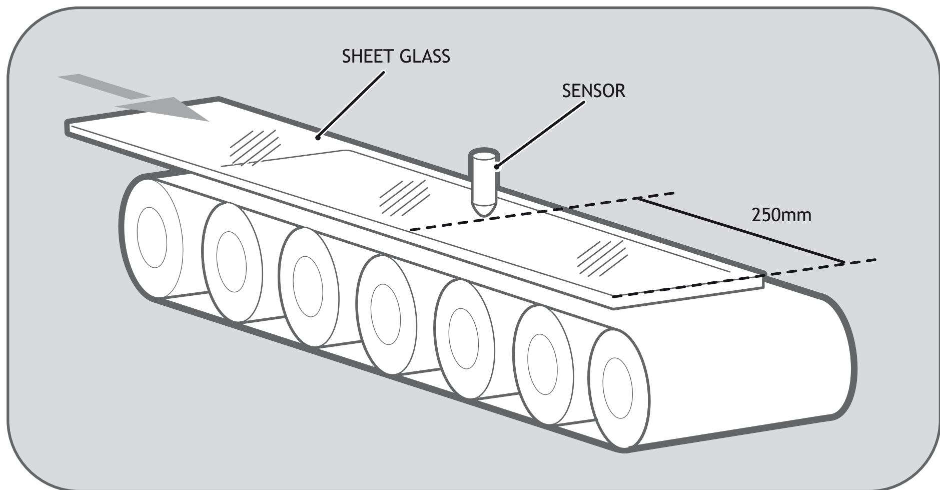
Alternate Format: **MM()**

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**.

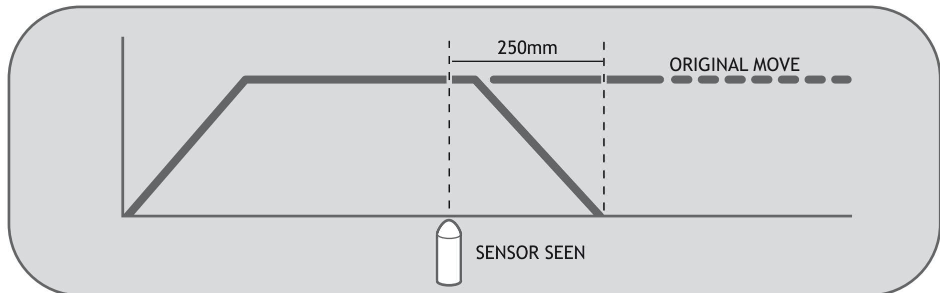
See also: **ENDMOVE**

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

Example 1: 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:

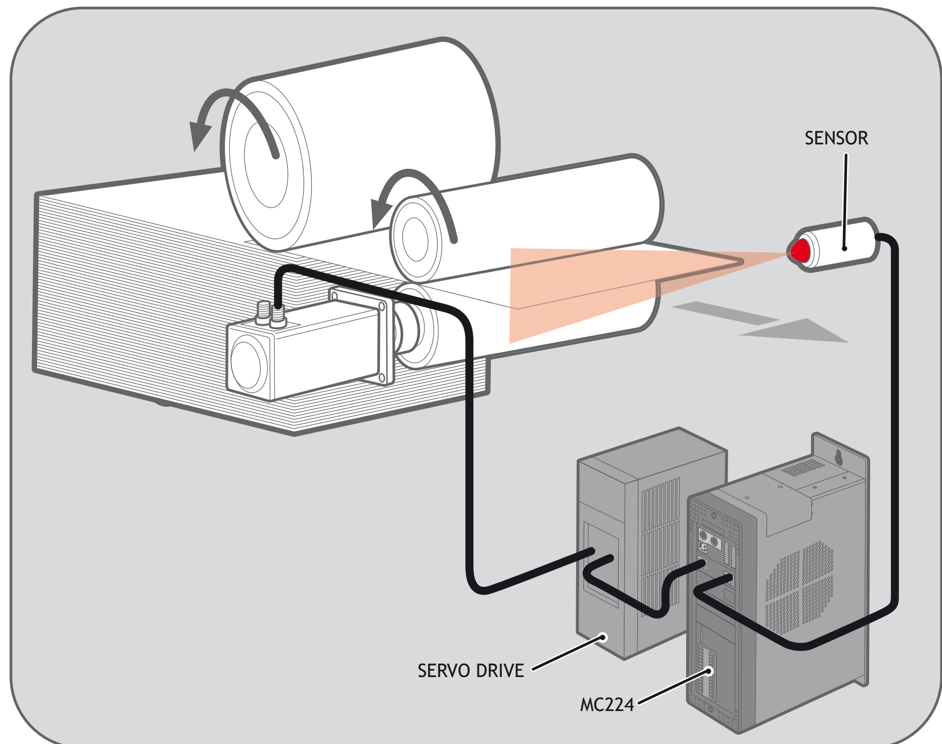



```
MOVE(10000)        'Start a long move on conveyor
REGIST(3)          'set up registration
WAIT UNTIL MARK    'MARK goes TRUE when sensor detects glass edge
OFFPOS = -REG_POS  'set position where mark was seen to 0
WAIT UNTIL OFFPOS=0'wait for OFFPOS to take effect
MOVEMODIFY(250)   'change move to stop at 250mm
```



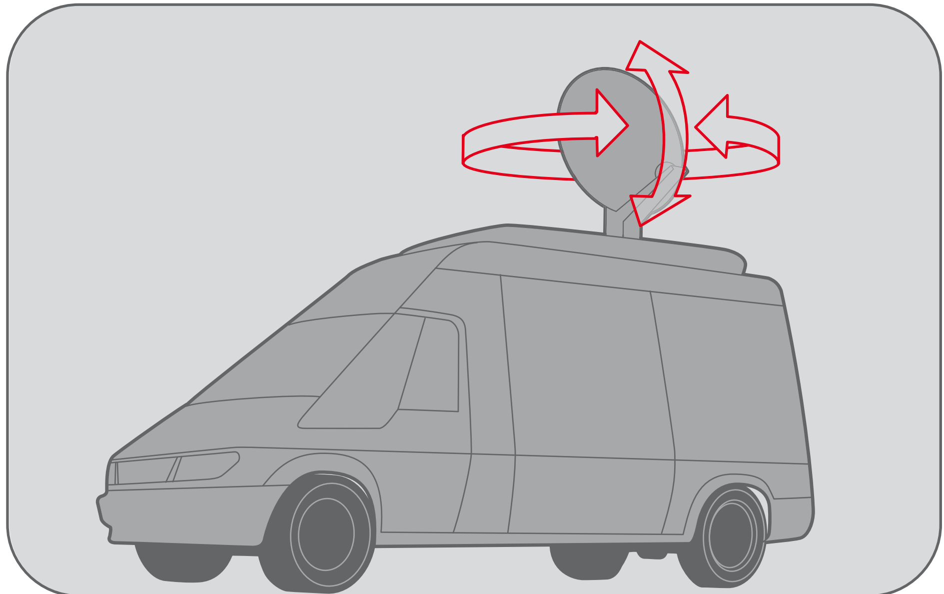
Eample 2: A paper feed system slips. To counteract this, a proximity sensor is positioned one third of the way into the movement. This detects at which position the paper passes and so how much slip has occurred. The move is then modified to account for this variation.

```
paper_length=4000
DEFPOS(0)
REGIST(3)
MOVE(paper_length)
WAIT UNTIL MARK
slip=REG_POS-(paper_length/3)
offset=slip*3
MOVEMODIFY(paper_length+offset)
```



Eample 3: A satellite receiver sits on top of a van; it has to align correctly to the satellite from data processed in a computer. This information is sent to the controller through the serial link and sets VR's 0 and 1. This information is used to control the two axes. **MOVEMODIFY** is used so that the position can be continuously changed even if the previous set position has not been achieved.

```
bearing=0                               'set lables for VRs
elevation=1
UNITS AXIS(0)=360/counts_per_rev0
UNITS AXIS(1)=360/counts_per_rev1
WHILE IN(2)=ON
  MOVEMODIFY(VR(bearing))AXIS(0)    'adjust bearing to match VR0
  MOVEMODIFY(VR(elevation))AXIS(1)  'adjust elev to match VR1
  WA(250)
WEND
RAPIDSTOP                               'stop movement
WAIT IDLE AXIS(0)
MOVEABS(0) AXIS(0)                    'return to transport position
WAIT IDLE AXIS(1)
MOVEABS(0) AXIS (1)
```



MOVESP

Type: Motion Command

Only available in system software versions where "LookAhead" is enabled.

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

Description: Works as **MOVE** and additionally allows vector speed to be changed when using multiple moves in the look ahead buffer when **MERGE=ON**, using additional parameters **FORCE_SPEED** and **ENDMOVE_SPEED**.

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 maximum number of parameters, and therefore axes interpolated, is 4.

Example: In a series of buffered moves using the look ahead buffer with **MERGE=ON**, an incremental move is required where the incoming vector speed is 40units/second and the finishing vector speed is 20 units/second.

```
FORCE_SPEED=40
ENDMOVE_SPEED=20
MOVESP(100,100)
```

Note: For more information see **MOVE**.

MSPHERICAL

Type: Motion Command

Syntax: **MSPHERICAL(endx, endy, endz, midx, midy, midz, mode)**

Description: Moves the three axis group defined in BASE along a spherical path with a vector speed determined by the SPEED set in the X axis. There are 2 modes of operation with the option of finishing the move at an endpoint different to the start, or returning to the start point to complete a circle. The path of the movement in 3D space can be defined either by specifying a point somewhere along the path, or by specifying the centre of the sphere.

Parameters:

endx, endy, endz: Mode=0 or 1: Coordinates of the end point.
Mode=2: Coordinates of a second point on the curve.

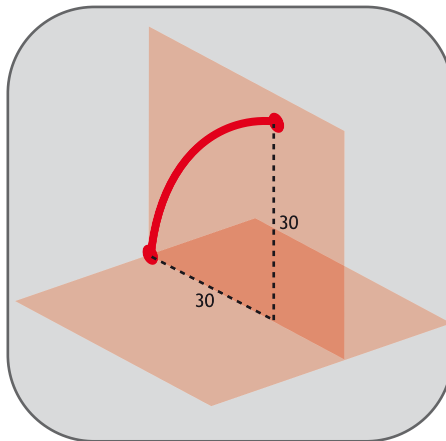
midx, midy, midz: Mode=0 or 2: Coordinates of a point along the path of the curve.
Mode=1 or 3: Coordinates of the sphere centre.

mode: Specifies the way the end and mid parameters are used in calculating the curve in 3D space.
0 = specify end point and mid point on curve.
1 = specify end point and centre of sphere.
2 = mid point 2 and mid point 1 are specified and the curve completes a full circle.
3 = mid point on curve and centre of sphere are specified and the curve completes a full circle.

Note: *The coordinates of the mid point and end point must not be co-linear. Semi-circles cannot be defined by using mode 1 because the sphere centre would be co-linear with the endpoint.*

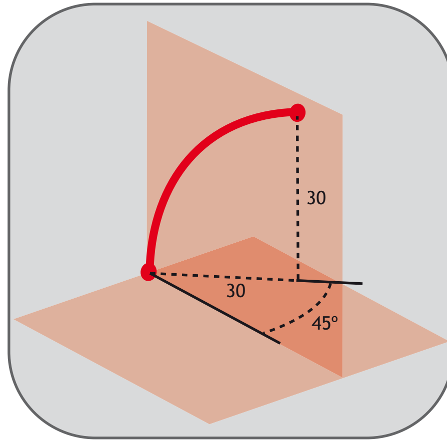
Example 1: A move is needed that follows a spherical path which ends 30mm up in the Z direction:

```
BASE(3,4,5)
MSPHERICAL(30,0,30,8.7868,0,21.2132,0)
```



Example 2: A similar move that follows a spherical path but at 45 degrees to the Y axis which ends 30mm above the XY plane:

```
BASE(0,1,2)
MSPHERICAL(21.2132,21.2132,30,6.2132,6.2132,21.2132,0)
```



MOVETANG

Type: Motion Command

Only available in system software versions where "LookAhead" is enabled.

Syntax: **MOVETANG(absolute_position, [link_axis])**

Description: Moves the axis to the required position using the programmed **SPEED**, **ACCEL** and **DECEL** for the axis. The direction of movement is determined by a calculation of the shortest path to the position assuming that the axis is rotating and that **REP_DIST** has been set to PI radians (180 degrees) and that **REP_OPTION=0**.

Important: The **REP_DIST** value will depend on the **UNITS** value and the number of steps representing PI radians. For example if the rotary axis has 4000 pulses/turn and **UNITS=1** the **REP_DIST** value would be 2000.

If a **MOVETANG** command is running and another **MOVETANG** is executed for the same axis, the original command will not stop, but the endpoint will become the new absolute position.

Parameters:

absolute_position: The absolute position to be set as the endpoint of the move. Value must be within the range -PI to +PI in the units of the rotary axis. For example if the rotary axis has 4000 pulses/turn, the UNITS value=1 and the angle required is PI/2 (90 deg) the position value would be 1000.

link_axis An optional link axis may be specified. When a link_axis is specified the system software calculates the absolute position required each servo cycle based on the link axis **TANG_DIRECTION**. The **TANG_DIRECTION** is multiplied by the REP_DIST/PI to calculate the required position. Note that when using a link_axis the absolute_position parameter becomes unused. The position is copied every servo cycle until the **MOVETANG** is **CANCEL**led.

Example 1: An X-Y positioning system has a stylus which must be turned so that it is facing in the same direction as it is travelling at all times. A tangential control routine is run in a separate process.

```
BASE(0,1)
WHILE TRUE
  angle=TANG_DIRECTION
  MOVETANG(angle) AXIS(2)
WEND
```

Example 2: An X-Y positioning system has a stylus which must be turned so that it is facing in the same direction as it is travelling at all times.

The XY axis pair are axes 4 and 5. The tangential stylus axis is 2:

```
MOVETANG(0,4) AXIS(2)
```

Example 3: An X-Y cutting table has a “pizza wheel” cutter which must be steered so that it is always aligned with the direction of travel. The main X and Y axes are controlled by Motion Coordinator axes 0 and 1, and the pizza wheel is turned by axis 2.

Control of the Pizza Wheel is done in a separate program from the main X-Y motion program. In this example the steering program also does the axis initialisation.

Program TC_SETUP.BAS:

```
'Set up 3 axes for Tangential Control

WDOG=OFF

BASE(0)
P_GAIN=0.9
VFF_GAIN=12.85
```

```
UNITS=50 'set units for mm
SERVO=ON

BASE(1)
P_GAIN=0.9
VFF_GAIN=12.30
UNITS=50 'units must be the same for both axes
SERVO=ON

BASE(2)
UNITS=1 ' make units 1 for the setting of rep_dist
REP_DIST=2000 'encoder has 4000 edges per rev.
REP_OPTION=0
UNITS=4000/(2*PI) 'set units for Radians
SERVO=ON

WDOG=ON
' Home the 3rd axis to its Z mark
DATUM(1) AXIS(2)
WAIT IDLE
WA(10)

'start the tangential control routine
BASE(0,1) 'define the pair of axes which are for X and Y
' start the tangential control
BASE(2)
MOVETANG(0, 0) ' use axes 0 and 1 as the linked pair
```

Program MOTION.BAS:

```
'program to cut a square shape with rounded corners
MERGE=ON
SPEED=300

nobuf=FALSE     'when true, the moves are not buffered
size=120        'size of each side of the square
c=30            'size (radius) of quarter circles on each corner

DEFPOS(0,0)
WAIT UNTIL OFFPOS=0
WA(10)

MOVEABS(10,10+c)
REPEAT
  MOVE(0,size)
  MOVECIRC(c,c,c,0,1)
```

```
  IF nobuf THEN WAIT IDLE:WA(2)
  MOVE(size,0)
  MOVECIRC(c,-c,0,-c,1)
  IF nobuf THEN WAIT IDLE:WA(2)
  MOVE(0,-size)
  MOVECIRC(-c,-c,-c,0,1)
  IF nobuf THEN WAIT IDLE:WA(2)
  MOVE(-size,0)
  MOVECIRC(-c,c,0,c,1)
  IF nobuf THEN WAIT IDLE:WA(2)
UNTIL FALSE
```

RAPIDSTOP

Type: Motion Command

Syntax: **RAPIDSTOP**

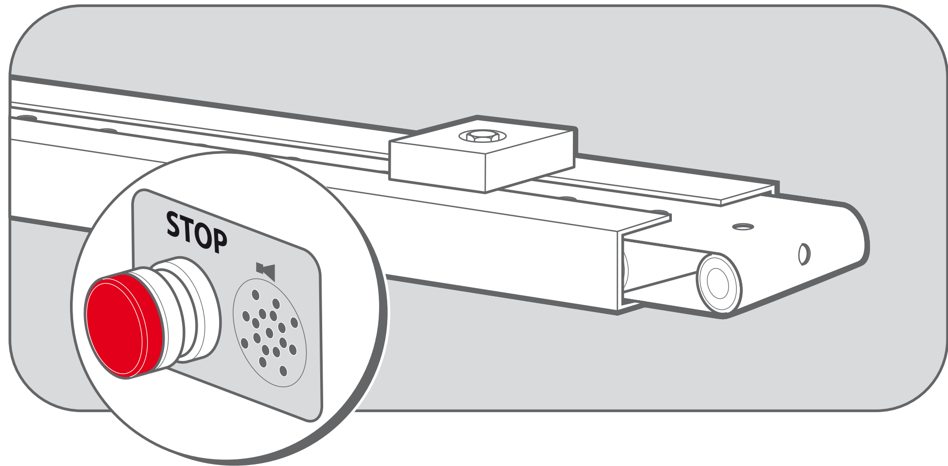
Alternate Format: **RS**

Description: Rapid Stop. The **RAPIDSTOP** command cancels the currently executing move on ALL axes. Velocity profiled move types such as **MOVE, MOVEABS, MHELICAL** etc. will be ramped down at the axes' programmed **DECEL** rate. Others will be immediately cancelled.

The next-move buffers and the process buffers are NOT cleared.

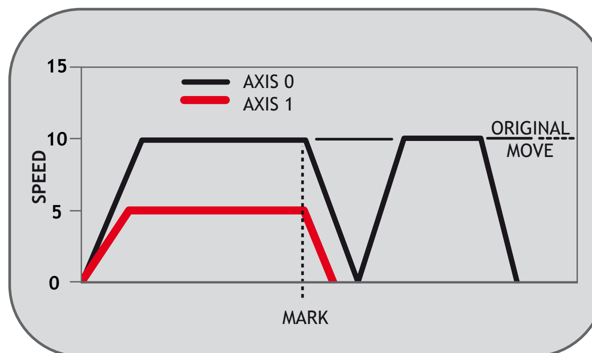
Example 1: Implementing a stop override button that cuts out all motion.

```
CONNECT (1,0) AXIS(1)    'axis 1 follows axis 0
BASE(0)
REPAEAT
  MOVE(1000) AXIS (0)
  MOVE(-100000) AXIS (0)
  MOVE(100000) AXIS (0)
UNTIL IN (2)=OFF         'stop button pressed?
RAPIDSTOP
WA(10)                   'wait to allow running move to cancel
RAPIDSTOP                'cancel the second buffered move
WA(10)
RAPIDSTOP                'cancel the third buffered move
```



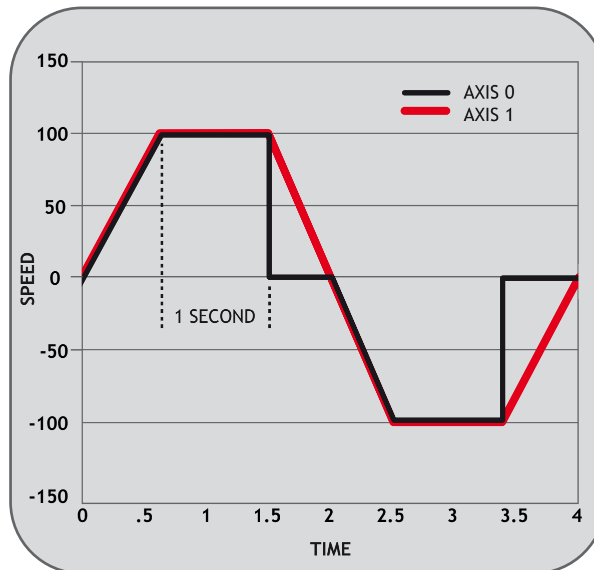
Example 2: Using **RAPIDSTOP** to cancel a **MOVE** on the main axis and a **FORWARD** on the second axis. After the axes have stopped, a **MOVEABS** is applied to re-position the main axis.

```
  BASE(0)
  REGIST(3)
  FORWARD AXIS(1)
  MOVE (100000) 'apply a long move
  WAIT UNTIL MARK
  RAPIDSTOP
  WAIT IDLE     'for MOVEABS to be accurate, the axis must stop
MOVEABS(3000)
```



Example 3: Using **RAPIDSTOP** to break a connect, and stop motion. The connected axis stops immediately on the **RAPIDSTOP** command, the forward axis decelerates at the decel value.

```
BASE(0)
CONNECT(1,1)
FORWARD AXIS(1)
WAIT UNTIL VPSPEED=SPEED 'let the axis get to full speed
WA(1000)
RAPIDSTOP
WAIT IDLE AXIS(1)       'wait for axis 1 to decel
CONNECT(1,1)            're-connect axis 0
REVERSE AXIS(1)
WAIT UNTIL VPSPEED=SPEED
WA(1000)
RAPIDSTOP
WAIT IDLE AXIS(1)
```



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 MC206X 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.

The Enhanced Servo Daughter Board has similar functionality to the MC206X, with the dual registration capability extended to 2 separate 24V inputs in addition to the Z mark. Mode numbers 10 to 13 cover the use of inputs R0 and R1.

Parameters: **mode:** Determines the position to capture.

All registration capable products:

1 - Absolute position when Z Mark rising edge

2 - Absolute position when Z Mark falling edge

3 - Absolute position when R Input rising edge

4 - Absolute position when R Input falling edge

5 - Unused

6 - R Input rising into REG_POS & Z Mark rising into REG_POSB.

7 - R Input rising into REG_POS & Z Mark falling into REG_POSB.

8 - R Input falling into REG_POS & Z Mark rising into REG_POSB.

9 - R Input falling into REG_POS & Z Mark falling into REG_POSB

P201 Enhanced Servo Daughter Board only:

10 - R0 Input rising into REG_POS & R1 Input rising into REG_POSB.

11 - R0 Input rising into REG_POS & R1 Input falling into REG_POSB.

12 - R0 Input falling into REG_POS & R1 Input rising into REG_POSB.

13 - R0 Input falling into REG_POS & R1 Input falling into REG_POSB

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:

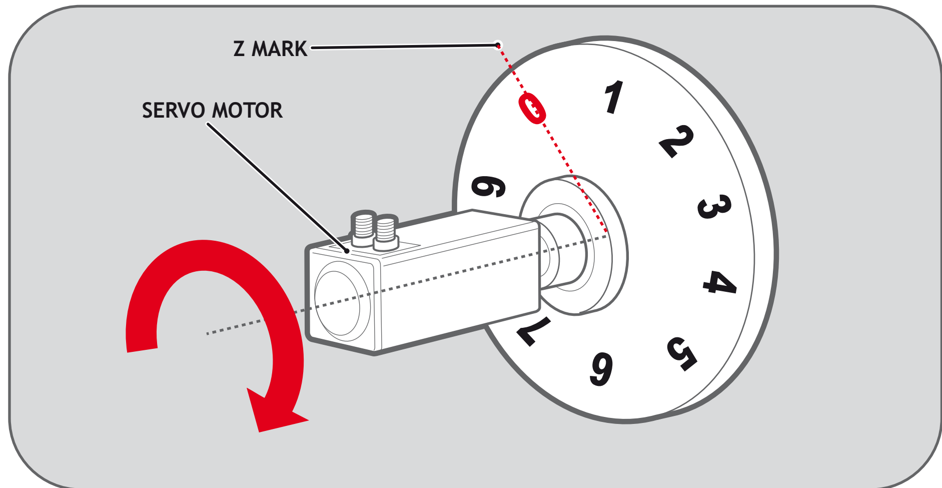
When the windowing function is applied signals will be ignored if the axis measured position is not in the range:

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

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

Example1 : A disc used in a laser printing process requires registration to the Z marker before printing can start. This routine locates to the Z marker, then sets that as the zero position.

```
REGIST(1)            'set registration point on Z mark
FORWARD              'start movement
WAIT UNTIL MARK
CANCEL               'stops movement after Z mark
WAIT IDLE
MOVEABS (REG_POS)    'relocate to Z mark
WAIT IDLE
DEFPOS(0)            'set zero position
```

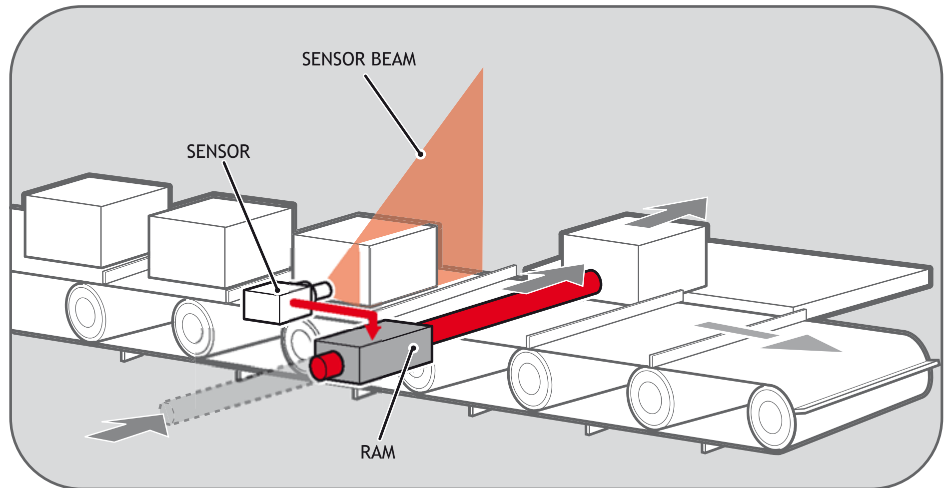


Example 2: Registration with windowing

It is required to detect if a component is placed on a flighted belt so windowing is used to avoid sensing the flights. The flights are at a pitch of 120 mm and the component will be found between 30 and 90mm. If a component is found then an actuator is fired to push it off the belt.

```
REP_DIST=120       'sets repeat distance to pitch of belt flights
REP_OPTION=ON
OPEN_WIN=30              'sets window open position
CLOSE_WIN=90             'sets window close position
REGIST(4+256)            'R input registration with windowing
FORWARD                   'start the belt
box_seen=0
REPEAT
  WAIT UNTIL MPOS<60  'wait for centre point between flights
  WAIT UNTIL MPOS>60  'so that actuator is fired between flights
  IF box_seen=1 THEN  'was a box seen on the previous cycle?
    OP(8,ON)          'fire actuator
    WA(100)
    OP(8,OFF)         'retract actuator
    box_seen=0
  ENDIF
  IF MARK THEN box_seen=1 'set "box seen" flag
  REGIST(4+256)
UNTIL IN(2)=OFF
CANCEL                    'stop the belt
```

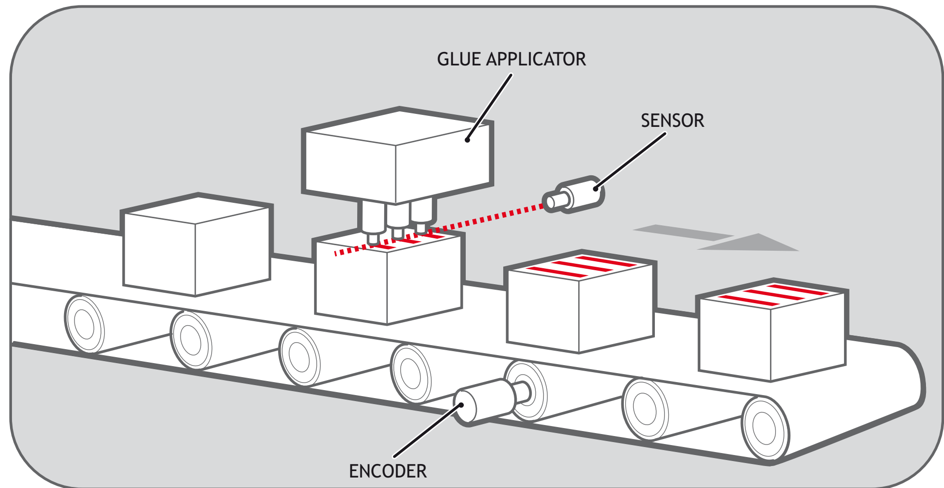
```
WAIT IDLE
```



Example 3: Dual Input Registration

A machine adds glue to the top of a box by switching output 8. It must detect the rising edge (appearance) of and the falling edge (end) of a box. Additionally it is required that the mpos be reset to zero on the detection of the z position.

```
  reg=6 'select registration mode 6 (rising edge R, rising edge Z)
  REGIST(reg)
  FORWARD
  WHILE IN(2)=OFF
    IF MARKB THEN 'on a Z mark mpos is reset to zero
      OFFPOS=-REG_POSB
      REGIST(reg)
    ELSEIF MARK THEN 'on R input output 8 is toggled
      IF reg=6 THEN
      'select registration mode 8 (falling edge R, rising edge Z)
      reg=8
      OP(8,ON)
    ELSE
      reg=6
      OP(8,OFF)
    ENDIF
    REGIST(reg)
  ENDIF
  WEND
CANCEL
```



REGIST_SPEED

Type: Axis Parameter (Read Only)

Description: Stores the change_of_position in user units per msec captured when **MARK** goes TRUE.

In most real-world systems there are delays built into the registration circuit; the external sensor and the input opto-isolator will have some fixed response time. As machine speed increases, the fixed electrical delays will have an effect on the captured registration position.

REGIST_SPEED returns the value of axis speed captured at the same time as **REG_POS**. The captured speed and position values can be used to calculate a registration position that does not vary with speed because of the fixed delays.

Example:
```
fixed_delays=0.020 ' circuit delays in milliseconds
REGIST(3)
WAIT UNTIL MARK
captured_position = REG_POS-(REGIST_SPEED*fixed_delays)
```

Note: This parameter has the units of user_units/msec at all **SERVO_PERIOD** settings.

REVERSE

Type: Axis Command

Syntax: **REVERSE**

Alternate Format: **RE**

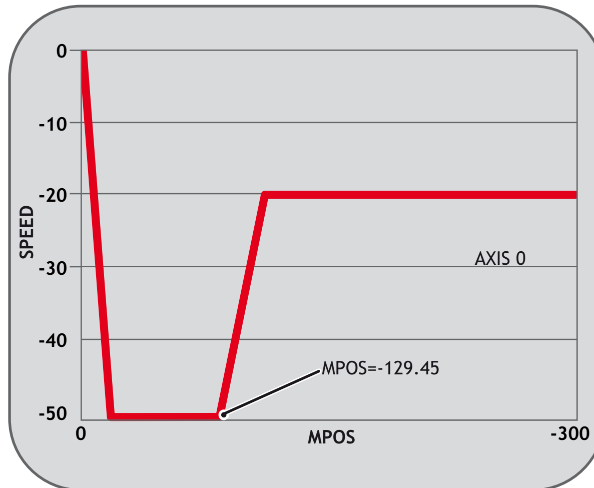
Description: Sets continuous reverse movement on the specified or base axis. The axis accelerates at the programmed **ACCEL** rate and continues moving at the **SPEED** value until either a **CANCEL** or **RAPIDSTOP** command are encountered. It then decelerates to a stop at the programmed **DECEL** rate.
If the axis reaches either the reverse limit switch or reverse soft limit, the **REVERSE** will be cancelled and the axis will decelerate to a stop.

Example 1: Run an axis in reverse. When an input signal is detected on input 5, stop the axis.

```
back:
  REVERSE
  'Wait for stop signal:
  WAIT UNTIL IN(5)=ON
  CANCEL
  WAIT IDLE
```

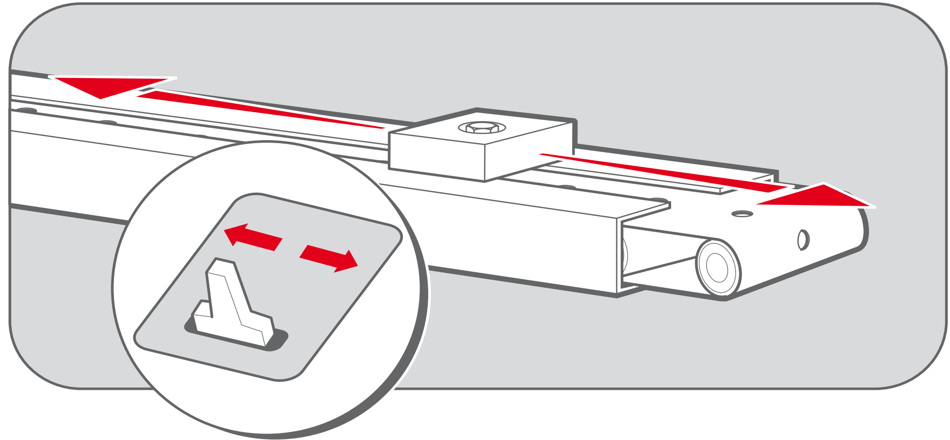
Example 2: Run an axis in reverse. When it reaches a certain position, slow down.

```
  DEFPOS(0)      'set starting position to zero
  REVERSE
  WAIT UNTIL MPOS<-129.45
  SPEED=slow_speed
  WAIT UNTIL VP_SPEED=slow_speed 'wait until the axis slows
  OP(11,ON)      'turn on an output to show that speed is now slow
```



Example 3: A joystick is used to control the speed of a platform. A deadband is required to prevent oscillations from the joystick midpoint. This is achieved through setting reverse, which sets the correct direction relative to the operator, the joystick then adjusts the speed through analogue input 0.

```
REVERSE
WHILE IN(2)=ON
  IF AIN(0)<50 AND AIN(0)>-50 THEN 'sets a deadband in the input
    SPEED=0
  ELSE
    SPEED=AIN(0)*100                'sets speed to a scale of AIN
  ENDIF
WEND
CANCEL
```



STEP_RATIO

Type: Axis Command

Syntax: **STEP_RATIO(output_count, dpos_count)**

Description: This command sets up an Integer ratio for the axis' stepper output. Every servo-period the number of steps is passed through the step_ratio function before it goes to the step pulse output.

The **STEP_RATIO** function operates before the divide by 16 factor in the stepper axis. This maintains the good timing resolution of the stepper output circuit.

Parameters:

output_count:	Number of counts to output for the given dpos_count value. Range: 0 to 16777215.
dpos_count:	Change in DPOS value for corresponding output count. Range: 0 to 16777215.

Note 1: *Large ratios should be avoided as they will lead to either loss of resolution or much reduced smoothness in the motion. The actual physical step size x 16 is the basic resolution of the axis and use of this command may reduce the ability of the Motion Coordinator to accurately achieve all positions.*

Note 2: *`STEP_RATIO` does not replace `UNITS`. Do not use `STEP_RATIO` to remove the x16 factor on the stepper axis as this will lead to poor step frequency control.*

Example 1: Two axes are set up as X and Y but the axes' steps per mm are not the same. Interpolated moves require identical **UNITS** values on both axes in order to keep the path speed constant and for **MOVECIRC** to work correctly. The axis with the lower resolution is changed to match the higher step resolution axis so as to maintain the best accuracy for both axes.

```
'Axis 0: 500 counts per mm (31.25 steps per mm)
'Axis 1: 800 counts per mm (50.00 steps per mm)

  BASE(0)
  STEP_RATIO(500,800)
  UNITS = 800
  BASE(1)
  UNITS = 800
```

Example 2: A stepper motor has 400 steps per revolution and the installation requires that it is controlled in degrees. As there are 360 dgrees in one revolution, it would be better from the programmer's point of view if there are 360 counts per revolution.

```
BASE(2)
STEP_RATIO(400, 360)
           'Note: this has reduced resolution of the stepper axis
MOVE(360*16) 'move 1 revolution
```

Example 3: Remove the step ratio from an axis.

```
BASE(0)
STEP_RATIO(1, 1)
```

Input / Output Commands

AIN

Type: Function

Syntax: **AIN(analogue chan)**

Description Reads a value from an analogue input. A variety of analogue input modules may be connected to the *Motion Coordinator* and some *Motion Coordinator*s have one or two analogue inputs built-in. The value returned is the decimal equivalent of the binary number read from the A to D converter.

Parameters: **analogue chan:** analogue input channel number 0...71

0 to 31:	P325 CAN Analog input channels.
32 to 39:	Analogue inputs built-in to the *Motion Coordinator*. including when P184 is fitted to Euro209 and PCI208.
40 to 71:	P225 Analog Input Daugther Board.

	Resolution	Biploar / Unipolar / Scale
MC206X:	10 bit	Unipolar, 0 - 12V, 0 - 1023
Euro295x:	12 bit	Unipolar, 0-10V
MC224:	12 bit	Unipolar, 0 - 10V
P325:	12 bit	Bipolar, -10V - +10V, -2048 - +2047
P225:	16 bit	Unipolar, 0 - 10V, 0 - 65535
MC302-k	Analogue input of Servodrive	

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
```

The analogue 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**.

Note: Speed of analogue response depends on which module it comes from. P325 updates at 10msec, P225 at the selected **SERVO_PERIOD** and built-in analogue ports at 1 msec.

If no P325 CAN Analog modules are fitted, **AIN(0)** and **AIN(1)** will read the built-in channels so as to maintain compatibility with previous versions.

AIN0..3 / AINBI0..3

Type: System Parameter

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

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

AOUT0...3

Type: Reserved Keyword

CHANNEL_READ

Type: Command

Syntax: **CHANNEL_READ(<channel>,<buffer_base>,<buffer_size>[,<delimiter_base>,<delimiter_size>[,<escape_character>[,<crc>]]])**

Description: **CHANNEL_READ** will read bytes from the channel and store them into the VR data starting at buffer_base.

CHANNEL_READ will stop when it has read buffer_size bytes, the channel is empty, or the character read from the channel is specified in the delimiter buffer.

If the escape character received then the next character is not interpreted. This allows delimiter characters to be received without stopping the **CHANNEL_READ**.

The calculated CRC will be stored in the **VR(<crc>)**

Parameters:

<channel>	Communication or file channel.
<buffer_base>	Number of the first VR for the buffer.
<buffer_size>	Size of the buffer.

<delimiter_base>	Position in the VR data to the start of the delimiter list.
<delimiter_size>	Size of the delimiter list.
<escape_character>	When this character is received the following character is not interpreted.
<crc>	Position in the VR data where the CRC will be stored.

CHANNEL_WRITE

Type: Command

Syntax: **CHANNEL_WRITE(<channel>,<buffer_base>,<buffer_size>)**

Description: **CHANNEL_WRITE** will send buffer_size bytes from the VR data starting at buffer_base to the channel

Parameters:

<channel>	Communication or file channel.
<buffer_base>	Position in the VR data to the start of the buffer.
<buffer_size>	Size of the buffer.

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 versions of BASIC.

Example:

```
>>PRINT CHR(65);
A
PRINT #1,CHR($32);CHR(71);CHR(75);
```

CLOSE

Type: Command

Syntax: **CLOSE #<channel>**

Description: **CLOSE** will close the file on the specified channel.

Parameters: **<channel>** The TrioBASIC I/O channel to be associated with the file. It is in the range 40 to 44.

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#4,CURSOR(60);">Bottom line";
```

DEFKEY

Type: Command

Syntax: **DEFKEY(key no, keyvalue1, [keyvalue2, [keyvalue3 ..]])**

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.

Parameters:

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

ENABLE_OP

Type: Reserved Keyword

FILE

Type: Function

Syntax: **FILE <function> [<parameters>]**

Description: This command enables the user to manage the data on the SDCARD.

Parameters: The parameters depend on the function

"CD" "<directory>"	Change to the given directory. There is one active directory on the controller all **SDCARD** commands are relative to this directory.
"DEL" "<file>"	Delete the given file inside the current directory.
"DETECT"	Returns **TRUE** if an **SDCARD** is detected correctly.

"DIR"	Print the contents of the current directory to the current output channel.
"FIND_FIRST" <type> <VR>	Initialises the internal **FIND** structures and locates the first directory entry of the given type. If a directory entry is found then the function returns **TRUE** and the VR variable at index vr is the start of the VRSTRING that contains the name of the directory entry. If no directory entry is found or there is an error initialising the internal **FIND** structures then the function returns **FALSE**. Valid values for type are: 0.FILE or DIRECTORY 1.FILE 2.DIRECTORY
"FIND_NEXT" <VR>	Finds the next directory entry of the type given in the corresponding **FIND_FIRST** command. If a directory entry is found then the function returns **TRUE** and the VR variable at index vr is the start of the VRSTRING that contains the name of the directory entry. If no directory entry is found or there is an error initialising the internal FIND structures then the function returns **FALSE**.
"FIND_PREV" <VR>	Finds the previous directory entry of the type given in the corresponding **FIND_FIRST** command. If a directory entry is found then the function returns **TRUE** and the VR variable at index vr is the start of the VRSTRING that contains the name of the directory entry. If no directory entry is found or there is an error initialising the internal FIND structures then the function returns **FALSE**.
"LOAD_PROGRAM" "<name>"	Load the given program into the internal RAM on the *Motion Coordinator*. Only .BAS files are handled at the moment.
"LOAD_PROJECT" "<name>"	Read the given MotionPerfect project file and load all the programmes into internal RAM on the *Motion Coordinator*.
"RD" "<directory>"	Delete the given directory inside the current directory.
"MD" "<directory>"	Create the given directory inside the current directory.
"PWD"	Prints the path of the current directory to the current output channel.

"SAVE_PROGRAM" "<name>"	Save the given program to the corresponding file on the SDCARD inside the current directory. Only .BAS files are handled at the moment.
"SAVE_PROJECT" "<name>"	Create a *Motion* Perfect project with the given name inside the current directory. This implies creating the directory and the corresponding project and program files within this directory.
"TYPE" "<file>"	Read the contents of the file inside the current directory and print it to the current output channel.

FLAG

Type: Command/Function

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

Description: The **FLAG** command is used to set and read a bank of 24 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..23.

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(21,ON)' Set flag bit 21 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 24 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

```
REPEAT
    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
UNTIL TRUE
```

HEX

Type: Command

Description: The **HEX** command is used in a print statement to output a number in hexadecimal format.

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.

```
WHILE TRUE
  MOVEABS(IN(4,7)*1.5467)
  WAIT IDLE
WEND
```

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 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 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.

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.
On all controllers except the MC302X, add 100 to the channel number to return the number of characters in the buffer.
On the MC302, the **Key#** channel returns the number of characters in the buffer.

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(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. (See example 3).

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 (18*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 to 7 do not exist.

Example 3:
```
read_output:
  VR(0)=OP
  'SET OUTPUTS 8..15 ON SIMULTANEOUSLY
  VR(0)=VR(0) AND $FF00
  OP(VR(0))
```
Note how this example can also be written:

```
  'SET OUTPUTS 8..15 ON SIMULTANEOUSLY
  OP(OP AND $FF00)
```

See also READ_OP()

OPEN

Type: Command

Syntax: **OPEN #<channel> AS "<name>" for <access>**

Description: **OPEN** will open the specified file for the given access type and assign it to the specified TrioBASIC I/O channel. Once the file has been opened then it can be manipulated by the standard TrioBASIC channel commands. If the file is opened with read access then the **GET, INPUT, LINPUT, KEY** commands can be used on the channel. If the file is opened with write access then the **PRINT** command can be used on the channel.

Parameters:

<channel> The TrioBASIC I/O channel to be associated with the file. It is in the range 40 to 44.

<access> The operations permitted on the file. The valid access types are:

INPUT

The file will be opened for reading. When the end of the file is reached KEY will return FALSE, and the GET and INPUT functions will fail.

OUTPUT

The file will be opened for writing. If the file does not exist then it will be created. If the file does exist then it will be overwritten.

FIFO_READ

The file will be opened for reading and will be managed as a circular buffer. This is only valid for files stored in internal RAM.

FIFO_WRITE(<size>)

The file will be opened for writing and will be managed as a circular buffer. This is only valid for files in internal RAM. If the file does not exist it will be created <size> bytes long. If the file does exist then it must be of type FIFO and the size parameter is ignored.

<name> Name of the file to be opened.
The format is **"[memory:]filename"** where memory is either **RAM** or **SD.** If the prefix is omitted or is **RAM:** then filename refers to an internal RAM directory entry. If the prefix is **SD:** then filename refers to an **SDCARD** directory entry.

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"
```

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: Suppose VR(1)=6 and variab=1.5:

```
PRINT VR(1)[4,1],variab[6,2]
```

print output will be:

```
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#**. The **PRINT** statement has limits to the size of big numbers that it can display. Max value that you can put in a variable and then display it is: 2147483999. (The variable actually holds 2147483648)

The largest negative value is -2147483999. i.e the variable holds the value -2147483648.

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
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 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 5 enable **PSWITCH** on DPOS
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 daughter board axis 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)
```

READ_OP()

Type: Function.

Syntax: **READ_OP(output no[,final output])**

Description: Returns the value of digital outputs. If called with one parameter whose value is less than the highest output channel, it returns the value (1 or 0) of that particular output channel. If called with 2 parameters **READ_OP()** returns, in binary, the sum of the group of outputs. In the 2 parameter case the outputs should be less than 24 apart.

Parameters:

output no:	output to return the value of/start of output group
[final output]:	last output of group

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

```
test:
  WAIT UNTIL READ_OP(12)=ON
  GOSUB place
```

Example 2: Check the group of 8 outputs and call a routine if any of them are ON.

```
  op_bits = READ_OP(16,23)
  IF op_bits<>0 THEN
    GOSUB check_outputs
  ENDIF
```

Note: READ_OP checks the state of the output logic. No actual output needs to be present for the returned value to be ON.

In the Euro205x, READ_OP(8 ... 15) is different to IN(8 ... 15) because there are separate inputs and outputs at these addesses.

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

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: Message type 1: data1 is the VR variable number on the destination *Motion Coordinator*.
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. 10 is the next node and 24 is the fifteenth node away from the keypad.

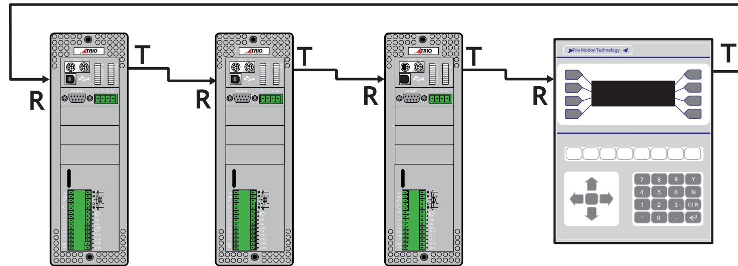
data2: Only used if message is type 1. In this case it contains the value for the specified variable.

Example 1: Two *Motion Coordinator*s are fibre-optic networked together. One is acting under instruction from the other. Instructions are given by setting VR(100) to different values on the receiving *Motion Coordinator*. The program on the master *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 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; 3 *Motion Coordinator*s and 1 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][,variable][,timeout][,linetype])**

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

By default the controller sets 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: 7 or 8

stopbits: 1 or 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 (16 bit Integer)
5 : Hostlink Slave
6 : Hostlink Master
7 : MODBUS protocol (32 bit IEEE floating point)
8 : REMOTE end of TrioPC ActiveX synchronous link
9 : MODBUS protocol (32bit long word integers)

variable: Determines the target variable array for MODBUS transfers.
0 : VR()
1 : TABLE()

Timeout Communications timeout (msec). Default is 3

linetype 0 = 4 wire Rs485, 1 = 2 wire Rs485

Example 1:

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

SETCOM(19200,7,2,2,1,1)
```

Example 2:

```
'Set port 2 (RS485) to 9600 baud, 8 data bits, 1 stop bit
'no parity and no XON/XOFF handshake

SETCOM(9600,8,1,0,2,0)
```

Example 3: 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)
```

Example 4: Set port 1 to receive commands from a PC running the TrioPC ActiveX component.

```
'set up RS232 port at 38400 baud, 8 data, 1 stop, even parity
'then start the REMOTE process which will handle the commands
'received from TrioPC.

SETCOM(38400,8,1,2,1,8)
REMOTE(0)
```

TIMER

Type: Command

Syntax: **TIMER(timer_no, output, pattern, time[,option])**

Description: The **TIMER** command allows an output or a selection of outputs to be set or cleared for a predefined period of time. There are 8 timer slots available, each can be assigned to any of the first 32 outputs. The timer can be configured to turn the output ON or OFF.

Parameters:

Timer_no:	The timer number in the range 0-7
Output:	Selects the physical output or first output in a group. This should be in the range 0..31.
Pattern:	1 for a single output. If set to a number this represents a binary array of outputs to be turned on
Time:	The period of operation in milliseconds
Option:	Inverts the output, set to 1 to turn OFF at start and ON at end.

Example1: Use the **TIMER** function to flash an output when there is a motion error. The output lamp should flash with a 50% duty cycle at 5Hz.

```
WAIT UNTIL MOTION_ERROR
WHILE MOTION_ERROR
  TIMER(0,8,1,100) 'turns ON output 8 for 100milliseconds
  WA(200) 'Waits 200 milliseconds to complete the 5Hz period
WEND
```

Example2: Setting outputs 10, 12 and 13 OFF for 70 milliseconds following a registration event. The first output is set to 10 and the pattern is set to 13 (1 0 1 1 in binary) to enable the three outputs. Output 11 is still available for normal use. The option value is set to 1 to turn OFF the outputs for the period, they return to an ON state after the 70 milliseconds has elapsed.

```
WHILE running
  REGIST(3)
  WAIT UNTIL MARK
  TIMER(1,10,13,70,1)
WEND
```

Example3: Firing output 10 for 250 milliseconds during the tracking phase of a MOVELINK Profile

```
WHILE feed=ON
  MOVELINK(30,60,60,0,1)
  MOVELINK(70,100,0,60,1)
  WAIT LOADED 'Wait until the tracking phase starts
```

```
  TIMER(2,10,1,250) 'Fire the output during the tracking phase
  MOVELINK(-100,200,50,50,1)
WEND
```

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. MC206X 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.

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 the command is stepped through with no action. 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 GOTO 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 the command is stepped through with no action. 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
```

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
```

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.

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.

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.

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
```

System Parameters and Commands

ADDRESS

Type: System Parameter

Syntax: **ADDRESS=value**

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

Example:
```
ADDRESS=5
SETCOM(19200,8,1,2,1,4)
```

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.**

Note: This command should only be used on the Command Line Terminal.

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

BATTERY_LOW

Type: System Parameter (Read only)

Syntax: **var = BATTERY_LOW**

Description: For controllers fitted with non rechargeable batteries, this parameter returns the current state of the battery condition. If **BATTERY_LOW** returns 1 then the battery needs to be changed. If **BATTERY_LOW** returns 0 then battery condition is ok.

BOOT_LOADER

Type: System Command

Description: This command is used to enter the boot loader software. This is not normally required by users unless instructed by Trio or a Distributor.

BREAK_ADD

Type: System Command

Syntax: **BREAK_ADD "program name" line_number**

Description: Used by Motion Perfect to insert a break point into the specified program at the specified line number.

Example: **BREAK_ADD "simpletest" 8**
Will add a break point at line 8 of program "simpletest"

Note 1: If there is no code at the given line number **BREAK_ADD** will add the breakpoint at the next available line of code. i.e. If line 8 is empty but line 9 has "NEXT x" and a **BREAK_ADD** is issued for line 8, the break point will be added to line 9.

Note 2: If a non existent line number is selected (i.e. line 50 when the program only has 40 lines), the controller will return an error.

BREAK_DELETE

Type: System Command

Syntax: **BREAK_DELETE "program name" line_number**

Description: Used by *Motion* Perfect to remove a break point from the specified program at the specified line number.

Example: **BREAK_DELETE "simpletest" 8**
Will remove the break point at line 8 of program "simpletest"

Note: If a non existent line number is selected (i.e. line 50 when the program only has 40 lines), the controller will return an error.

BREAK_LIST

Type: System Command

Syntax: **BREAK_LIST "program name"**

Description: Returns a list of all the break points in the given program name. Displays the line number of the breakpoint and the code associated with that line.

Example: For a program called "simpletest" with break points inserted on lines 8 and 11;

```
>>BREAK_LIST "simpletest"

Program: SIMPLETEST
Line 8: SERVO=ON
Line 11: BASE(0)
```

BREAK_RESET

Type: System Command

Syntax: **BREAK_RESET "program name"**

Description: Used by *Motion* Perfect to remove all break points from the specified program.

Example: **BREAK_RESET "simpletest"**
Will remove all break points from program "simpletest"

CAN

Type: System Function

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

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 analogue 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 or the OKI ML9620 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#,...,[rw])

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 Register:	**val=CAN(channel#,0,register#)**
1	Write 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,[rw])**
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 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.

Functions 8 and 9 are only available in system software V1.62 and later.

Products based on the OKI ML9620 chip require the optional rw parameter in the **CAN(5 ..)** command. rw is 0 to set up a read buffer, and 1 for a write buffer.

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 Trio 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 CAN I/O modules.

Value	**Function**
32	**Trio CAN I/O Master 64in/64out**
40	**CanOpen I/O Master 64in/64out**
41	**CanOpen I/O Master 128in/128out**

An additional function of **CANIO_ADDRESS** is to set the initial bit rate for the CANbus port on power up. This enables the CANbus port to come online at the correct rate when installed in factory networks like DeviceNet. Bits 8 and 9 have the following meaning:

Bit 9 , 8 value	Decimal value	Initialisation Baudrate:
0 , 0	0	500 KHz
0 , 1	256	256 KHZ
1 , 0	512	125 KHz
1 , 1	768	1 MHz

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 Trio CAN I/O or 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

CANOPEN_OP_RATE

Type: System Parameter

Description: Used to adjust the transmission rate of CanOpen I/O PDO telegrams. Default is 5msec. Adjustable in 1msec steps.

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 MC302X **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
9	Port 1 Rx Overrun
10	Port 1 Parity Error
11	Port 1 Rx Frame Error
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 MC206X, a communications daughter board will respond with its type if the **COMMSTYPE** is requested from slot(0).

#	Description
20	CAN Communications card
21	USB Communications card
22	SLM Communications card
23	Profibus Communications card
24	SERCOS Communications card
25	Ethernet Communications card
26	P184 4 Analog Out card for PCI208
27	P185 8 Analog Out card for PCI208
28	Analog Input card
29	Enhanced CAN Communications card
30	ETHERNET IP

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**
MC302X	293
Euro205x	255
Euro209	259
MC206X	207
PCI208	208
MC224	224

Note: When the Motion Coordinator is LOCKED, 1000 is added to the above numbers. eg a locked MC206X will return 1207.

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 (MC224 Only)

Description: Returns/ Sets the current date held by the real time clock.

Syntax: WRITE:**DATE=DD:MM:YY, DATE=DD:MM:YYYY**

READ:**d=DATE**

d=DATE(0), d=DATE(1), d=DATE(2)

Parameters:

Option	
none	Returns the number of days since 01/01/1900
1	Returns the day of the current month
2	Returns the month of the current year
3	Returns the current year

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

Example 2:
```
>>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.

Example 3:
```
>>DATE=05:08:2008
>>PRINT DATE(1);"/";DATE(0);"/";DATE(2) 'Prints the date in US format.
08/05/2008
```

DATE$

Type: Command (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 (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 (MC224 only)

Description: Prints the current day as a string.

Example:
```
>>? DAY$
Wednesday
```

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*.

Parameters:

slot:	Specifies the communications slot where the CAN daughter board is placed. Set -1 for built-in CAN port and 0 for a CAN daughter board in the MC206X.
func:	0 = Start the DeviceNet slave protocol on the given slot. 1 = Stop the DeviceNet protocol. 2 = Put startup baudrate into Flash EPROM
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. Range 0..4
poll out len:	Number of words to be sent during poll. Range 0..4

Polled IO data is transferred periodically:
From PLC to [TABLE(poll_base) -> TABLE(poll_base + poll_in_len)]
To PLC from [TABLE(poll_base + poll_in_len + 1) -> TABLE(poll_base + poll_in_len + poll_out_len)]

Example 1:

```
'Start the DeviceNet protocol on the built-in CAN port;
DEVICENET(-1,0,500,30,0,4,2)
```

Example 2:

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

Example 3:

```
'Set the CAN board in slot 0 to have a baudrate of 125k bps on
power-up;
DEVICENET(0,2,125)
```

DIR

Type: System Command

Alternate Format: **LS**

Description: Prints a list of all programs in memory, their size and their **RUNTYPE**. Alternative formats:
DIR F may be used to list the programs stored in the FlashStick if present.
DIR D lists the programs stored in SD card 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 Euro205x and MC206X 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

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

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 **SELECT**ed 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

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

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 *Motion Coordinator*s with a FlashStick socket...

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 information in the MC206X 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.

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 MC206X 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). (R)
8 = Default Gateway
9 = Data configuration. VR() variables (0) or TABLE (1). (R)
10 = Modbus TCP port number. (initialised to 502)

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 (R) is used straight away.

Example 1: Set the IP address, subnet mask and default gateway for the Ethernet daughter board in slot 0.

```
ETHERNET(1,0,0,192,200,185,2)
ETHERNET(1,0,2,255,255,255,0)
ETHERNET(1,0,8,192,200,185,210)
```

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

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

Example 3: Set the Modbus TCP port number in the Ethernet daughter board in slot 1.

```
ETHERNET(1,1,10,1024)
```

Example 4: Initialise the Modbus TCP port for floating point TABLE data. Must be entered before the Modbus master opens the port connection.

```
ETHERNET(2,1,7,1)
ETHERNET(2,1,9,1)
```

Note: Examples 1 to 3 must be entered from the terminal. Example 4 is placed in a startup program as the values are stored in ram.

ETHERNET_IP

Type: System Command

Syntax: Read: **ETHERNET_IP(slot, function, index, #params, vr_base_index)**

Write: **ETHERNET_IP(slot, function, index, param1, [param2, param3, param4])**

Description: Provides access to the memory and functions of the Anybus EIP module on the P298.

Parameters:

slot	the physical daughter board slot where the P298 is located (0 for MC206).
function	0 = read, 1 = write.
Index	memory address to be accessed.
#params	number of parameters to be read consecutively from the address. 1 = only the parameter at the address is read. 2 = address and address+1 are read. 3 = address > address+2 are read. 4 = address > address+3 are read.
vr_base_index	the starting index of a block of up to 4 VRs to read data into.
param1 ..param4	values to be written to address, address+1, address+2, address+3.

The Ethernet_IP function returns either TRUE (-1) or FALSE (0) to indicate the success or otherwise of the function call.

EX

Type: System Command

Syntax: **EX(processor)**

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

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.

Parameters:

0 or None:	Software resets the controller and maintains communications.
1:	Software resets the controller and communications.

Note: When running *Motion* Perfect executing an **EX** command is not allowed. The same effect as an **EX** can be obtained by using "Reset the controller..." under the "Controller" menu in *Motion* Perfect. To simply re-start the programs, use the **AUTORUN** command.

EXECUTE

Type: System Command

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

FB_SET

Type: System Parameter

Description: This special parameter is available on certain *Motion Coordinators* only. Fieldbus Set controls the source for the second value returned by a DeviceNet I/O poll response. The values can be set as follows:

0	I/O Poll returns VR(0) as 16 bit Integer
1	I/O Poll returns inputs 0-15
2	I/O Poll returns inputs 16-31

FB_STATUS

Type: System Parameter

Description: This Read-only parameter returns the current status of the fieldbus connection. At present, only the DeviceNet connection status is supported.

FB_STATUS returns the following values:

0 I/O Polling is OFF

1 I/O Polling is ON

Example:
```
'Test the Polled I/O status to see if PLC is still online
IF FB_STATUS=0 THEN
'PLC link has failed; set global flag and stop motion
  RAPIDSTOP
  VR(50) = 0
ENDIF
```

FEATURE_ENABLE

Type: System Function

Syntax: **FEATURE_ENABLE(feature number)**

Description: Many *Motion Coordinator*s, have the ability to unlock additional axes by entering a "Feature Enable Code". This function is used to enable protected features, such as additional servo axes or remote CAN/SERCOS/Analogue feedback axes, of a controller. It is recommended to use *Motion* Perfect 2 to enter and store the feature enable codes.

Controllers with features which can be enabled in this way 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 via the Trio website or from a Trio 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(function, [flashpage, tablepage])**

Description: Copies user data in RAM to the permanent flash memory.

Parameters:

function: Specifies the required action.
0 to 1023: Store single VR in Flash EPROM
-1: Store one page of TABLE to the Flash EPROM and use it to replace the RAM table data on power-up.
-2: Stop using the EPROM copy of table during power-up.
-3: Write a page of TABLE data into flash EPROM.
-4: Read a page of flash memory into TABLE data.

flashpage: The index number (0 … 15) of a 16k page of Flash EPROM where the table data is to be stored to or retrieved from.

tablepage: The index number (0 … INT(TSIZE/16384)) of the page in table memory where the data is to be copied from or restored to.

Note: Where this feature is provided on controllers which do not have battery backed ram **VR()** storage, each **FLASHVR** command generates a write to flash eprom. After 8000 writes the flash sector will be erased and the firmware writes the data into a second sector. Each sector can be erased over 1,000,000 times. It is therefore possible to use the **FLASHVR([0 ... 1023])** 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.

The **FLASHVR(-1)** and **FLASHVR(-2)** functions can be used with all *Motion Coordinator*'s that have system software 1.52 or later. These functions write a whole block of data to flash memory and the programmer must ensure that they are only used occasionally.

FLASHVR(-3) and **FLASHVR(-4)** is only available with system software 1.6411 or later. Each "page" of table data transferred with this command is 16,384 floating point numbers.

Example 1:
```
VR(25)=k
FLASHVR(25)          'store one VR variable in the MC302X
```

Example 2:
```
FOR v=1 to 10
  FLASHVR(v)         'store a sequence of VR variables
NEXT v
```

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

Example 4:
```
FLASHVR(-3,5,2)      'Store TABLE page 2 to flash EPROM page 5
```

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 robot's geometry.

A number of pre-defined **FRAME**s are available. Please contact your Trio distributor for details.

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.

List Frame types:

0 - Default
1 - 2 axis SCARA robot
2 - XY single belt
3 - Double XY single belt
4 - 2 axis pick and place

5 - 2x2 Matrix transform
6 - Polar to Cartiesian transformation
10 - Cartesian to polar transformation
13 - Dual arm robot transformation

Note: See www.triomotion.com or your distributor for more details.

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 *Motion Coordinator* compiles programs before they are run, this means that a little under twice the memory is required to be able to run a program.

Example 1:

```
>>PRINT FREE
47104.0000
>>
```

See Also: **DIR, TABLE**

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.

HLM_COMMAND

Type: Hostlink Command

Syntax: **HLM_COMMAND(command, port[, node[, mc_area/mode[, mc_offset]]])**

Description: The **HLM_COMMAND** command performs a specific Host Link command operation to one or to all Host Link Slaves on the selected port. Program execution will be paused until the response string has been received or the timeout time has elapsed. The timeout time is specified by using the **HLM_TIMEOUT** parameter. The status of the transfer can be monitored with the **HLM_STATUS** parameter.

Parameters: command

The selection of the Host Link operation to perform:

HLM_MREAD (or value 0)	This performs the Host Link PC MODEL READ (MM) command to read the CPU Unit model code. The result is written to the MC Unit variable specified by mc_area and mc_offset.
HLM_TEST (or value 1)	This performs the Host Link TEST (TS) command to check correct communication by sending string "MCxxx TEST STRING" and checking the echoed string. Check the **HLM_STATUS** parameter for the result.
HLM_ABORT (or value 2)	This performs the Host Link **ABORT (XZ)** command to abort the Host Link command that is currently being processed. The **ABORT** command does not receive a response.
HLM_INIT (or value 3)	This performs the Host Link **INITIALIZE** (**) command to initialize the transmission control procedure of all Slave Units.
HLM_STWR (or value 4)	This performs the Host Link **STATUS WRITE (SC)** command to change the operating mode of the CPU Unit.

port The specified serial port. (See specific controller specification for numbers)

node (for **HLM_MREAD, HLM_TEST, HLM_ABORT and HLM_STWR**):

The Slave node number to send the Host Link command to.
Range: [0, 31].

mode (for **HLM_STWR**)

The specified CPU Unit operating mode.

0 PROGRAM mode
2 MONITOR mode
3 RUN mode

mc_area (for **HLM_MREAD**)

The MC Unit's memory selection to write the received data to.

mc_offset (for **HLM_MREAD**)

The address of the specified MC Unit memory area to read from.

mc_area	Data area
MC_TABLE (or value 8)	Table variable array
MC_VR (or value 9)	Global (VR) variable array

Note 1: When using **HLM_COMMAND**, be sure to set-up the Host Link Master protocol by using the **SETCOM** command.

Note 2: The Host Link Master commands are required to be executed from one program task only to avoid any multi-task timing problems.

Example 1: The following command will read the CPU Unit model code of the Host Link Slave with node address 12 connected to the RS-232C port. The result is written to VR(233).

```
HLM_COMMAND(HLM_MREAD,1,12,MC_VR,233)
```

If the connected Slave is a C200HX PC, then VR(233) will contain value 12 (hex) after successfull execution.

Example 2: The following command will check the Host Link communication with the Host Link Slave (node 23) connected to the RS-422A port.

```
HLM_COMMAND(HLM_TEST,2,23)
PRINT HLM_STATUS PORT(2)
```

If the **HLM_STATUS** parameter contains value zero, the communication is functional.

Example 3: The following two commands will perform the Host Link **INITIALIZE** and **ABORT** operations on the RS-422A port 2. The Slave has node number 4.

```
HLM_COMMAND(HLM_INIT,2)
HLM_COMMAND(HLM_ABORT,2,4)
```

Example 4: When data has to be written to a PC using Host Link, the CPU Unit can not be in **RUN** mode. The **HLM_COMMAND** command can be used to set it to **MONITOR** mode. The Slave has node address 0 and is connected to the RS-232C port.

```
HLM_COMMAND(HLM_STWR,2,0,2)
```

HLM_READ

Type: Hostlink Command

Syntax: **HLM_READ(port,node,pc_area,pc_offset,length,mc_area,mc_offset)**

Description: The **HLM_READ** command reads data from a Host Link Slave by sending a Host Link command string containing the specified node of the Slave to the serial port. The received response data will be written to either VR or Table variables. Each word of data will be transferred to one variable. The maximum data length is 30 words (single frame transfer). Program execution will be paused until the response string has been received or the timeout time has elapsed. The timeout time is specified by using the **HLM_TIMEOUT** parameter. The status of the transfer can be monitored with the **HLM_STATUS** parameter.

Parameters:

port The specified serial port. (See specific controller specification for numbers)

node The Slave node number to send the Host Link command to. Range: [0, 31].

pc_area The PC memory selection for the Host Link command.

pc_area	Data area	Hostlink command
PLC_DM (or value 0)	DM	RD
PLC_IR (or value 1)	CIO/IR	RR
PLC_LR (or value 2)	LR	RL
PLC_HR (or value 3)	HR	RH
PLC_AR (or value 4)	AR	RJ
PLC_EM (or value 6)	EM	RE

pc_offset The address of the specified PC memory area to read from. Range: [0, 9999].

length The number of words of data to be transfered. Range: [1, 30].

mc_area The MC Unit's memory selection to write the received data to.

mc_offset The address of the specified MC Unit memory area to write to.

mc_area	Data area
MC_TABLE (or value 8)	**Table variable array**
MC_VR (or value 9)	**Global (VR) variable array**

Note 1: When using the **HLM_READ**, be sure to set-up the Host Link Master protocol by using the **SETCOM** command.

Note 2: The Host Link Master commands are required to be executed from one program task only to avoid any multi-task timing problems.

HLM_STATUS

Type: System Command.

Description: Returns the status of the Host Link serial communications.

HLM_TIMEOUT

Type: Host Link Command.

Description: Sets the timeout value for Hostlink communications.

Example:
```
HLM-TIMEOUT = 600
```

Note: Default value is 500msec.

HLM_WRITE

Type: Hostlink Command

Syntax: **HLM_WRITE(port,node,pc_area,pc_offset,length,mc_area,mc_offset)**

Description: The **HLM_WRITE** command writes data from the MC Unit to a Host Link Slave by sending a Host Link command string containing the specified node of the Slave to the serial port. The received response data will be written from either VR or Table variables. Each variable will define on word of data which will be transferred. The maximum data length is 29 words (single frame transfer). Program execution will be paused until the response string has been received or the timeout time has elapsed. The timeout time is specified by using the **HLM_TIMEOUT** parameter. The status of the transfer can be monitored with the **HLM_STATUS** parameter.

Parameters:

port The specified serial port. (See specific controller specification for numbers)

node The Slave node number to send the Host Link command to. Range: [0, 31].

pc_area The PC memory selection for the Host Link command.

pc_area	Data area	Hostlink command
PLC_DM (or value 0)	**DM**	**RD**
PLC_IR (or value 1)	**CIO/IR**	**RR**
PLC_LR (or value 2)	**LR**	**RL**
PLC_HR (or value 3)	**HR**	**RH**
PLC_AR (or value 4)	**AR**	**RJ**
PLC_EM (or value 6)	**EM**	**RE**

pc_offset The address of the specified PC memory area to write to. Range: [0, 9999].

length The number of words of data to be transfered. Range: [1, 30].

mc_area The MC Unit's memory selection to read the data from.

mc_offset The address of the specified MC Unit memory area to read from.

mc_area	Data area
MC_TABLE (or value 8)	**Table variable array**
MC_VR (or value 9)	**Global (VR) variable array**

Note 1: When using the **HLM_WRITE**, be sure to set-up the Host Link Master protocol by using the **SETCOM** command.

Note 2: The Host Link Master commands are required to be executed from one program task only to avoid any multi-task timing problems.

Example: The following example shows how to write 25 words from MC Unit's VR addresses 200-224 to the PC EM area addresses 50-74. The PC has Slave node address 28 and is connected to the RS-232C port.

```
HLM_WRITE(1, 28, PLC_EM, 50, 25, MC_VR, 200)
```

HLS_MODEL

Type: Hostlink Parameter

Description: Defines the model number returned to a Hostlink Master. Default value is 250.

HLS_NODE

Type: Hostlink Parameter

Description: Sets the Hostlink node number for the slave node. Used in multidrop RS485 Hostlink networks or set to 0 for RS232 single master/slave link.

INCLUDE

Type: System Command.

Syntax: **INCLUDE "filename"**
(filename - The program to be included).

Description: The **INCLUDE** command resolvies all local variable definitions in the included file at compile time and allows all the local variables to be declared "globally". Whenever an included program is modified, all program that depend on it are re-compiled as well, avoiding inconsistencies.

Example: PROGRAM "T1":

```
       'include global definitions
INCLUDE "GLOBAL_DEFS"
       'Motion commands using defined vars
FORWARD AXIS(drive_axis)
CONNECT(1, drive_axis) AXIS(link_axis)
```

PROGRAM "GLOBAL_DEFS":

```
drive_axis=4
link_axis=1
```

Note: (1) Nested INCLUDEs are not allowed.
(2) The **INCLUDE** command must be the first BASIC statement in the program.
(3) Only variable definitions are allowed in the include file. It cannot be used as a general subroutine with any other BASIC commands in it.
(4) Not available on the MC302 range.

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

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 **SELECT**ed 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.

LIST_GLOBAL

Type: System Command (Terminal only)

Syntax: **LIST_GLOBAL**

Description: When executed from the command line, (terminal channel 0) all the currently set **GLOBAL** and **CONSTANT** parameters will be printed to the terminal.

Example: In an application where the following **GLOBAL** and **CONSTANT** have been set;

```
CONSTANT "cutter", 23
GLOBAL "conveyor",5
>>LIST_GLOBAL
Global           VR
---------------- ----
conveyor 5
Constant         Value
---------------- -------
cutter 23.0000
>>
```

LOAD_PROJECT

Type: System Command

Description: Used by *Motion* Perfect to load projects to the controller.

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 I167.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
MC302X	**MC302Xvnnnnn.s37**
Euro205X	**Knnn.OUT**
Euro209	**Qnnn.OUT**
MC206X	**Mnnn.OUT**
PCI208	**Jnnn.OUT**
MC224	**Innn.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 **LOCK**ed, 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.

MC_TABLE

Type: Reserved Keyword

MC_VR

Type: Reserved Keyword

MOTION_ERROR

Type: System Parameter

Description: This system parameter returns a non-zero value 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
>>
```

N_ANA_OUT

Type: System Parameter (Read Only)

Description: This parameter returns the number of analogue output channels available to the controller

NAIO

Type: System Parameter

Description: Description: This parameter returns the number of analogue input channels available to the Motion Coordinator. For example an MC224 will return 10 if there is 1 x P325 CAN module connected as it has 2 internal analogue inputs and the 8 inputs from the P325.

If no external I/O is fitted, **`NAIO`** returns the number of Analogue inputs within the *Motion Coordinator*.

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

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 MC224 the first 8 channels are inputs, the next 8 bi-directional. If an MC224 has 2 P316 CAN-16 I/O modules connected the **NIO** parameter will return 48.

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

Though normally used as a read-only parameter, **NIO** can be set to any value for simulation purposes. Any I/O read or written that is not physically there, will have no function.

PEEK

Type: System Command

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

Description: The **PEEK** command returns value of a memory location of the controller **AND**ed 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.

PORT

Type: Modifier

Description: Reserved keyword.

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.

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. It can also be set by the Flashstick or by the **trioinit.bas** file on the SD Card.

Note: When using the Memory Stick users should refer to the overview in the MC206X 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 MC206X/Euro205x or the slot number of an MC224.
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

PROTOCOL

Type: System Command

Description: Reserved keyword.

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.

Example: Set port 1 to receive commands from a PC running the TrioPC ActiveX component.

```
'set up RS232 port at 38400 baud, 8 data, 1 stop, even parity
'then start the REMOTE process which will handle the commands
'received from TrioPC.

SETCOM(38400,8,1,2,1,8)
REMOTE(0)
```

RENAME

Type: System Command

Syntax: **RENAME oldname newname**

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.

RS232_SPEED_MODE

Type: System

Syntax: **RS232_SPEED_MODE=modevalue**

Description: Sets the default programming port speed on power-up.

Parameters: 0 = low speed defaults (9600 baud)

1 = high speed defaults (38400 baud)

Cycle the power to the *Motion Coordinator* after setting. High speed mode is shown on power-up by the ok and status LEDs flashing alternately.

Note: This command dhould only be used on the command line terminal.

RUN

Type: System Command

Syntax: **RUN “progname” [, process#[, interrupt#]]**

Description: **Runs a program on the controller.**

Parameters:

program:	Name of program to be run. Must be contained within quotation marks.
process#:	Optional process number. If this is left off, the next available number will be used, starting with the highest.
interrupt#:	Optional value between 0 and 2 to select the exact interrupt slot in the servo cycle that the process will run on. **PROC_MODE** must be set to 1 to use this parameter.

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: e.g. **RUN "CYCLE"**

Example: **RUN** - this will run currently selected program, normally used in the terminal.

Example 2: **RUN “SAUSSAGE”** - this will run the named program, normally used in the terminal

Example 3: **RUN “SAUSSAGE”,3** - run the named program on a particular process, normally used in the terminal

Example 4:
```
RUN “MAIN”,1,2   'run 2 programs in the same interrupt slot.
RUN “HMI”,2,2
RUN “MOTION”,1,1 'run motion in it’s own interrupt slot.
```

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 “progname” to run automatically on power up on process 10

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

Note 1: 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

Syntax: **SCOPE(control,period,table start,table stop,p0[,p1[,p2[,p3]]])**

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 1: 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.

SERCOS

Type: System Function

Syntax: **SERCOS(function#,slot,{parameters})**

Description: This function allows the SERCOS ring to be controlled from the Trio BASIC programming system. A SERCOS ring consists of a single master and 1 or more slaves daisy-chained together using fibre-optic cable. During initialisation the ring passes through several ‘communication phases’ before entering the final cyclic deterministic phase in which motion control is possible. In the final phase, the master transmits control information and the slaves transmit status feedback information every cycle time.

Once the SERCOS ring is running in CP4, the standard Trio BASIC motion commands can be used.

The *Motion Coordinator* SERCOS hardware uses the Sercon 816 SERCOS interface chip which allows connection speeds up to 16Mhz. This chip can be programmed at a register level using the **SERCOS** command if necessary. To program in this way it is necessary to obtain a copy of the chip data sheet.

The **SERCOS** command provides access to 11 separate functions:

Slot:	The slot number is in the range 0 to 3 and specifies the hardware channel	
Function:	0	Read SERCOS Asic:
	1	Write SERCOS Asic:
	2	Initialise command:
	3	Link SERCOS drive to Axis
	4	Read parameter
	5	Write parameter
	6	Run SERCOS procedure command
	7	Check for drive present
	8	Print network parameter
	9	Reserved
	10	SERCOS ring status

Parameters:

Function 0	SERCOS(0, slot, ram/reg, address)
Slot	The communication slot in which the SERCOS is fitted.
ram/reg	0 = read value from RAM 1 = read value from register.
address	The index address in RAM or register.

Example:

```
>>?SERCOS(0, 0, 1, $0c)
```

Parameters:

Function 1	SERCOS(1, slot, ram/reg, address, value)
Slot	The communication slot in which the SERCOS is fitted.
ram/reg	0 = write value to RAM 1 = write value to register.
address	The index address in RAM or register.
value	Date to be written

Example: Do not use this function without referencing the Sercon 816 data sheet.

Parameters:

Function 2	SERCOS(2, slot [,intensity [,baudrate [, period]]])
Slot	The communication slot in which the SERCOS is fitted.
intensity	Light transmission intensity (1 to 6). Default value is 3.
baudrate	Communication data rate. Set to 2, 4, 6, 8 or 16.
period	Sercos cycle time in microseconds. Accepted values are 2000, 1000, 500 and 250usec.

Example: **>>SERCOS(2, 3, 4, 16, 500)**

Parameters:

Function 3	SERCOS(3, slot, slave addr, axis [slave drive type])
Slot	The communication slot in which the SERCOS is fitted.
slave addr	Slave address of drive to be linked to an axis.
axis	Axis number which will be used to control this drive.
slave drive type	Optional parameter to set the slave drive type. All standard SERCOS drives require the GENERIC setting. The other options below are only required when the drive is using non-standard SERCOS functions. 0 Generic Drive 1 Sanyo-Denki 3 Yaskawa + Trio P730 4 PacSci 5 Kollmorgen

Example: **>>SERCOS(3, 1, 3, 5, 0) ‘links drive at address 3 to axis 5**

Parameters:

Function 4	SERCOS(4, slot, slave address, parameter ID [, parameter size[, element type [, list length offset, [VR start index]]])
Slot	The communication slot in which the SERCOS is fitted.
slave addr	SERCOS address of drive to be read.
parameter ID	SERCOS parameter IDN

parameter size	Size of parameter data expected: 2 = 2 byte parameter (default). 4 = 4 byte parameter 6 = list of parameter IDs 7 = ASCII string
element type	SERCOS element type in the data block: 1 ID number 2 Name 3 Attribute 4 Units 5 Minimum Input value 6 Maximum Input value 7 Operational data (default)
List length offset	Optional parameter to offset the list length. For drives that return 2 extra bytes, use -2.
VR start index	Beginning of VR array where list will be stored.

Note: This function returns the value of 2 and 4 byte parameters but prints lists to the terminal in *Motion* Perfect unless VR start index is defined.

Example:

```
>>SERCOS(4, 0, 5, 140, 7)'request "controller type"
>>SERCOS(4, 0, 5, 129)   'request manufacturer class 1 diagnostic
```

Parameters:

Function 5	SERCOS(5, slot , slave address, parameter ID, parameter size, parameter value [, parameter value ...])
Slot	The communication slot in which the SERCOS is fitted.
slave addr	SERCOS address of drive to be written.
parameter ID	SERCOS parameter IDN
parameter size	Size of parameter data to be written. 2, 4, or 6.
parameter value	Enter one parameter for size 2 and size 4. Enter 2 to 7 parameters for size 6 (list).

Example:

```
>>SERCOS(5, 1, 7, 2, 2, 1000)      'set SERCOS cycle time
>>SERCOS(5, 0, 2, 16, 6, 51, 130)  'set IDN 16 position feedback
```

Parameters:		
	Function 6	SERCOS(6, slot , slave address, parameter ID [, time-out,[command type]])
	Slot	The communication slot in which the SERCOS is fitted.
	slave addr	SERCOS address of drive.
	parameter ID	SERCOS procedure command IDN.
	time out	Optional time out setting (msec).
	command type	Optional parameter to define the operation: -1 Run & cancel operation (default value) 0 Cancel command 1 Run command

Example:
```
>>SERCOS(6, 0, 2, 99)  'clear drive errors
```

Parameters:		
	Function 7	SERCOS(7 , slot , slave address)
	Slot	The communication slot in which the SERCOS is fitted.
	slave addr	SERCOS address of drive. Returns 1 if drive detected, -1 if not detected.

Example:
```
IF SERCOS(7, 2, 3) <0 THEN
     PRINT#5, "Drive 3 on slot 2 not detected"
END IF
```

Parameters:		
	Function 8	SERCOS(8 , slot , required parameter)
	Slot	The communication slot in which the SERCOS is fitted.
	required parameter	This function will print the required network parameter, where the possible 'required parameter' values are: 0: to print a semi-colon delimited list of 'slave Id, axis number' pairs for the registered network configuration (as defined using function 3). Used in Phase 1: Returns 1 if drive is detected, 0 if no drive detected. 1: to print the baud rate (either 2, 4, 6, or 8), and 2: to print the intensity (a number between 0 and 6).

Example:
```
>>?SERCOS(8,0, 1 )
```

Parameters:

Function 10	SERCOS(10,<slot>)
Slot	The communication slot in which the SERCOS is fitted.

This function checks whether the fibre optic loop is closed in phase 0. Return value is 1 if network is closed, -1 if it is open, and -2 if there is excessive distortion on the network.

Example:

```
>>?SERCOS(10, 1)
IF SERCOS (10, 0) <> 1 THEN
  PRINT "SERCOS ring is open or distorted"
END IF
```

Notes: MotionPerfect2 contains support for commissioning SERCOS rings. This tool simplifies the creation of a Trio BASIC startup program which consists of SERCOS statements to initialise the ring following power-on, and configure the ring in the deterministic cyclic phase.

SERCOS_PHASE

Type: System Parameter

Syntax: **SERCOS_PHASE SLOT(n) = value**

Description: Sets the phase for the sercos ring attached to the daughter board in slot n.

Example 1: Set the sercos ring attached to daughter board in slot 0 to phase 3

```
SERCOS_PHASE SLOT(0) = 3
```

Example 2: Check the phase of sercos ring attached to daughter board in slot 2

```
IF SERCOS_PHASE SLOT(2)<>4 THEN OP(8,ON)
```

SERIAL_NUMBER

Type: System Parameter (Read only)

Syntax: **SERIAL_NUMBER**

Description: Returns the unique Serial Number of the controller.

Example: For a controller with serial number 00325:

```
>>PRINT SERIAL_NUMBER
325.0000
>>
```

SERVO_PERIOD

Type: System Parameter

Description: This parameter allows the controller servo period to be specified.

SERVO_PERIOD is specified in microseconds. Only the values 2000, 1000, 500 or 250 usec may be used and the *Motion Coordinator* must be reset before the new servo period will be applied.

Example:

```
' check controller servo_period on startup
IF SERVO_PERIOD<>250 THEN
  SERVO_PERIOD=250
  EX
ENDIF
```

SLOT

Type: Slot Modifier

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

Example: **PRINT COMMSTYPE SLOT(1)**

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
```

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 **SELECT**ed 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**

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, [process_number]
```

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

STICK_READ

Type: System Function

(A) Flashstick fitted

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

Description: Copy one block of 128 values from a sector on the NexFlash FlashStick to TABLE memory.

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"
```

(B) SD Card fitted

Syntax: **STICK_READ(<flash_file#>,<table_start>[,<format>])**

Description: If an SDCARD is detected then the file SD<flash_file#>.BIN or SD<Flash_file#.CSV> is opened. All the binary data in the file is read into TABLE memory. By default, if the format parameter is left off, the data is read in IEEE floating point binary format, little-endian, i.e. the least significant byte first.

Perameters:

flash_file#: A number which when appended to the characters "SD" will form the data filename.

table start: The start point in the TABLE where the data values will be transferred to.

format: 0 = Binary floating point format
1 = ASCII comma seperated values

Example:

```
STICK_READ (1984, 16500, 1)
'reads the ASCII file SD1984.csv from the SD card and copies the
'data to the table starting at TABLE(16500)
```

The function returns TRUE (-1) if the **STICK_READ** was successful and FALSE (0) if the command failed, if for example the FlashStick or SD Card is not present.

STICK_WRITE

Type: System Function

(A) Flashstick fitted

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

Description: Copy one block of 128 values from TABLE memory to a sector on the NexFlash Flash-Stick.

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:

```
STICK_WRITE (25, 1000)
IF check = TRUE THEN PRINT "stick write ok"
```

(B) SD Card fitted

Syntax: **STICK_WRITE(<flash_file#>,<table_start>[,<length>[,<format>]])**

Description: If an SDCARD is detected then the file **SD<flash_file#>.BIN** or **SD<Flash_file#.CSV>** is created. If this file already exists, it is overwritten.

If no format is specified, or <format>=0 then the data is stored in IEEE floating point binary format little-endian, i.e. the least significant byte first, and has the extension "BIN".

If **<format>** is specified and non 0 then the data is stored in ASCII format and has the extension "CSV", one value per line.

Parameters:

flash_file#:	A number which when appended to the characters “SD” will form the data filename.
table start:	The start point in the TABLE where the values will be transferred from.
length:	The number of the table values to be transferred.
format:	0 = Binary floating point format 1 = ASCII comma seperated values

Example:

```
STICK_WRITE (1501, 1000, 2000,0)
'transfers 2000 values starting at TABLE(1000) to the SD Card file
'called SD1501.BIN
```

The function returns TRUE (-1) if the **STICK_WRITE** was successful and FALSE (0) if the command failed, if for example the FlashStickor SD Card is not present.

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.

SYNC_TIMER

Type: System Parameter (Write Only, MC224 Only)

Syntax: **SYNC_TIMER = value**

Description: **SYNC_TIMER** is a system parameter automatically set by the controller. In normal use it should not be adjusted by the user, but in some cases with the MC224 and the P225 (analogue input daughter board) it needs to be changed. Please contact your Trio Distributor or Trio directly if you need to use this parameter.

Example:

```
' set last axis and sync_timer
SPEED AXIS(23)=2000 'write to an axis parameter to set LAST_AXIS
to 23
SYNC_TIMER=12850 'This value is for example only (MC224 V1.67)
```

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 32000 points on all *Motion Coordinator*s EXCEPT the MC302X which has a 16000 point table length and the MC224 which has 256k. 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 MC302X 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 (MC224 only)

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

Example 1: **Sets the real time clock in 24 hour format; hh:mm:ss**

```
  'Set the real time clock
>>TIME = 13:20:00
```

Example 2:

```
  'calculate elapsed time in seconds
time1 = TIME
  'wait for event
time2 = TIME
timeelapsed = time1-time2
```

TIME$

Type: System Command (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 started by a previous **TRON** command, 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"** (Not applicable to MC224)

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. 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 MC206X/MC224 or the slot number of a Euro205x.
function:	Specifies the function to be performed. 0: Read register 1: Write register 2: Open / initialise USB chip 3: Close USB port
register:	The register number to read or write
value:	The value to write into a register

Example:

```
USB(1, 3) 'manually reset the USB port
WA(200)
USB(1, 2)
```

USB_HEARTBEAT

Type: System Parameter

Description: Indicates that the USB Heartbeat function is operating. When the value is 1, the heartbeat is running and if no data is received via the USB link then after 60 seconds, the USB port in the controller will be reset automatically.

The value defaults to 0 on power-up and is automatically set to 1 when a PC opens the USB connection. The user can disable the heartbeat function by manually setting the value to 0 again.

Example 1:

```
'test to see if USB port is open and heartbeat is running
          IF USB_HEARTBEAT=1 THEN
            PRINT "USB port is in use"
          ENDIF
```

Example 2:

```
'turn off the usb heartbeat function from the terminal
          >>USB_HEARTBEAT = 0
```

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
   1.5000
```

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 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: Variable

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 1024 variable locations which are accessed as variables 0 to 250.

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 MC302X) 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 ‘val’ 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)
```

VRSTRING

Type: Command

Syntax: **VRSTRING(vr start)**

Description: Combines the contents of an array of VR() variables so that they can be printed as a text string. All printable characters will be output and the string will terminate at the first null character found. (i.e. VR(n) contains 0)

Parameters:

vr start: number of first VR() in the character array.

Example:
```
PRINT #5,VRSTRING(100)
```

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 **WDOG** setting and turn watchdog contact OFF. In addition the analogue outputs and step/direction outputs are also disabled when **WDOG=OFF**.

Example:
```
WDOG=ON
```

Note 1: **WDOG=ON / WDOG=OFF** is issued automatically by *Motion* Perfect when the "Drives Enable" button is clicked on the control panel

Note 2: When the **DISABLE_GROUP** function is in use, the watchdog relay and WDOG remain on if there is an axis error. In this case, the digital enable signal is removed from the drives in that group only.

WDOGB

Type: System Parameter

Syntax: **WDOGB=state**

Description: Controls the second "watchdog" relay contact on the MC224. See **WDOG** for more details.

Parameters: State

-1	WDOGB follows the state of WDOG
0	WDOGB is OFF
1	WDOGB is ON

Example:
```
WDOGB=OFF' Disconnects the second WDOG relay from the first and
sets ' it state to OFF
WDOG=ON' Turns ON the first WDOG (WDOG A)
```

:

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 100 characters per line (79 in system software V1.66 and lower).

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.

Example 1:
```
VR(10)=$8F3B
OP($CC00)
Process Parameters and Commands
```

BITREV8

Type: Mathematical function

Syntax: **BITREV8(byte)**

Description: The BITREV8 function reverses the order of the lowest 8 bits in a variable.

Parameters: **byte** Any variable in which you want to reverse the lowest 8 bits.

Example:
```
byte_in = $a3
byte_out = BITREV8(byte_in)
PRINT "Result = ";HEX(byte_out)

Result = c5
```

Note: MC302X, MC302-K only

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
4: Write

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)
```

PROC_MODE

Type: Process Parameter

Description: Enables user control of processes and interupt slot numbers with the extended RUN command.

Example:
```
PROC_MODE(0)     'set “standard” multi-tasking control
                '(compatible with older system software versions)
PROC_MODE(1)     'set up advanced multi-tasking control
RUN “prog1”, 4,0 'prog 1 runs as process 4 sharing the same
RUN “prog2”, 5,0 'interrupt slot.
```

PROC_STATUS

Type: Process Parameter (Read Only)

Description: Returns the status of another process, referenced with the **PROC(x)** modifier.

Returns

0	Process Stopped
1	Process Running
2	Process Stepping
3	Process Paused

Example:
```
RUN "progname",12
WA(100)                ' wait for program to start
WAIT UNTIL PROC_STATUS PROC(12)=0
                       ' Program "progname" has now finished.
```

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)
```

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
```

SHIFTR

Type: Mathematical Function

Syntax: **SHIFTR(variable, n)**

Description: Shifts the bits in a variable to the right by 'n' number of times.

Parameters:
variable Any local variable or VR containing the value to be shifted.
n: number of times to shift the value.

Example:
```
'Convert a 16 bit word to 2 bytes by dividing the MSByte by 256
msbyte = SHIFTR(word, 8)
lsbyte = word AND $ff
```

Notes: MC302X, MC302-K only (ARM processor based)

Use normal divide operator in MC2xx Motion Coordinators. DSP processor executes the divide as quickly as a shift function.

STRTOD

Type: Function

Syntax: **STRTOD(format,...)**

Description: Converts a string into a decimal number.

Parameters:

STRTOD(0,start,end)

Read string starting at **VR(start)** and parse number until it finds a non floating point character. **VR(end)** will contain the index of the character which stops the parsing. The number format accepted here is as follows:

<number> ::= [<sign>]<integer>[<fraction>][<exponent>]

<sign> ::= +|-

<integer> ::= <digit> | <integer> <digit>

<digit> ::= '0'|'1'|'2'|'3'|'4'|'5'|'7'|'8'|'9'

<fraction> ::= '.' <integer>

<exponent> ::= 'E' [<sign>][<integer>]

STRTOD(1,channel,count,terminator)

Read string from the specified channel and parse number until it finds a non floating point character. **VR(count)** will contain the number of characters accepted. **VR(terminator)** will contain the character that terminates the parsing. The number format accepted here is the same as **STRTOD(0)**.

STRTOD(2,start,end) Read string starting at **VR(start)** and parse number until it finds a non integer character or a number that cannot be represented as a 32 bit integer. **VR(end)** will contain the index of the character which stops the parsing. The number format accepted here is as follows:

```
<number> ::=  [<sign>]<integer>

<sign> ::= +|-

<integer> ::= <digit> | <integer> <digit>

<digit> ::= '0'|'1'|'2'|'3'|'4'|'5'|'7'|'8'|'9'
```

STRTOD(4,start,end) Read string starting at **VR(start)** and parse number until it finds a non floating point character. If the number can be represented as a 32 bit integer then integer maths is used, otherwise floating point maths is used. **VR(end)** will contain the index of the character which stops the parsing. The number format accepted here the same as **STRTOD(0)**. This avoids precision errors inherent in floating point calculations.

TABLE_POINTER

Type: Axis Parameter(Read Only)

Syntax: **value=TABLE_POINTER**
Where value is returned of type X.Y where X is the current TABLE location and Y represents the interpolated distance between the start and end location of the current TABLE location.

Description: The ability to adjust a CAM based profiles from within the Trio BASIC program adds more flexibility to Trio's Motion Coordinators. Using the **TABLE_POINTER** command it is possible to determine which TABLE memory location is currently being used by the CAM allowing the user to load new CAM data into previously processed TABLE location ready for the next CAM cycle. This is ideal for allowing a technician to finely tune a complex process, or changing recipes on the fly whilst running.
TABLE_POINTER returns the current table location that the CAM function is using. The returned number contains the table location and divides up the interpolated distance between the current and next TABLE location to indicate exact location.

Example: In this example a CAM profile is loaded into TABLE location 1000 and is setup on axis 0 and is linked to a master axis 1. A copy of the CAM table is added at location 100. The Analogue input is then read and the CAM TABLE value is updated when the table pointer is on the next value.

```
' CAM Pointer demo
' store the live table points
TABLE(1000,0,0.8808,6.5485,19.5501,39.001,60.999,80.4499,93.4515)
TABLE(1008,99.1192,100)
' Store another copy of original points
TABLE(100,0,0.8808,6.5485,19.5501,39.001,60.999,80.4499,93.4515)
TABLE(108,99.1192,100)
' Initialise axes
BASE(0)
WDOG=ON
SERVO=ON

' Set up CAM
CAMBOX(1000,1009,10,100,1, 4, 0)

' Start Master axis
BASE(1)
SERVO=ON
SPEED=10
FORWARD

' Read Analog input and scale CAM based on input
pointer=0
WHILE 1
' Read Analog Input (Answer 0-10)
scale=AIN(32)*0.01
' Detects change in table pointer
IF INT(TABLE_POINTER)<>pointer THEN
    pointer=INT(TABLE_POINTER)
    ' First value so update last value
    IF pointer=1000 THEN
        TABLE(1008,(TABLE(108)*scale))
    ' Second Value, so must update First & Last but 1 value
    ELSEIF pointer=1001 THEN
        TABLE(1000,(TABLE(100)*scale))
        TABLE(1009,(TABLE(109)*scale))
    ' Update previous value
    ELSE
        TABLE(pointer-1, (TABLE(pointer-901)*scale))
    ENDIF
ENDIF
```

```
WEND
STOP
```

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

^ Power

Type: Arithmetic operation

Syntax **<expression1> ^ <expression2>**

Description: Raises expression1 to the power of expression2

Parameters:

Expression1: Any valid Trio BASIC expression

Expression2: Any valid Trio BASIC expression

Example:

```
x=2^6
PRINT x
```

Trio BASIC raises the first number (2) to the power of the second number (6). Therefore x has the value of 64

= 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 analogue 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 "Analogue 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
```

B_SPLINE

Type: Command

Syntax: **B_SPLINE(type, {parameters})**

Description: This function expands data to generate higher resolution motion profiles. It operates in two modes using either B Spline or Non Uniform Rational B Spline (NURBS) mathematical methods.

Syntax: **B_SPLINE(1, data_in, #in, data_out, #expand)**

Description: Expands an existing profile stored in the TABLE area using the B Spline mathematical function. The expansion factor is configurable and the **B_SPLINE** stores the expanded profile to another area in the TABLE.

This is ideally used where the source CAM profile is too coarse and needs to be extrapolated into a greater number of points.

Parameters:

type	1 Standard B-Spline
data_in	Location in the TABLE where the source profile is stored.
#in	Number of points in the source profile.
data_out	Location in the TABLE where the expanded profile will be stored.
#expand	The expansion ratio of the **B_SPLINE** function. (i.e. if the source profile is 100 points and #expand is set to 10 the resulting profile will be 1000 point (100 * 10).

Example:

```
B_SPLINE(1,0,10,200,10)
```

Expands a 10 point profile in TABLE locations 0 to 9 to a larger 100 point profile starting at TABLE address 200.

Syntax:

```
B_SPLINE(2, dimen, Curve_type, weight_op, points, knots, expansion, in_data, out_data)
```

Description: Non Uniform Rational B-Splines, commonly referred to as NURBS, have become the industry standard way of representing geometric surface information designed by a CAD system. NURBS is the basis behind many 3D files such as IGES, STEP and PHIGS.

NURBS provide a unified mathematical basis for representing analytic shapes such as conic sections and quadratic surfaces, as well as free form entities, such as car bodies and ship hulls. NURBS are small for data portability and can be scaled to increase the number of target points along a curve, increasing accuracy. A series of NURBS are used to describe a complex shape or surface.

NURBS are represented as a series of XYZ points with knots + weightings of the knots.

Parameters:

type	2 Non Uniform Rational B-Spline.
Dimen	Defines the number of axes. Reserved for future use must be 3.
Curve_type	Classification of the type of NURBS curve. Reserved for future use must be 3.
Weight_op	Sets the weighting of the knots 0=All weighting set to 1.
points	Number of data points.
knots	Number of knots defined.

expansion	Defines the number of points the expanded curve will have in the table. Total output points = Number of points * expansion. Minimum value = 3.
in_data	Location of input data. Data is stored with X0,Y0,Z0,X1,Y1,Z1...,followed by knots data N0, N1, N2 ...
Out_data	Table start location for output points stored X0, Y0, Z0 etc.

Example:

```
type=2                   '2 for NURBS
dimen=3                  'must be 3 at present (X Y Z)
curve_type=3             'XYZ axes
weight_op=0              '0 sets all weights to 1.0
points=9                 'number of data points
knots=13                 'number of knots
expansion=5              'Expansion factor
in_data=100              'data points
out_data=1000            'table location to construct output

' Data Points:
TABLE(100,150.709,353.8857,0)
TABLE(103,104.5196,337.7142,0)
TABLE(106,320.1131,499.4647,0)
TABLE(109,449.4824,396.4945,0)
TABLE(112,595.3350,136.4910,0)
TABLE(115,156.816,96.3351,0)
TABLE(118,429.4556,313.7982,0)
TABLE(121,213.3019,375.8004,0)
TABLE(124,150.709,353.8857,0)

' Knots:
TABLE(127,0,0,0,0,146.8154,325.6644,536.0555,763.4151)
TABLE(135,910.13,38,1109.08861109.0886,1109.0886,1109.0886)

'Expand the curve, generate 5*9=45 XYZ points
'or 137 table locations
B_SPLINE(type,dimen,curve_type,weight_op,points,knots,
  expansion,in_data,out_data)
```

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.

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 (64 constants in MC302-K).

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
```

CRC16

Type: Command

Syntax: **RESULT=CRC16(MODE, POLY/DATA_SOURCE, START, END, REG)**

Mode 0: CRC16(0, POLY)

Mode 1:CRC16(1, DATA_SOURCE, START, END, REG)

Description: Calculates a 16 bit CRC

Calculates the 16 bit CRC of data stored in contiguous Table Memory or VR Memory locations.

Parameters:

MODE:	Specifies the mode of the command 0 – Initialises the command with the Polynomial 1 – Returns the CRC in RESULT. Will return 0 if Initialise has not been run
POLY:	Polynomial used as seed for CRC check range 0-65535 (or 0-$FFFF)
DATA_SOURCE:	Defines where the data is loaded 0 – Table Memory 1 – VR Memory
START:	Start location of first byte
END:	End Location of last byte
REG:	Initial CRC value. Normally $0 - $FFFF

Examples: Using Table Memory:

```
poly = $90d9
reginit = $ffff
CRC16(0, poly) 'Initialise internal CRC table memory
TABLE(0,1,2,3,4,5,6,7,8) 'Load data into table memory location 0-7
calc_crc = CRC16(1,0,0,7,reginit) 'Source Data=TABLE(0..7)
```

Using VR Memory:

```
poly = $90d9
reginit = $ffff
CRC16(0, poly) 'Initialise internal CRC table memory
'Load 6 bytes into VR memory location 0-5
for i=0 to 5
   VR(i)=i+1
Next i
calc_crc = CRC16(1,1,0,5,reginit) 'Source Data=VR(0)..VR(5)
```

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.

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 “screw_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 (64 constants in MC302-K).

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.

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.

Parameters:

value:	Any Trio BASIC floating point variable or parameter.
n:	The byte number (0 - 3) to be extracted.

Example:

```
a = MPOS AXIS(2)
byte0 = IEEE_OUT(a, 0)
byte1 = IEEE_OUT(a, 1)
byte2 = IEEE_OUT(a, 2)
byte3 = IEEE_OUT(a, 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)

INTEGER_READ/INTEGER_WRITE

Type: Command

Syntax: **INTEGER_READ(<source>,<least_significant>,<most_significant>)**
INTEGER_WRITE(<destination>,<least_significant>,<most_significant>)

Description: TrioBASIC handles all numbers in 32 bit floating point format. The 32 bit format has 1 bit sign, 8 bit exponent and 23 bit mantissa with an implied most significant bit. This means that the maximum integer resolution is 24 bits. For most applications this is sufficient, but for applications with high precision encoders very quickly we can get beyond this 24 bit limit.

The **INTEGER_READ/INTEGER_WRITE** functions work around this limitation by performing a low level access to the 32 bit register splitting it into 2 16 bit segments.

Parameters:

<source>	2 bit value that will be read, can be VR, TABLE, or system variable.
<destination>	32 bit value that will be written, can be VR, TABLE, or system variable.
<least_significant>	Least significant (rightmost) 16 bits, can be any valid TrioBASIC expression.
<most_significant>	Most significant (leftmost) 16 bits, can be any valid TrioBASIC expression.

LN

Type: Mathematical Function

Syntax: **LN(expression)**

Description: Returns the natural logarithm of the expression.

Parameter: **expression:** Any valid Trio BASIC expression.

MOD

expression: Any valid Trio BASIC expression.

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 exclusive or 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

OFF

Type: Constant

Description: **OFF** returns the value 0

Example:
```
IF IN(56)=OFF THEN GOSUB label
'run subroutine label 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.
```

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
```

PI

Type: Constant

Description: **PI** is the circumference/diameter constant of approximately 3.14159

Example:
```
circum=100
PRINT "Radius=";circum/(2*PI)
```

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
```

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.

AFF_GAIN

Type: Axis Parameter

Syntax: **AFF_GAIN = value**

Description: Sets the acceleration Feed Forward for the axis. This is a multiplying factor which is applied to the rate of change of demand speed. The result is summed to the control loop output to give the **DAC_OUT** value.

Note: **AFF_GAIN** is only effective in systems with very high counts per revolution in the feedback. I.e. 65536 counts per rev or greater.

ATYPE

Type: Axis Parameter

Description: The **ATYPE** axis parameter indicates the type of axis fitted. On daughter board based axes, the **ATYPE** axis parameter is set by the system software at power up.

Controllers that use Feature Enable Codes to activate axes, such as the Euro205x and MC206X, have the **ATYPE** of each axis set by the system software depending on the Enabled Features on that *Motion Coordinator*. The **ATYPE** of Remote Axes must be set during initialisation in a suitable Trio BASIC program. e.g. STARTUP.BAS.

On the MC302X the **ATYPE** parameter must be set to select the axis function.

#	Description
0	No axis daughter board fitted
1	Stepper daughter board
2	Servo daughter board
3	Encoder daughter board
4	Stepper daughter with position verification / Differential Stepper
5	Resolver daughter board
6	Voltage output daughter board
7	Absolute SSI servo daughter board
8	CAN daughter board
9	Remote CAN axis
10	PSWITCH daughter board
11	Remote SLM axis
12	Enhanced servo daughter board
13	Embedded axis
14	Encoder output
15	Trio CAN
16	Remote SERCOS speed axis
17	Remote SERCOS position axis
18	Remote CANOpen position axis

#	Description
19	Remote CANOpen speed axis
20	Remote PLM axis
21	Remote user specific CAN axis
22	Remote SERCOS speed + registration axis
23	Remote SERCOS position + registration axis
24	SERCOS torque
25	SERCOS speed open
26	CAN 402 position mode
27	CAN 402 velocity mode
30	Remote Analog Feedback axis
31	Tamagawa absolute encoder + stepper
32	Tamagawa absolute encoder + servo
33	EnDat absolute encoder + stepper
34	EnDat absolute encoder + servo
35	PWM stepper
36	PWM servo
37	Step z
38	MTX dual port RAM
39	Empty
40	Trajexia Mechatrolink
41	Mechatrolink speed
42	Mechatrolink torque
43	Stepper 32
44	Servo 32
45	Step out 32
46	Tamagawa 32
47	Endat 32

#	Description
48	SSI 32
49	Mechatrolink servo inverter

Note: Some ATYPEs are not available on all products.

Example:
```
>>PRINT ATYPE AXIS(2)
1.0000
```
This would show that an stepper daughter board is fitted in this axis slot.

```
ATYPE AXIS(20)=16
```
Sets axis 20 to be a remote SERCOS speed axis. (This feature must be enabled with the correct Feature Enable Code first)

```
ATYPE AXIS(0)=4
```
Sets axis 0 to be a stepper with encoder verification axis on the MC302X.

AXIS_ADDRESS

Type: Axis Parameter

Description: The **AXIS_ADDRESS** axis parameter is used when control is being made of remote servo drives with SERCOS or CANOpen communications, or if an analogue input is used for feedback. The **AXIS_ADDRESS** holds the address of the remote servo drive or the AIN number of the analogue input to be used for feedback.

Note: Remote axes will require a Feature Enable Code to be entered before the remote axis can be used. When a SERCOS or CAN daughter board is fitted, 2 remote axes are enabled automatically.

AXIS_ENABLE

Type: Axis Parameter

Syntax: **AXIS_ENABLE = (ON/OFF)**

Description: Used when independent axis enabling is required with either SERCOS or MECHATRO-LINK. This parameter can be set ON or OFF for each axis individually. The default value is ON to maintain compatibility with earlier versions. The axis 'x' will be enabled if **AXIS_ENABLE AXIS(x) = ON and WDOG = ON.**

Note 1: **MOTION_ERROR** now returns a bit pattern showing the axes which have a motion error. i.e. if axes 2 and 5 have an error, the **MOTION_ERROR** value would be 40. (32+8)

Note 2: Both **WDOG** (non axis specific) & **AXIS_ENABLE** (axis specific) must be set ON for the axis to be enabled. If an axis has not been included in a **DISABLE_GROUP** and an error occurs on that axis, **WDOG** will be set OFF.

AXIS_MODE

Type: Axis Parameter

Syntax: **AXIS_MODE=value**

Description: Depending on the bit set this command applies special functions to the axis.

Parameters: **Value**

Bit 0 When set it allows Euro205X to use the O/C outputs and encoder inputs at the same time.

Bit 1 When set it changes the action of end limits on a **CONNECT** command; sets ratio to 0 instead of cancelling the command.

Bit 2 Reserved

Bit 3 Reserved

Bit 4 Reserved

Bit 5 Inverts fault input on Euro205X.

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**

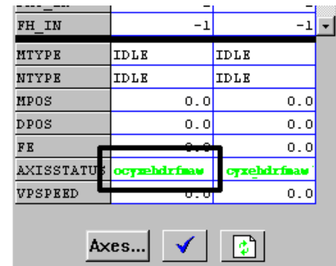
Bit	Description	Value	char
3	Remote drive error	8	**m**
4	In forward limit	16	**f**
5	In reverse limit	32	**r**
6	Datuming	64	**d**
7	Feedhold	128	**h**
8	Following error exceeds limit	256	**e**
9	In forward software limit	512	**x**
10	In reverse software limit	1024	**y**
11	Cancelling move	2048	**c**
12	Encoder power supply overload (MC206X)	4096	**o**
13	Set on SSI axis after initialisation	8192	
14	Status of FAULT input	16384	

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, DATUM(0)**

BACKLASH_DIST

Type: Axis Parameter

Syntax: **value = BACKLASH_DIST**

Description: Amount of backlash compensation that is being applied to the axis when **BACKLASH** is on.

Example:
```
IF BACKLASH_DIST>100 THEN
  OP (10, ON)  'show that backlash compensation has reached
               'this value
```

```
ELSE
  OP (10, OFF)
END IF
```

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(11)=ON`

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.`

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)
```

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 K_d and a change in following error d_e the contribution to the output signal is:

$$O_d = K_d \times \delta_e$$

Example: **D_GAIN=0.25**

D_ZONE_MIN

Type: Axis Parameter

Description: For Piezo Motor Control. This sets works in conjunction with **D_ZONE_MAX** to clamp the DAC output to zero when the demand movement is complete and the magnitude of the following error is less than the **D_ZONE_MIN** value. The servo loop will be reactivated when either the following error rises above the **D_ZONE_MAX** value, or a fresh movement is started.

Example:
```
D_ZONE_MIN = 3
D_ZONE_MAX = 10
```

With these 2 parameters set as above, the DAC output will be clamped at zero when the movement is complete and the following error falls below 3. When a movement is restarted or if the following error rises above a value of 10, the servo loop will be reactivated.

D_ZONE_MAX

Type: Axis Parameter

Description: This sets works in conjunction with **D_ZONE_MIN** to clamp the DAC output to zero when the demand movement is complete and the magnitude of the following error is less than the **D_ZONE_MIN** value. The servo loop will be reactivated when either the following error rises above the **D_ZONE_MAX** value, or a fresh movement is started.

Example:
```
D_ZONE_MIN = 3
D_ZONE_MAX = 10
```

With these 2 parameters set as above, the DAC output will be clamped at zero when the movement is complete and the following error falls below 3. When a movement is restarted or if the following error rises above a value of 10, the servo loop will be reactivated.

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 can take is:

DAC=32767 corresponds to a voltage of 10V
to
DAC=-32768 corresponds to a voltage of -10v

Note: See **DAC_SCALE** for a list of DAC types.

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 an integer multiplier which is applied between the control loop output and the Digital to Analog converter. **DAC_SCALE** can be set to value 16 on axes with a 16 bit DAC. This scales the values applied to the higher resolution DAC so that the gains required on the axis are similar to those required on axes with a 12 bit DAC.

DAC_SCALE may be set negative 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**.

Example: **DAC_SCALE AXIS(3)=-16**

Product	DAC Size	Default DAC_SCALE
P200 Servo DB	12 bit	1
P270 SSI Servo DB	12 bit	1
P201 Enhanced Servo DB	16 bit	1
P136 MC206X	16 bit	16
P156 Euro205x	12 bit	1
P184 / P185 PCI208	16 bit	16

Note: To obtain true 16 bit output with a 16 bit D to A converter, the **DAC_SCALE** must be set to 1 or -1 and the loop gains increased by a factor of 16 compared to those used on an equivalent 12 bit axis.

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..63, except in the PCI208 which has 0..31. If **DATUM_IN** is set to -1 (default) then no input is used as a datum.

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)`

DEMAND_SPEED

Type: Axis Parameter (Read Only)

Description: Returns the speed output of the UPU in edges or counts per millisecond. Normally used for low level debug of the motion system.

Example:
```
>>?DEMAND_SPEED
5.0000
```

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_CLEAR

Type: Axis Function

Syntax: **DRIVE_CLEAR**

Description: Reset and clear the local drive and clear the drive fault flags. Trio “Drive-In” module only. **DRIVE_CLEAR** will run the drive’s own error reset procedure so that if the external conditions allow, the drive will then be ready to run.

Example:
```
WHILE TRUE'Error Handler Program
     IF AXISSTATUS=256 OR AXISSTATUS=258 THEN  'Check for FE fault
     GOSUB reset_routine
     PRINT #5,"All Clear…"
     ENDIF
     WEND 'End program loop
Reset_routine:
    DATUM(0)'Clear FE fault in MC302-K
    DRIVE_CLEAR'Reset drive faults
    WA(100)'Wait in ms
    WDOG=OFF'Cycle enable (WDOG) to the drive…
    WA(50)
    SERVO=ON'Close position loop in MC302-K
    WDOG=ON'Enable drive
    WA(50)
    RETURN
```

DRIVE_CONTROL

Type: Axis Parameter

Description: Sets the value of a control word that is sent to a drive via a digital communications bus, e.g. SERCOS, CAN etc, or to the the local drive when used with a Trio “Drive-In” module.

DRIVE_ENABLE

Type: Axis Parameter

Description: Controls the cyclic communication to a remote drive. When set to 1 cyclic transmission is started. Cyclic comms include a sync telegram and set point telegram sent via the communications bus in use.

Example:
```
DRIVE_ENABLE AXIS(0)=1
DRIVE_ENABLE AXIS(1)=1
```

DRIVE_EPROM

Type: Drive Function

Syntax: **DRIVE_EPROM**

Description: Forces the local drive to perform a save function and save the drive perameters to the drive's flash eprom. Trio "Drive-In" module only.

DRIVE_HOME

Type: Drive Function

Syntax: **DRIVE_HOME**

Description: When the **DRIVE_HOME** is encountered in a Trio BASIC program, the drive will begin its internal homing sequence.

The mode of homing will be based on the settings of the drive's **DREF, NREF, VREF, IN1MODE**, and **REFMODE** parameters. See the homing example in the "Drive-In" Technical Reference Manual. The Trio BASIC program will pause on the **DRIVE_HOME** line until the drive completes the homing sequence (when the Motion Task Active is cleared).

DRIVE_INPUTS

Type: Axis Parameter

Syntax: **DRIVE_INPUTS**

Description: Read input word from a remote or "Drive-In" drive with digital communications capability.

Example: **PRINT DRIVE_INPUTS AXIS(2)**

DRIVE_INTERFACE

Type: Axis Parameter

Syntax: **DRIVE_INTERFACE (function, parameter value)**

Description: Low-level communications link between a "Drive-In" module and the local drive.

The **DRIVE_INTERFACE** provides direct access to the Dual Port RAM in the drive regardless of communication status between the "Drive-In" and the Drive. Even catastrophic drive errors such as "System Error" can be read back using function mode 5, letting a Trio BASIC program determine the drive's status.

Example: **DRIVE_INTERFACE(5,ERRCODE_Byte) 'get error code byte from S3000 drive**

Note: The above example returns either the Most Significant Word (MSW) when **ERRCODE_byte=0**; and the Least Significant Word (LSW) when ERRCODE_byte=1. This is the 32-bit value ERRCODE that is provided by the drive, with 1 bit per fault raised by the drive. A 0 indicates that the fault is not present and a 1 indicates that it is. Bit 0 indicates the status of F01 and bit 31 indicates the status of F32. For example, if faults F29 and F04 are present then **DRIVE_INTERFACE (5,0)** would return 4096 (or hex 1000) and **DRIVE_INTERFACE (5,1)** would return 8.

DRIVE_MODE

Type: Axis Parameter

Syntax: **DRIVE_MODE**

Description: Read or set the mode of a remote or "Drive-In" drive with digital communications capability.

Example: **DRIVE_MODE AXIS(5)=mode1**

DRIVE_MONITOR

Type: Axis Parameter

Syntax: **DRIVE_MONITOR**

Description: Read a monitor word from a remote or “Drive-In” drive with digital communications capability.

Example: **PRINT DRIVE_MONITOR AXIS(0)**

DRIVE_READ

Type: Drive Function

Syntax: **DRIVE_READ (register[, time])**

Description: Reads a drive parameter from the local drive. Trio “Drive-In” modules only.

Parameters:

Register:	Drive parameter 1
Time:	Optional time out value in msec (default=100)

Example: **PRINT DRIVE_READ ($0A, 256)**

DRIVE_RESET

Type: Axis Parameter

Syntax: **DRIVE_RESET [phase]**

Description: Reset the communications link between A “Drive-In” module and the local drive. The **DRIVE_RESET** is typically not required for normal operation. The optional communication phase parameter between the “Drive-In” module and the drive.

Example: **DRIVE_RESET**

Command used to re-establish communications after a “Network Timeout Error” or “Network Protocol Error” error. These errors are due to a command timeout between the MC302-K and the drive. The **DRIVE_RESET** command will reset the communications link between the MC302-K and the local drive.

DRIVE_STATUS

Type: Axis Parameter

Syntax: **DRIVE_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
>>
```

DRIVE_WRITE

Type: Drive Function

Syntax: **DRIVE_WRITE (register, value[, time])**

Description: Writes a value to a drive parameter in the local drive. Trio "Drive-In" modules only.

Parameters:

Register:	Drive parameter 1
Value:	Value to be written
Time	Optional time out value in msec (default=100)

ENCODER

Type: Axis Parameter (Read Only)

Description: The **ENCODER** axis parameter holds a raw copy of the encoder hardware register or the raw data received from a fieldbus controlled drive. On Servo daughter boards, for example, this can be a 12 bit (Modulo 4096) or 14 bit (Modulo 16384) number. On absolute axes the **ENCODER** register holds a value using the number of bits programmed with **ENCODER_BITS**.

The **MPOS** axis measured position is calculated from the **ENCODER** value automatically allowing for overflows and offsets. On MC302X and the built-in axes of a Euro205x or MC206X the **ENCODER** register is 14 bit.

ENCODER_BITS

Type: Axis Parameter

Description: This parameter is only used with an absolute encoder axis. It is used to set the number of data bits to be clocked out of the encoder by the axis hardware. There are 2 types of absolute encoder supported by this parameter; SSI and EnDat. For SSI, 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 to suit the encoder. With the EnDat encoder, bits 0..7 of the parameter are the <u>total</u> number of encoder bits and bits 8..14 are the number of multi-turn bits to be used.

If the number of **ENCODER_BITS** is to be changed, the parameter must first be set to zero before entering the new value.

Example 1:

```
'set up 2 axes of SSI absolute encoder
ENCODER_BITS AXIS(3) = 12
ENCODER_BITS AXIS(7) = 21
```

Example 2:

```
're-initialise MPOS using absolute value from encoder
SERVO=OFF
ENCODER_BITS = 0
ENCODER_BITS = databits
```

Example 3:

```
'A 25 bit EnDat encoder has 12 multi-turn and 13 bits/turn
'resolution. (total number of bits is 25)
ENCODER_BITS = 25 + (256 * 12)
```

Note: If the number of **ENCODER_BITS** is to be changed, the parameter must first be set to zero before entering the new value.

ENCODER_CONTROL

Type: Axis Parameter

Description: Endat encoders can be set to either cyclically return their position, or they can be set to a parameter read/write mode. The mode is controlled with the parameter **ENCODER_CONTROL**.

ENCODER_CONTROL = 1 ' sets parameter read/write mode

ENCODER_CONTROL = 0 ' sets cyclic position return mode

ENCODER_CONTROL is set to 0 on power up or reset. Using the **ENCODER_READ** or **ENCODER_WRITE** functions will set the parameter to 1 automatically.

On the PCI 208 the **ENCODER_CONTROL** should be set for the axis pairs 0/1, 2/3, 4/5 or 6/7 at the same time due to the configuration of the interface transceivers.

Example 1:
```
' Set axes to parameter mode in a pair (PCI 208)
ENCODER_CONTROL AXIS(0)=1
ENCODER_CONTROL AXIS(1)=1
```

ENCODER_ID

Type: Axis Parameter

Description: This parameter returns the ENID parameter from the encoder (fixed at 17 decimal).

(Tamagawa abolute encoder only)

ENCODER_READ

Type: Axis Command

Syntax: **ENCODER_READ (register address)**

Description: Read an internal register from an Absolute Encoder. EnDat absolute encoder only.

Example: **PRINT ENCODER_READ (endat_address)**

ENCODER_STATUS

Type: Axis Parameter

Syntax: **ENCODER_STATUS**

Description: This axis parameter returns both the status field SF and the ALMC encoder error field. The ALMC field is in bits 8..15. The SF field is in bits 0..7.

(Tamagawa abolute encoder only)

ENCODER_TURNS

Type: Axis Parameter

Description: **1. Tamagawa absolute encoder:** This axis parameter returns the number of multi-turn counts from fields ABM0/ABM1/AMB2 of the encoder. The multi-turn data is not automatically applied to the axis **MPOS** after initialisation. The application programmer must apply this from BASIC using **OFFPOS** or **DEFPOS** as required.

2. EnDat absolute encoder: This axis parameter returns the number of multi-turn counts from the encoder.

ENCODER_WRITE

Type: Axis Command

Syntax: **ENCODER_WRITE (register addres, value)**

Description: Write an internal register to an Absolute Encoder. EnDat absolute encoder only.

Example: **ENCODER_WRITE (endat_address, setvalue)**

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.

ENDMOVE_BUFFER

Type: Axis Parameter (Read only)

Only available in system software versions where “LookAhead” is enabled.

Description: This holds the absolute position at the end of the buffered sequence. It is adjusted by **OFFPOS/DEFPOS**. The individual moves in the buffer are incremental and do not need to be adjusted by **OFFPOS** (Look-ahead versions only).

Example: **>>? ENDMOVE_BUFFER AXIS(0)**

This will return the absolute position at the end of the current buffered sequence on axis 0.

ENDMOVE_SPEED

Type: Axis Parameter

Only available in system software versions where "LookAhead" is enabled.

Description: This is used in conjunction with **MOVESP, MOVEASBSSP, MOVECIRCSP** and **MHELICALSP**. It is loaded into the buffer at the same time as the move. The controller will (using the specified value of **ACCEL** or **DECEL**) change the speed of the vector moves so by the end of the **MOVE** starts in **MTYPE** the axis **VPSPEED = FORCE SPEED** (Look-ahead versions only).

Example: **SPEED=15**
(other moves are loaded into the buffer)

```
ENDMOVE_SPEED=10
MOVESP(20)
```

In this example the controller will start ramping down the speed (at the specified rate of **DECEL**) so at the end of the **MOVESP(20)** the **VPSPEED=10**. After which, if another SP move type isn't issued the speed will ramp back to a speed of 15. **ENDMOVE_SPEED** takes priority over **FORCE_SPEED**).

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 MC302X the default setting is 256. This will trip the enable relay only if a following error condition occurs.

For the MC206X and Euro205x, the default value is 268 which is set to also trap critical errors with digital drive communications.

After a critical error has tripped the enable relay, the *Motion Coordinator* must either be reset, or a **DATUM(0)** command must be executed to reset the error flags. **DATUM(0)** is a global command (affects all axes) and needs to run once only.

See Also: **AXISSTATUS, DATUM(0)**

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 may be used to set or read back the fast deceleration rate of each axis fitted. Fast deceleration is used when a **CANCEL** is issued, for example; from the user, a program, or from a software or hardware limit. If the motion finishes normally or **FASTDEC = 0** then the **DECEL** value is used.

Example:
```
DECEL=100       'set normal deceleration rate
FASTDEC=1000    'set fast deceleration rate
MOVEABS(10000) 'start a move
WAIT UNTIL MPOS= 5000 'wait until the move is half finished
CANCEL 'stop move at fast deceleration rate
```

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_LATCH

Type: Axis Parameter (Read Only)

Description: Contains the intial FE value which caused the axis to put the controller into "MOTION_ERROR". This value is only set when the FE exceeds the **FE_LIMIT** and the **SERVO** parameter has been set to 0. **FE_LATCH** is reset to 0 when the axis' **SERVO** parameter is set back to 1.

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.

FE_LIMIT_MODE

Type: Axis Parameter

Syntax: **FE_LIMIT_MODE = value**

Description: When this parameter is set to 0, the axis will cause a **MOTION_ERROR** immediately if the FE exceeds the **FE_LIMIT** value.

If **FE_LIMIT_MODE** is set to 1, the axis will only generate a **MOTION_ERROR** when the FE exceeds **FE_LIMIT** during 2 consecutive servo periods. This means that if **FE_LIMIT** is exceeded for one servo period only, it will be ignored.

The default value for **FE_LIMIT_MODE** is 0.

FE_RANGE

Type: Axis Parameter

Syntax: **FE_RANGE = 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.

FORCE_SPEED

Type: Axis Parameter

Only available in system software versions where "LookAhead" is enabled.

Description: This is used in conjunction with **MOVESP, MOVEASBSSP, MOVECIRCSP** and **MHELICALSP**. It is loaded into the buffer at the same time as the move. The controller will (using the specified value of **ACCEL** or **DECEL**) change the speed of the vector moves so at the point the move starts in **MTYPE** the axis **VPSPEED = FORCE SPEED** (LookAhead versions only).

Example:
```
SPEED = 15
```

(other moves are loaded into the buffer)

```
FORCE_SPEED = 10
MOVESP(20)
```

In this example the controller will ramp the speed down to a speed of 10 for the duration of the **MOVESP(20)**, after which it will ramp back to a speed of 15. (If **ENDMOVE_SPEED** is set then this takes priority over force speed).

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**.
FS_LIMIT is disabled when it has a value greater than **REP_DIST**.

FULL_SP_RADIUS

Type: Controller Parameter

Only available in system software versions where “LookAhead” is enabled.

Description: This sets the full speed radius in user UNITS. Once set the controller will use the full programmed **SPEED** value for radii above the value of **FULL_SP_RADIUS**. Where the radius is below the value of **FULL_SP_RADIUS** the controller will proportionally reduce the speed.

Example: In the following program, when the first **MOVECIRC** is reached the speed remains at 10 because the radius (8) is greater than that set in **FULL_SP_RADIUS**. For the second **MOVECIRC** the speed is reduced by 50% to a value of 5, because the radius is 50% of that stored in **FULL_SP_RADIUS**.

```
MERGE=ON
SPEED=10
FULL_SP_RADIUS=6
DEFPOS(0,0)

MOVE(10,10)
MOVE(10,5)
MOVE(5,5)
MOVECIRC(8,8,0,8,1)
MOVECIRC(3,3,0,3,1)
MOVE(5,5)
MOVE(10,5)
```

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.

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 K_i and a sum of position errors $\int e$, the contribution to the output signal is:

$$O_i = K_i \times \int e$$

Note: Servo gains have no effect on stepper motor axes.

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.

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**.

LIMIT_BUFFERED

Type: Controller Parameter

Only available in system software versions where “LookAhead” is enabled.

Description: This sets the maximum number of move buffers available in the controller. The maximum value (and also the default) is 16 (look-Ahead versions only).

Example:
```
LIMIT_BUFFERED=10
```

This will set the total number of available buffered moves in the controller to 10.

LINKAX

Type: Axis Parameter (Read Only)18

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: MARKB returns **TRUE** when the second registration position has been latched. This is set to **FALSE** by the **REGIST** command and set to **TRUE** when the registration event occurs. When **MARKB** is **TRUE** the **REG_POSB** is valid.

See also **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, P230, P240 and P280. 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 MC206X 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
```

MOVES_BUFFERED

Type: Axis Parameter (Read only)

Only available in system software versions where “LookAhead” is enabled.

Description: This returns the number of moves being buffered by the axis when using the look-ahead functionality (look-ahead versions only).

Example:
```
>>? VECTOR_BUFFERED AXIS(0)
```

This will return the total number of current buffered moves.

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
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.

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 value to be offset by any amount without affecting the motion which is in progress. **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 after the axis position is changed.

Example 1: Change the current position by 125, using the command line terminal:

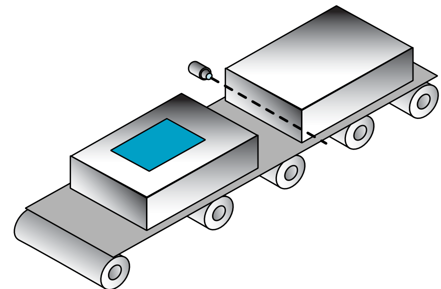
```
>>?DPOS
300.0000
>>OFFPOS=125
>>?DPOS
425.0000
```

Example 2: Define the current demand position as zero:

```
OFFPOS=-DPOS
WAIT UNTIL OFFPOS=0'   wait until applied
```

This is equivalent to **DEFPOS(0)**

Example 3: 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 first position of the window which will be used for registration marks if windowing is specified by the **REGIST()** command.

Example:
```
'only look for registration marks between 170 1nd 230mm
OPEN_WIN=170.00
CLOSE_WIN=230.0
REGIST(256+3)
WAIT UNTIL MARK
```

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. See DAC types for each controller.

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:

$$O_{ov} = K_{0v} \times \delta P_m$$

Note: Negative values are normally required. Servo gains have no effect on stepper motor axes.

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 K_p and position error E, its contribution to the output signal is:

$$O_p = K_p \times E$$

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
```

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 1: 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: If used in a Servo axis, increasing **PP_STEP** will require a proportionate decrease of all loop gain parameters.

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)

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
```

REP_DIST

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 **REP_DIST** is applied. In the default setting (**REP_OPTION bit 0=0**) **REP_DIST** 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: when set **ON, the** automatic repeat option of the **CAMBOX** or **MOVELINK** function will be turned OFF. When the system software has set the option OFF it automatically clears bit 1 of **REP_OPTION**.

REP_OPTION bit 2: when this is set **ON**, the functions **REP_DIST, DEFPOS and OFFPOS** will affect **MPOS** only. Bit 2 is an option for Stepper + Encoder axes, it is not appropriate for servo axes.

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 **JOG-SPEED** 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.**

RS_LIMIT is disabled when its value is outside the range of **REP_DIST**.

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.**

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
```

SPHERE_CENTRE

Type: Axis Command

Syntax: **SPHERE_CENTRE(tablex, tabley, tablez)**

Description: Returns the co-ordinates of the centre point (x, y, z) of the most recent **MOVE_SPHERICAL**. x, y and z are returned in the **TABLE** memory area and can be printed to the terminal as required.

Example:
```
SPHERE_CENTRE(10, 11, 30)
PRINT TABLE(10);", ";TABLE(11);", ";TABLE(12)
```

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 **MOVE-MODIFY** move types.

Note: The **SRAMP** factor should not be changed while a move is in progress.

TANG_DIRECTION

Type: Axis Parameter

Only available in system software versions where "LookAhead" is enabled.

Description: When used with a 2 axis X-Y system, this parameter returns the angle in radians that represents the vector direction of the interpolated axes. The value returned is between -PI and +PI and is determined by the directions of the interpolated axes as follows:

X	Y	value
0	1	0
1	0	PI/2
0	-1	PI/2 (+PI or -PI)
-1	0	-PI/2

Example1: Note scale_factor_x MUST be the same as scale_factor_y

```
UNITS AXIS(4)=scale_factor_x
UNITS AXIS(5)=scale_factor_y

BASE(4,5)
MOVE(100,50)
angle = TANG_DIRECTION
```

Example2:

```
BASE(0,1)
angle_deg = 180 * TANG_DIRECTION / PI
```

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**

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 refer to the **MICROSTEP** command when choosing **UNITS**.

VECTOR_BUFFERED

Type: Axis Parameter (Read only)

Only available in system software versions where “LookAhead” is enabled.

Description: This holds the total vector length of the buffered moves. It is effectively the amount the VPU can assume is available for deceleration. It should be executed with respect to the first axis in the group (look-ahead versions only).

Example:

```
>>BASE(0,1,2)
>>? VECTOR_BUFFERED AXIS(0)
```

This will return the total vector length for the current buffered moves whose axis group begins with axis(0).

VERIFY

Type: Axis Parameter

Description: The verify axis parameter is used to select different modes of operation on a stepper encoder, encoder or servo axis. Its use depends upon the hardware.

(A) P240, P280, MC302X, PCI208

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

(B) Euro205x

VERIFY=OFF

The encoder counting circuit is configured to accept **STEP** and **DIRECTION** signals hard wired to the encoder A and B inputs.

VERIFY=ON

The encoder circuit is configured for the usual quadrature input.

Take care that the encoder inputs do not exceed 5 volts.

(B) P270 SSI Daughter Board

VERIFY=ON

SSI Binary encoder operation.

VERIFY=OFF

SSI Gray code encoder operation.

Gray code / Binary option available on P270 with V1.2 FPGA onwards.

Example:
```
VERIFY AXIS(3)=ON
```

Note: *Motion Coordinator* that use Feature Enable Codes to activate axis functions will power up with VERIFY either OFF or ON depending on axis type. To ensure that VERIFY is in the correct state, set only the required FECs for the axis type required. Forcing the axis type with the ATYPE command alone will leave the axis with the wrong encoder operation.

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 K_{vff} and change in position δP_d , the contribution to the output signal is:

$$O_v ff = K_{vff} \times \delta P_d$$

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
```