Trio Motion Technology
PCI 208 *Motion Coordinator*
Technical Reference Manual

Edition 1.1 • September 2004

All goods supplied by Trio are subject to Trio's standard terms and conditions of sale.

This manual applies to systems based on the *Motion Coordinator* PCI 208 with system software version 1.63, or higher.

The material in this manual is subject to change without notice. Despite every effort, in a manual of this scope errors and omissions may occur. Therefore Trio cannot be held responsible for any malfunctions or loss of data as a result.

Revision 1.1 September 2004

Copyright (C) 2000..2004 Trio Motion Technology Ltd.

All Rights Reserved

UK

Trio Motion Technology Ltd.
Shannon Way
Tewkesbury
GL20 8ND
United Kingdom

Phone: +44 (0)1684 292333
Fax: +44 (0)1684 297929

USA

Trio Motion Technology LLC.
1000 Gamma Drive, Suite 206
Pittsburgh
PA 15238,
USA

Phone: +1 412 968 9744
Fax: +1 412 968 9746

⚠ SAFETY WARNING ⚠

During the installation or use of a control system, users of Trio products must ensure there is no possibility of injury to any person, or damage to machinery.

Control systems, especially during installation, can malfunction or behave unexpectedly. Bearing this in mind, users must ensure that even in the event of a malfunction or unexpected behaviour the safety of an operator or programmer is never compromised.

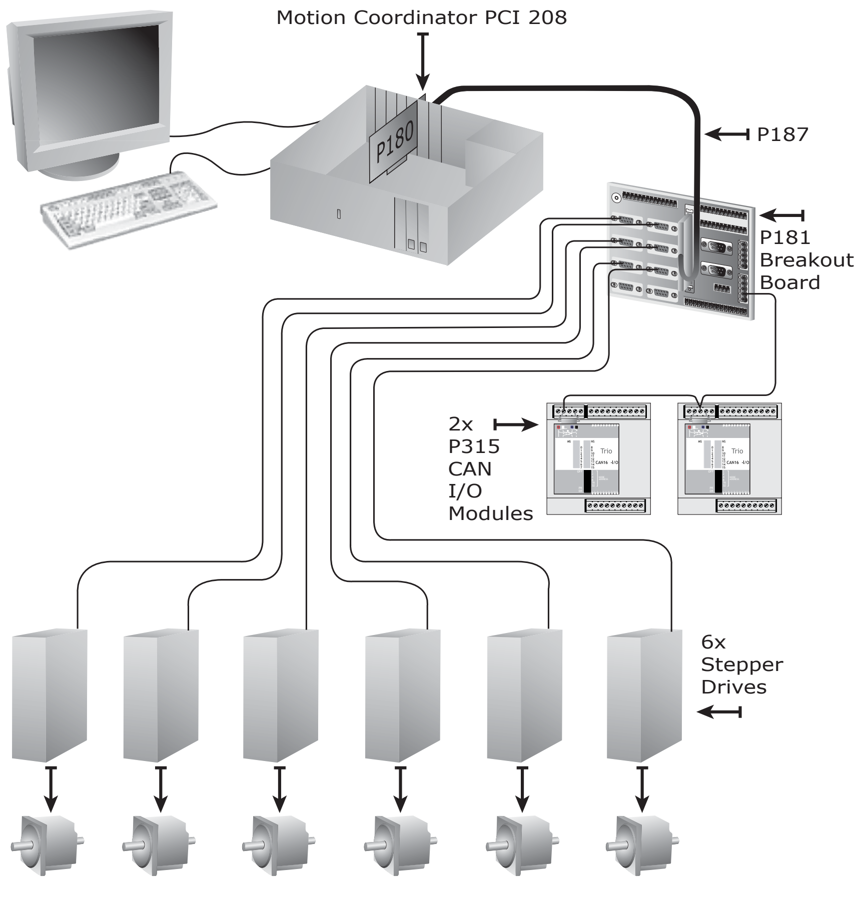
CONTENTS

CHAPTER

1

Introduction

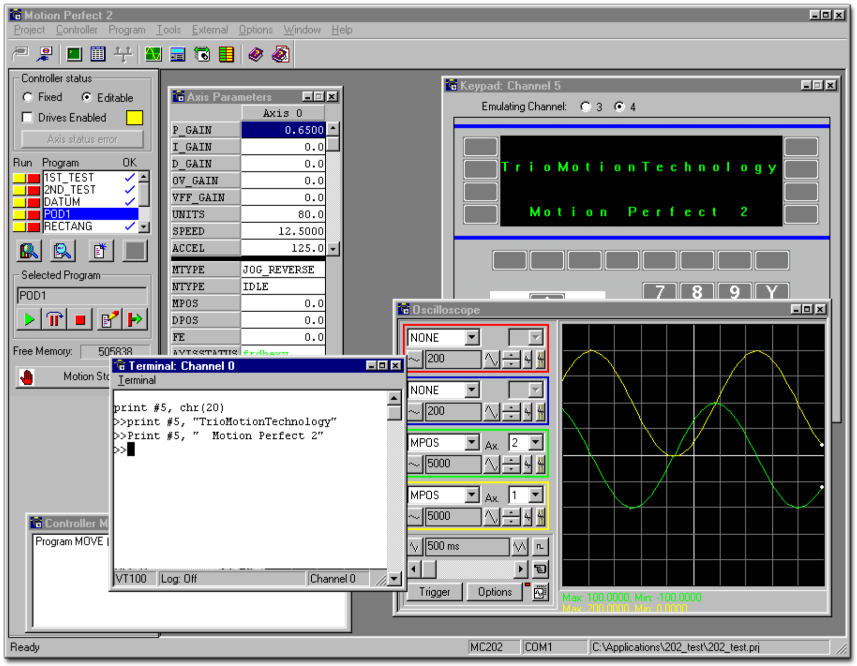
Trio Motion Technology's range of *Motion* Coordinator products are designed to enable the control of industrial machines with a minimum of external components. The products may be combined to build a control system capable of driving a multi-axis machine and its auxiliary equipment. The *Motion* Coordinator PCI 208 card allows you to control up to 8 servo or stepper motors, digital and analog I/O and additional equipment via 2 CAN ports. Multiple PCI 208 cards can be put in a PC to increase capacity. The controller is programmed via an Active X component from PC based programming languages and in addition can execute programs internally using the *Trio* BASIC programming language.



Typical System Configuration

Setup and Programming

The PCI 208 is inserted into a free PCI slot within the PC running Windows 2000 or Windows XP. Trio's *Motion* Perfect application development tool can be used to view the status of the axes, IO and other facilities of the PCI 208. *Motion* Perfect can be run alongside the users application running via the Active X link into the PCI 208.



Axis Parameters

	Axis 0
P_GAIN	0.6500
I_GAIN	0.0
D_GAIN	0.0
OV_GAIN	0.0
VFF_GAIN	0.0
UNITS	80.0
SPEED	12.5000
ACCEL	125.0
MTYPE	JOG_REVERSE
NTYPE	IDLE
MPOS	0.0
DPOS	0.0
FE	0.0
AXISSTATUS	frdhexv

Terminal: Channel 0

```
print #5, chr(20)
>>print #5, "TrioMotionTechnology"
>>Print #5, "  Motion Perfect 2"
>>
```

VT100 Log: Off Channel 0

```
TrioMotionTechnology
  Motion Perfect 2
```

Motion Perfect

Users programs are usually constructed on the PC in languages such as, Visual BASIC, Delphi or Visual C. The programs use the Trio PC Active X component to access the PCI 208 card functions via the dual port ram. *Motion* Perfect can be used if required to help debug the users application program. In addition *Motion* Perfect provides an easy, rapid way to develop control programs. All the standard program constructs are provided; variables, loops, input/output, maths and conditions. Extensions to this basic instruction set exist to permit a wide variety of motion control facilities, such as single axis moves, synchronized multi axis moves and unsynchronised multi axis moves as well as the control of the digital I/O.

Products

This manual covers the *Motion* Coordinator PCI 208 and its option modules:

Motion Coordinator PCI 208 and Options:

Product Code	Name	Description
P180	**PCI 208**	PCI card Servo/stepper *Motion* Coordinator for up to 8 axes. 20 opto-isolated inputs and 10 opto-isolated outputs are built in. Multi-tasking *Trio* BASIC. 2 CAN channels for I/O expansion.
P181	**Breakout Board**	DIN rail mounting PCB to allow easy connection to all the PCI 208 functions in prototype and low volume applications.
P182	**Additional Stepper Axis**	The standard PCI 208 has 2 stepper/servo axes. Additional stepper axes (up to a total of 8) can be activated by purchasing P182 options. The P182 is supplied as a "Feature Enable Code". This is a code word to enable this feature of the P180.
P183	**Additional Servo Axis**	The standard PCI 208 has 2 stepper/servo axes. Additional servo axes (up to a total of 8) can be activated by purchasing P183 options. *Analog voltage output servo axes will in addition require a P184 or P185 module.*
P184	**4 Axis DAC Module**	Provides 4 x +/-10 volt outputs for analog servo drive control. The outputs are opto-isolated and have 16 bit resolution. The P184 in addition provides 2 x 0-10 volt analog inputs.
P185	**8 Axis DAC Module**	Provides 8 x +/-10 volt outputs for analog servo drive control. The outputs are opto-isolated and have 16 bit resolution.
P187	**2.5m 100 Way Cable**	Cable to connect PCI 208 to breakout board.
P315	**CAN 16 I/O Module**	DIN Rail mounted 24v I/O expander module provides 16 opto-isolated channels each of which may be used as an Input or an Output. Up to 16 P315 modules can be connected to the PCI 208 to expand the I/O.
P325	**CAN Analog Input Module**	8 x +/-10 volt input channels. Up to 4 P325 modules can be connected to the PCI 208 to expand the analog input capacity.

I/O Expansion options

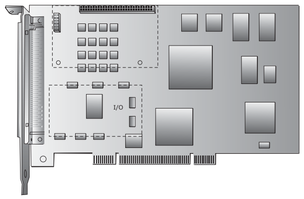
P315 - CAN-16 I/O

P325 - CAN-8 Analog Inputs

System Examples

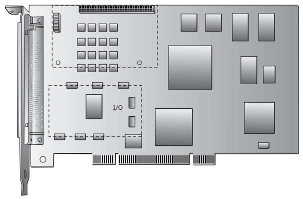
Example 1 Simplest Possible System - 2 Axis Stepper System

- 1 x P180

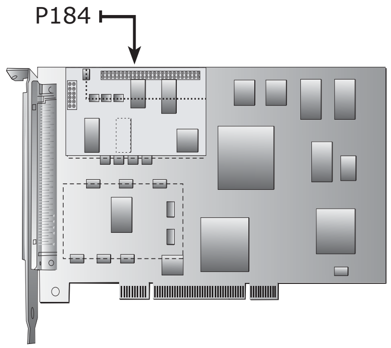


Example 2 8 Axis Stepper System

- 1 x P180 PCI 208 *Motion* Coordinator
- 6 x P182 Additional stepper axis

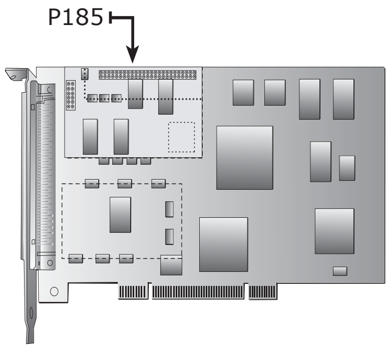


Example 3 4 Axis Servo System



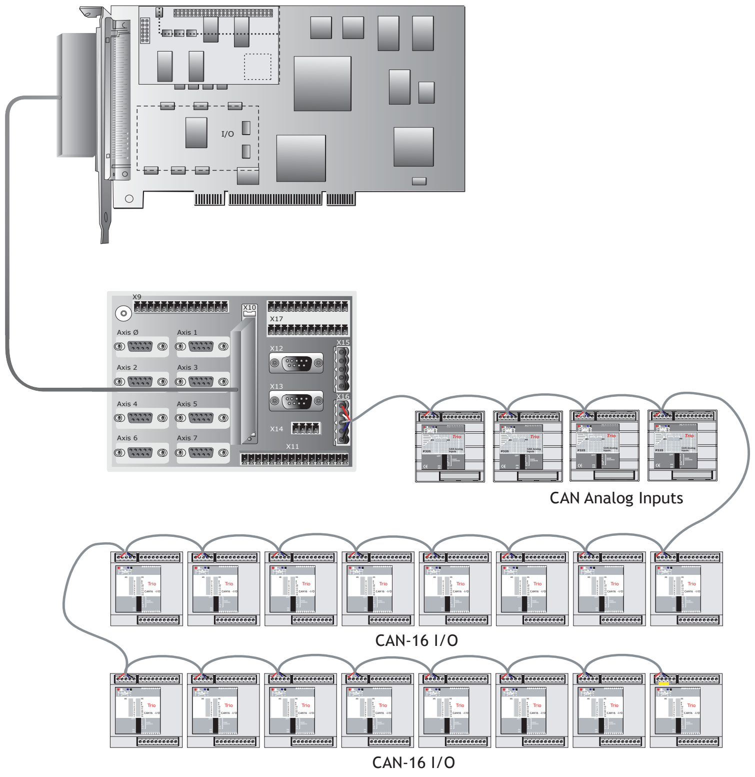
• 1 x P180	PCI 208 *Motion* Coordinator
• 2 x P183	Additional servo axis
• 1 x P184	4 axis DAC module

Example 4 6 Axis Servo + 2 Axis Stepper System



• 1 x P180	PCI 208 *Motion* Coordinator
• 4 x P183	Additional servo axis
• 2 x P182	Additional stepper axis
• 1 x P185	8 Axis DAC module

Example 5 8 Axis Servo System with 256 expansion I/O and 32 analog inputs



•	1 x P180	PCI 208 *Motion* Coordinator
•	1 x P181	Breakout board
•	1 x P187	2.5 metre breakout board cable
•	1 x P185	8 Axis DAC module
•	6 x P183	Additional servo axis
•	16 x P315	16 I/O module
•	4 x P325	8 Analog input module

Features and Typical Applications

The PCI 208 software contains accurate motion control functions for the generation of complex movements of various types, including:

- Linear interpolation of up to 8 axes
- Circular and helical interpolation
- Variable speed and acceleration profiles
- Electronic gearboxes
- Electronic cam profiles
- Linked motion
- Axis superimposition
- Imaginary axes
- Hardware registration

The operator interface is normally achieved using the PC user interface.

The system is able to control a wide range of mechanisms and equipment including:

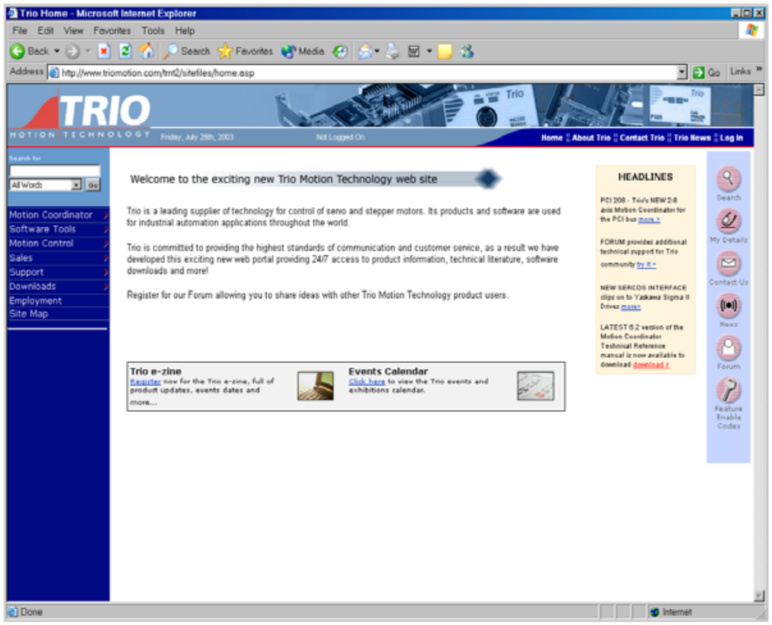
- Brushless servo motors
- Stepper motors
- Brushed DC servo motors
- Hydraulic servo valves
- Hydraulic proportional valves
- Pneumatic/hydraulic solenoids
- Relays/contactors
- Switches / Thumbwheels
- Status lamps

Typical applications:

• Cut to length	• Coil winding	• Automotive welding
• Flying shears	• Laser guidance	• Spark erosion
• Glue laying	• Electronic assembly	• Drilling
• Web control	• Printing	• Milling
• Tension control	• Collating	
• Pick & Place	• Packaging	• YOUR application

The Trio Motion Technology Website

The Trio website contains up to the minute news, information and support for the whole *Motion* Coordinator product range.



Welcome to the exciting new Trio Motion Technology web site

Trio is a leading supplier of technology for control of servo and stepper motors. Its products and software are used for industrial automation applications throughout the world.

Trio is committed to providing the highest standards of communication and customer service, as a result we have developed this exciting new web portal providing 24/7 access to product information, technical literature, software downloads and more!

Register for our Forum allowing you to share ideas with other Trio Motion Technology product users.

Trio e-zine
Register now for the Trio e-zine, full of product updates, events dates and more....

Events Calendar
Click here to view the Trio events and exhibitions calendar.

HEADLINES

PCI 208 - Trio's NEW 2-8 axis Motion Coordinator for the PCI bus more >

FORUM provides additional technical support for Trio community try it >

NEW SERCOS INTERFACE clips on to Yaskawa Sigma II Drives more>

LATEST 6.2 version of the Motion Coordinator Technical Reference manual is now available to download download >

Website Features

- Latest News
- Product Information
- Manuals
- Support Software
- System Software Updates
- Technical Support
- User's Forum
- Application Examples
- Employment Opportunities

WWW.TRIOMOTION.COM

CHAPTER

2

Hardware Overview

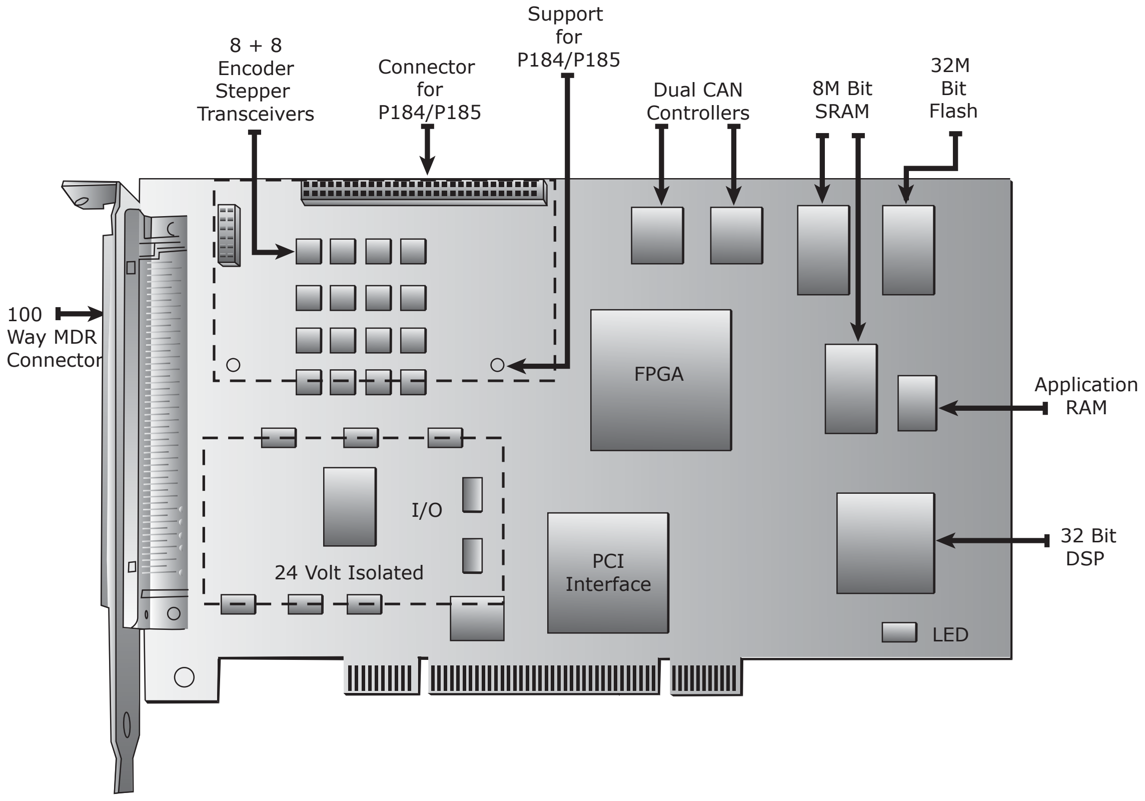
Motion Coordinator PCI 208 - Product Code P180

Overview The PCI 208 is a powerful *Motion Coordinator* designed to fit into the PCI slot of a PC. It can control up to 8 servo or stepper motors, has built-in opto-isolated I/O and has one or two CAN channels. An expansion connector allows for the addition of +/-10 volt outputs for the control of voltage output servos. The PCI 208 is designed to provide a powerful yet cost-effective control solution for OEM machine builders who are building machines centred around a PC.

Programming The PCI 208 can be programmed via an OCX software component from most PC programming languages. In addition it can be optionally programmed to execute programs on board using Multi-tasking Trio BASIC.

Breakout Options An optional "Breakout" board for the PCI 208 can be used to ease the task of making connections to the 100 way PCI 208 backplate connector. For some simple applications connections can be wired to a mating connector which plugs directly into the PCI 208. For serial production a customised connection PCB can provide the best solution. For many other applications the breakout board is the most convienient solution.

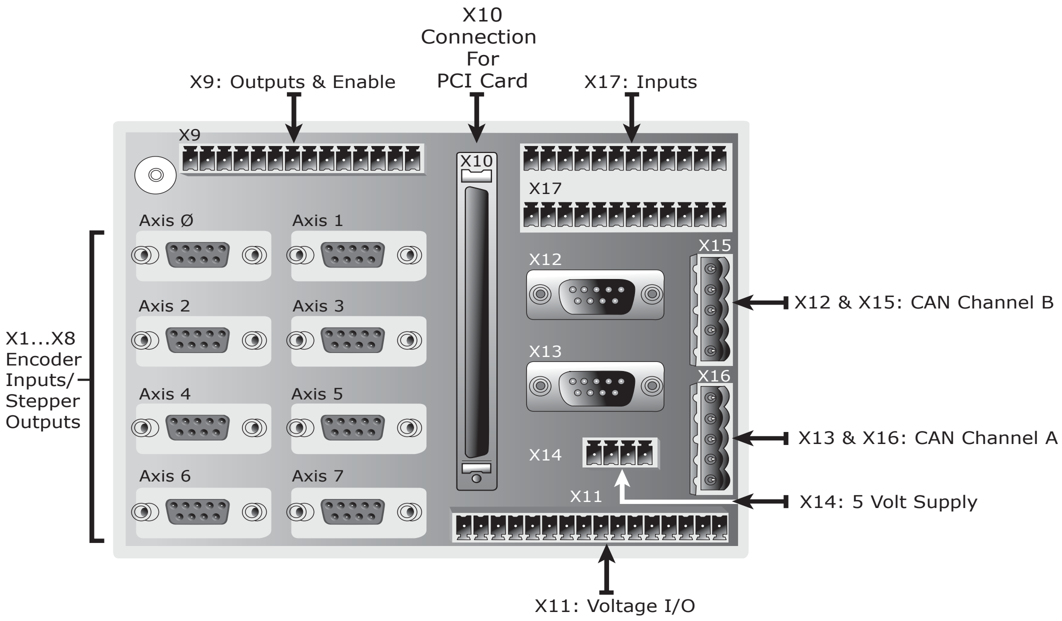
PCI 208 Block Diagram



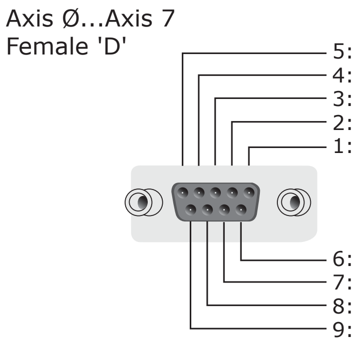
Connectors

Connectors

Breakout Board



Encoder/Stepper Connectors:

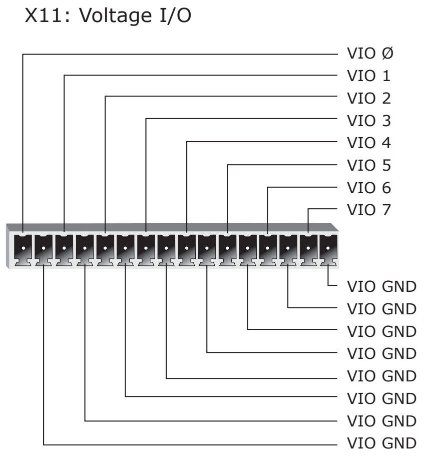


Pin	Servo Axis	Stepper Axis
1	Enc. A	Step +
2	Enc. /A	Step -
3	Enc. B	Direction +
4	Enc. /B	Direction -
5	GND	GND
6	Enc. Z	Boost +
7	Enc. /Z	Boost -
8	5v	5v
9	Not Connected	Not Connected
shell	Screen	Screen

Each of the axes 0 to 7 has its encoder or stepper connections brought out to a separate female D connector. Each axis can be switched to have either an encoder input or differential stepper outputs provided the necessary Feature Enable Codes have been installed.

Voltage I/O:

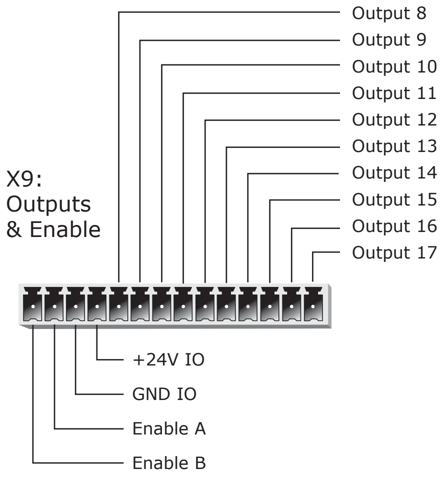
X11 is used to provide isolated voltage I/O when the P184 or P185 DAC modules have been fitted to the PCI208. When the P184 module has been fitted VIO0 to VIO3 provide 4 x +/-10 volt voltage outputs and VIO4 to VIO7 provide 4 x 0..10 volt analog inputs. When the P185 module has been fitted VIO0 to VIO7 provide 8 x +/-10 volt voltage outputs. The analog outputs have 16 bit resolution and are normally used for controlling analog servo drives. Each analog input/output pin has a GND connection alongside to ease connections. If neither a P184 nor P185 are fitted X11 has no function.



24volt Outputs and Enable

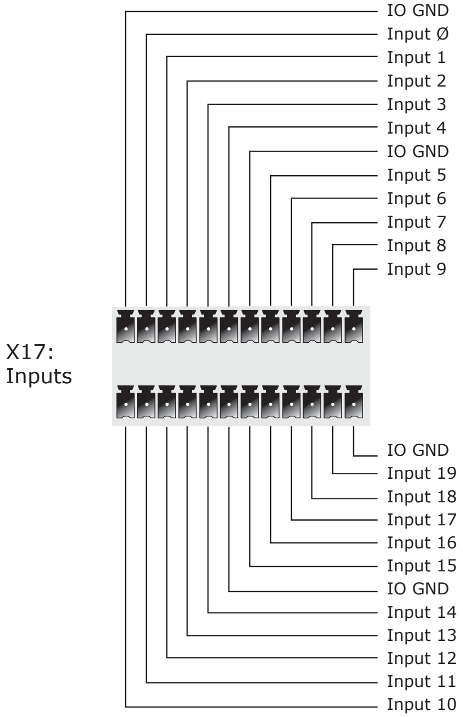
X9 provides 10 x 24 V sourcing opto-isolated output channels. The outputs are current limited and are rated at 250mA. In addition there is a 1 amp rating for all channels combined. The IO GND is common with the isolated input channels.

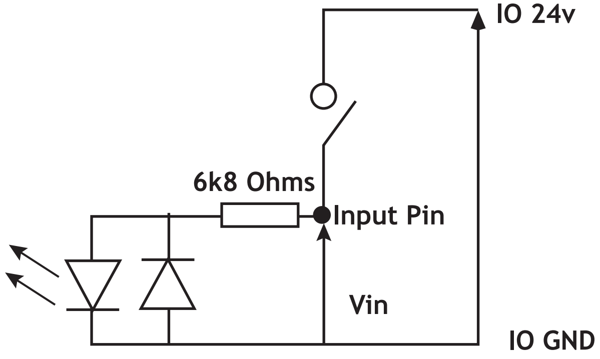
The +24 V IO connection is used to provide power to the isolated 24 volt outputs ONLY



24 volt Inputs

X17 provides 20 x 24 V opto-isolated input channels. The IO GND is common with the isolated output channels.

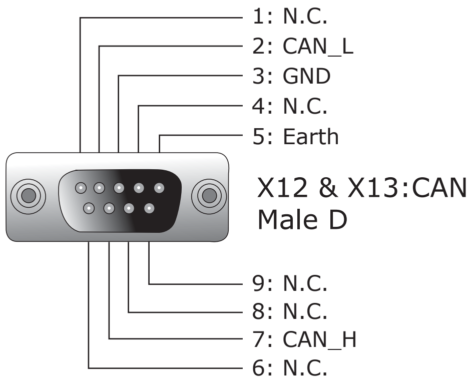


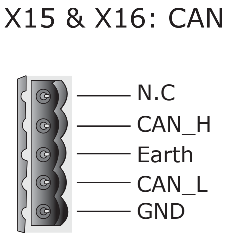


Simplified Input Schematic

CAN connections

Each of the two CAN channels A and B is provided with 2 connectors. This is to suit either DeviceNet or CANopen style connections.



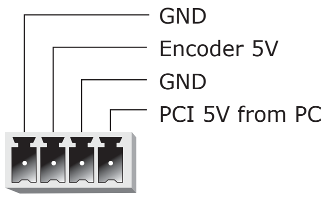


CAN Channels	CANopen Style	DeviceNet Style	Description
CAN Channel A	X13	X16	Used for connection of Trio IO modules if used. Addressed from CAN command as channel -1 E.G CAN(-1,2,1)
CAN Channel B	X12	X15	Auxiliary CAN channel. Addressed from CAN command as channel 0 E.G CAN(0,2,1)

5 Volt Supply

The PCI 208 can provide up to 200mA for powering external encoders, from the PC power supply. This is limited by an electronic circuit. The X14 connector allows an external 5 volt power supply to be easily connected if this is insufficient to power the any connected encoders.

The PCI 5 V and Encoder 5 V pins MUST be wired together to power the encoders from the PCI card.

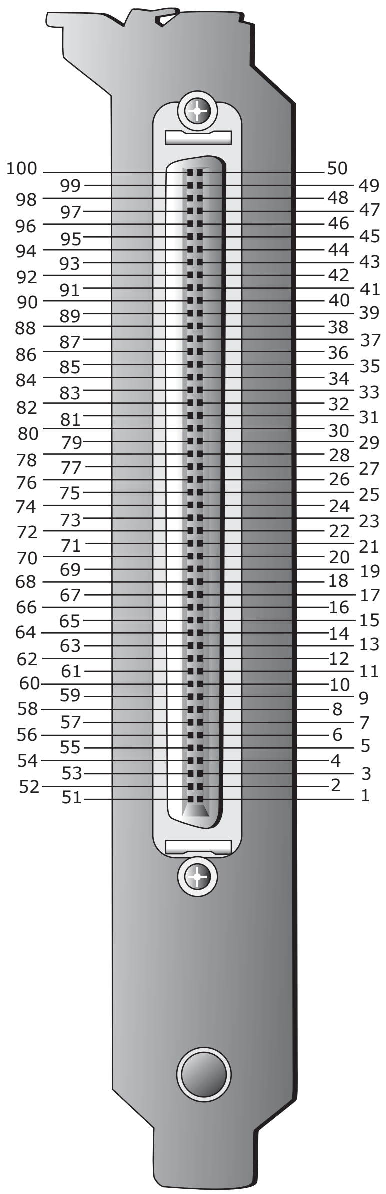


X14: 5 Volt Jumper

PCI 208 - Feature Summary

Size	175mm x 106mm x 18mm PCB overall (excludes PC bracket)
Weight	150g
Operating Temperature	0 – 45 degrees C
Control Inputs	Forward Limit, Reverse Limit, Datum Input, Feedhold Input, Software assignable for each axis. Registration Input
PCI Port	PCI v 2.2 32bit 33Mhz 32bit x 4096 memory locations
Position Resolution	32 bit position count
Interpolation modes	Linear 1-8 axes, circular, helical, CAM Profiles, speed control, electronic gearboxes.
Programming	Via PC OCX component from PC programs. Additionally Multi-tasking Trio BASIC system may be used, maximum 7 simultaneous Trio BASIC programs.
Speed Resolution	32 bit. Speed may be changed at any time. Moves may be merged.
Servo Cycle	1000, 500, or 250 usec for all 8 axes
Memory	32 Mbits Flash, 9 Mbits system RAM, 2 Mbits application RAM, 128kBits PCI dual port RAM
Power Input	Powered via PC. Isolated 24 volt outputs require external power
Drive Enable Output	Solid state relay 24 volt normally open contacts
+/-10 volt DAC Output	4 or 8 with 16 bit resolution optionally provided by P184 or P185 options
Encoder Inputs	Up to 8 axes, differential 5 V inputs, 6Mhz maximum edge rate
Stepper Outputs	Up to 8 axes, step and direction. 2Mhz maximum step rate
24 volt Inputs	20 Opto-isolated 24 volt inputs. 8 may be used as hardware registration inputs
24 volt Outputs	10 Opto-isolated 24 volt outputs, 250 mA /channel rating + 1 Amp all channels limit

PCI 208 Backplate Connector :

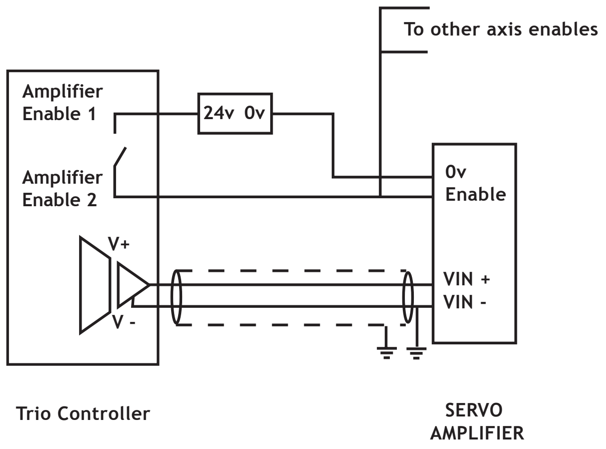


100	VOUT 1	VOUT 0	50
99	VOUT 3	VOUT 2	49
98	VIO 5	VIO 4	48
97	VIO 7	VIO 6	47
96	VIO GND	+5 V ENC	46
95	ENC GND	ENC GND	45
94	ENABLE A	/ENC Z7	44
93	ENABLE B	ENC Z7	43
92	/ENC B7	/ENC A7	42
91	ENC B7	ENC A7	41
90	/ENC Z6	/ENC B6	40
89	ENC Z6	ENC B6	39
88	/ENC A6	/ENC Z5	38
87	ENC A6	ENC Z5	37
86	/ENC B5	/ENC A5	36
85	ENC B5	ENC A5	35
84	/ENC Z4	/ENC B4	34
83	ENC Z4	ENC B4	33
82	/ENC A4	/ENC Z3	32
81	ENC A4	ENC Z3	31
80	/ENC B3	/ENC A3	30
79	ENC B3	ENC A3	29
78	/ENC Z2	/ENC B2	28
77	ENC Z2	ENC B2	27
76	/ENC A2	/ENC Z1	26
75	ENC A2	ENC Z1	25
74	/ENC B1	/ENC A1	24
73	ENC B1	ENC A1	23
72	/ENC Z0	/ENC B0	22
71	ENC Z0	ENC B0	21
70	/ENC A0	CAN BL	20
69	ENC A0	CAN BH	19
68	CAN AL	+24 V IO	18
67	CAN AH	0V IO	17
66	IN 0	OUT 8	16
65	IN 1	OUT 9	15
64	IN 2	OUT 10	14
63	IN 3	OUT 11	13
62	IN 4	OUT 12	12
61	IN 5	OUT 13	11
60	IN 6	OUT 14	10
59	IN 7	OUT 15	9
58	IN 8	OUT 16	8
57	IN 9	OUT 17	7
56	IN 10	IN 11	6
55	IN 12	IN 13	5
54	IN 14	IN 15	4
53	IN 16	IN 17	3
51	IN 18	IN 19	2
51	+24 V IO	0V IO	1

Drive Enable (Watchdog) Relay Output

An internal relay contact is used to enable external drives when the controller has powered up correctly and the system and application software is ready. The drive enable is a single pole relay with a set of normally open contacts. The enable relay contact will be open circuit if there is no power on the controller OR a following error exists on a servo axis OR the user program sets it open with the WDOG=OFF command.

The drive enable relay may, for example, be incorporated within a hold-up circuit or chain that must be intact before a 3-phase power input is made live.



drive Enable Output

Note: ***ALL STEPPER AND SERVO DRIVES MUST BE INHIBITED WHEN THE DRIVE ENABLE OUTPUT IS OPEN CIRCUIT***

CHAPTER

3

Installation

Motion Coordinator PCI 208

Packaging

The *Motion Coordinator* PCI 208 is designed to be fitted into any free PCI slot within most modern PC's. If P184 or P185 option modules are to be fitted they should be installed in a static free environment by a static protected operator. Take care to avoid touching any of the P180's electronic components during installation. Ensure that the P180 is fully seated into the PCI slot and the break-out board is connected prior to switching on the PC.

Connection To Other Trio Products

The PCI 208 may be connected to other *Motion Coordinator* modules on the CAN bus only.

Environmental Considerations

Avoid violent shocks to, or vibration of, the modules whilst in use or storage.

EMC Considerations

Most pieces of electrical equipment will emit noise either by radiated emissions or conducted emissions along the connecting wires. This noise can cause interference with other equipment near by which could lead to that equipment malfunctioning. These sort of problems can usually be avoided by careful wiring and following a few basic rules.

1) Mount noise generators such as contactors, solenoid coils and relays as far away as possible from the modules.
2) Where possible use solid-state contactors and relays.
3) Fit suppressors across coils and contacts.
4) Route heavy current power and motor cables away from signal and data cables.
5) Ensure all the modules have a secure earth connection.
6) Where screened cables are used terminate the screen with a 360 degree termination, if possible, rather than a “pig-tail” and connect both ends of the screen to ground.

The screening should be continuous, even where the cable passes through a cabinet wall or connector.

These are just very general guidelines and for more specific advice on specific controllers, see the installation requirements later in this chapter. The consideration of EMC implications is now more important than ever since the introduction of the EC EMC directive which makes it a legal requirement for the supplier of a product to the end customer to ensure that it does not cause interference with other equipment and that it is not itself susceptible to interference from other equipment.

Background to EMC Directive

Since 1st January 1996 all suppliers of electrical equipment to end users must ensure that their product complies with the 89/336/EEC Electromagnetic Compatibility directive. The essential protection requirements of this directive are:

1) Equipment must be constructed to ensure that any electromagnetic disturbance it generates allows radio and telecommunications equipment and other apparatus to function as intended.
2) Equipment must be constructed with an inherent level of immunity to externally generated electromagnetic disturbances.

Suppliers of equipment that falls within the scope of this directive must show "due diligence" in ensuring compliance. Trio has achieved this by having products that it considers to be within the scope of the directive tested at an independent test house.

As products comply with the general protection requirements of the directive they can be marked with the CE mark to show compliance with this and any other relevant directives. At the time of writing this manual the only applicable directive is the EMC directive. The low voltage directive (LVD) which took effect from 1st January 1997 does not apply to current Trio products as they are all powered from 24V which is below the voltage range that the LVD applies to.

Just because a system is made up of CE marked products does not necessarily mean that the completed system is compliant. The components in the system must be connected together as specified by the manufacturer and even then it is possible for some interaction between different components to cause problems but obviously it is a step in the right direction if all components are CE marked.

Testing Standards

For the purposes of testing a typical system configuration had to be chosen because of the modular nature of the *Motion Coordinator* products. Full details of this and copies of test certificates can be supplied by Trio if required. For each typical system configuration testing was carried out to the following standards:

Emissions - BS EN55022:1995 or BS EN50081-1:1992

(depending on the particular product.)

Note that both standards specify the same limits for radiated emissions which is the only applicable part of the standards to Trio products. Most products conform to the Class A limits but some products, such as the range of membrane keypads, are within Class B limits.

Immunity - BS EN50082-2:1995.

This standard sets limits for immunity in an industrial environment and is a far more rigorous test than part 1 of the standard.

Installation Requirements to Ensure Conformance

Motion Coordinator PCI 208 and options

When the Trio products are tested they are wired in a typical system configuration. The wiring practices used in this test system must be followed to ensure the Trio products are compliant within the completed system.

A summary of the guidelines follows:

1) The PC in which the PCI 208 is installed must meet the required EMC standards and be earthed.
2) If any IO lines are not to be used they should be left unconnected rather than being taken to a terminal block, for example, as lengths of unterminated cable hanging from an IO port can act as an antenna for noise.
3) Screened cables should be used for encoder, stepper and registration input feedback signals and for the demand voltage from the controller to the servo drive if relevant. The demand voltage wiring must be less than 1m long and preferably as short as possible. The screen should be connected to earth at both ends. Termination of the screen should be made in a 360 degree connection to a metallised connector shell. If the connection is to a screw terminal e.g. demand voltage or registration input the screen can be terminated with a short pig-tail to earth.
4) Connection to the breakout board should be made with a Trio supplied cable, or equivilent with earthed screen and twisted pair connections for all differential signals. The cable between the PCI 208 and the breakout board must not exceed 2.5 metres.
5) The PCI 208, breakout board and breakout cable should not be handled whilst the PC or 24 volt power is connected.

As well as following these guidelines, any installation instructions for other products in the system must be observed.

CHAPTER

4

AXIS CONFIGURATION

Overview The PCI 208 is designed to allow for a wide variety of possible axis configurations. The flexibility is achieved by the physical installation of option modules and the installation, via software, of "Feature Enable Codes". In addition the programmer can adjust, with restrictions, the axis types (ATYPEs) after the initial configuration.

Axis Configuration on Power Up

The axis configuration of the PCI 208 is controlled by the installed feature enable codes (FEC's) and by the fitting of P184 / P185 option modules. The P184 provides 4 voltage outputs for the +/-10 volt controlled servo drives. The P185 provides 8 voltage outputs for +/-10 volt analog servo drives. FEC's are passwords which are unique for each PCI 208. If the FEC for servo operation has been installed the axis can be set (using ATYPE=1) for servo or stepper operation. Any mix of stepper/servo axes can be used.

Axis Configurations:

	Stepper:	Voltage Output Servo:	
Axis 0			Axis 0 & 1 operation doesn't require FEATURE ENABLE CODES (FEC's)
Axis 1			Requires P184
Axis 2	FEC 2 REQUIRED	FEC 10 REQUIRED	
Axis 3	FEC 3 REQUIRED	FEC 11 REQUIRED	
Axis 4	FEC 4 REQUIRED	FEC 12 REQUIRED	Requires P185
Axis 5	FEC 5 REQUIRED	FEC 13 REQUIRED	
Axis 6	FEC 6 REQUIRED	FEC 14 REQUIRED	
Axis 7	FEC 7 REQUIRED	FEC 15 REQUIRED	

Note that axes 0 and 1 can be set to any axis configuration without restriction.

There are 7 types of axis currently available supported:

Axis Description	ATYPE/ COMMS TYPE
Unused Axis	0
Stepper	1
Servo Encoder	2
Reference Encoder	3
Stepper Encoder **	4
Remote CANopen Position Control Servo	18
Remote CANopen Speed Control Servo	19
Remote SERCOS *	24

* future enhancement

** The PCI 208 hardware interface can be set for encoder input or stepper output. When ATYPE=4 the “encoder” interface can therefore only count the stepper pulses. This can be useful for registration.

Note: For volume / OEM applications, Trio can produce custom interfaces for specific customer applications where required.

Changing Axis ATYPEs After Power Up

Each axis of the PCI 208 is set to and ATYPE of 0 (unused), 1 (stepper) or 2 (servo encoder) when the unit powers up, or when the unit is reset.

The initial ATYPE that the axis is set to on power up is configured by the installed feature enable codes. For example on axis 2; if FEC 2 is set axis 2 will be set to ATYPE=1, if FEC 10 is set axis 2 will be set to ATYPE=2, if BOTH FEC codes are set axis 2 will be set to ATYPE=2.

The software will allow the ATYPE to be changed by the programmer with some restrictions:

If the axis FEC set it as a stepper axis, it can be set via software to ATYPE =0, 4, or 1.

If the axis FEC set it as a servo encoder axis, it can be set via software to ATYPE=0,1,2,3 or 4.

The ATYPE can be changed in BASIC by assignment of the axis parameter:

```
ATYPE=1
```
or

```
ATYPE AXIS(4)=0
```

P184/P185 DAC Modules

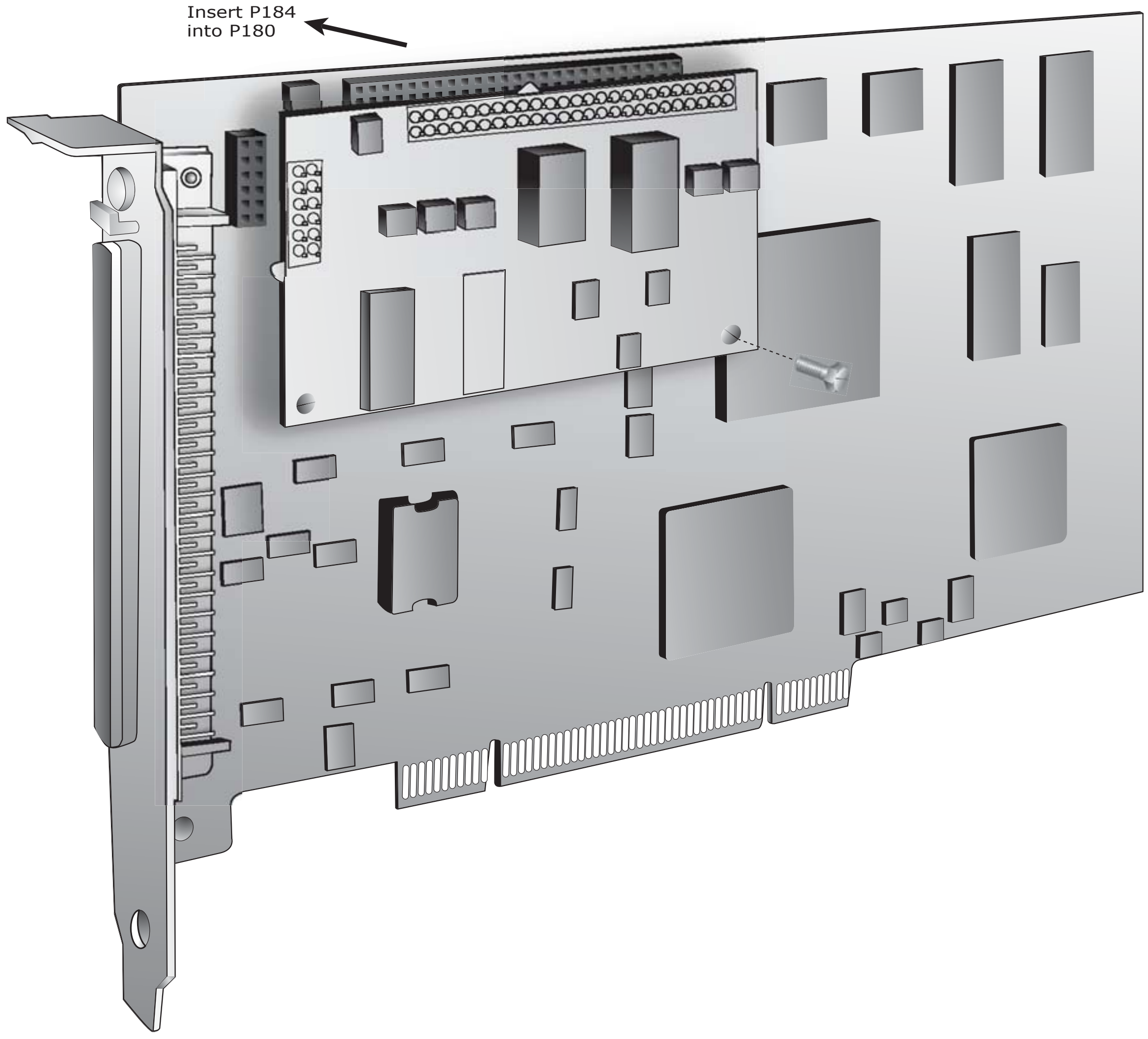
The P184 and P185 DAC Modules are designed to provide 4 or 8 voltage outputs for analog servo drives. The analog outputs are not built on to the main P180 board since they are not required by stepper systems, CAN open servos, SERCOS servo drives, and other digital interfaces.

The P184 and P185 are detected by the software automatically. The user can check the parameter COMMSTYPE SLOT(0) to see the module fitted:

Module Fitted	COMMS TYPE
Unused	0
P184 (4 DAC Modules)	26
P185 (8 DAC Modules)	27

P184/P185 Module Insertion

The P184 or P185 is inserted into the PCI 208 as shown. A standoff pillar secures the board to the PCI 208. Check that the pillar is attached with 2mm screws to both the P184/P185 and the PCI 208.



CHAPTER

5

I/O Modules

Input/Output Modules

General Description

Trio can supply a range of Input/Output Modules for the PCI 208. The *Motion Coordinator* controllers allow for I/O expansion by having a CAN interfaces. The PCI 208 has 2 built-in CAN interfaces of which channel A is always used for Trio I/O modules. This allows the I/O modules to form a network up to 100m in length.

The Trio I/O modules use a dedicated protocol which must be run on a physically separate CAN interface to CANopen or DeviceNet. *If the built-in Trio protocol is not to be used it is necessary to set the CANIO_ADDRESS value to a value other than it's default of 32.*

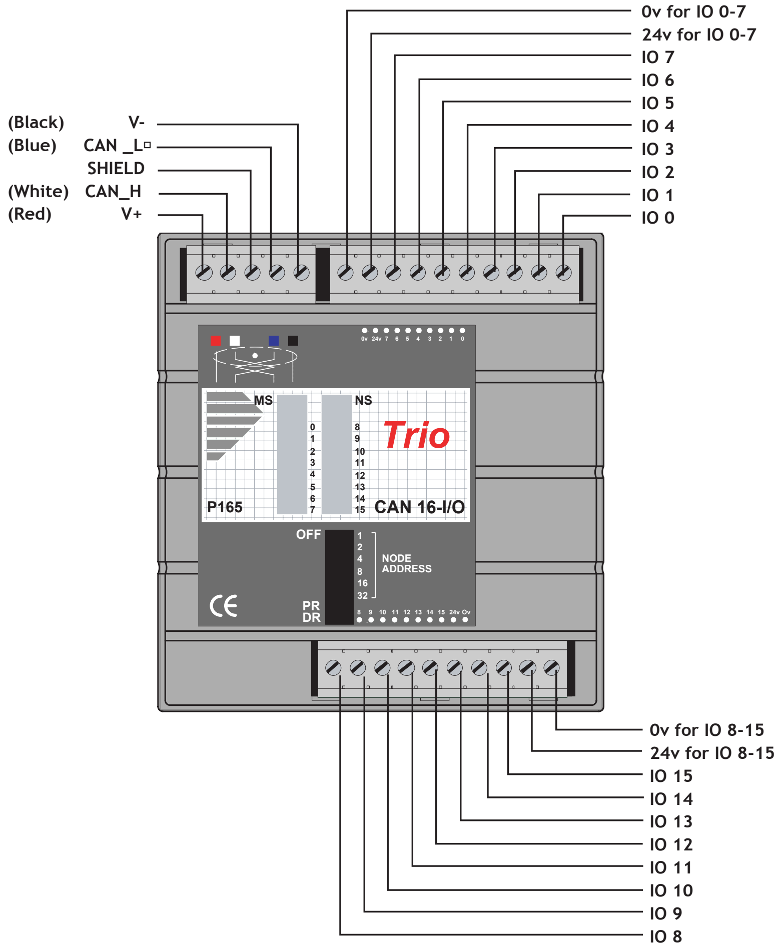
Product:	Product Code:
CAN 16-I/O Module	P315
CAN Analog Input Module	P325

CAN 16-I/O Module (P315)

The CAN 16-I/O Module allows the 24 volt digital inputs and outputs of the *Motion Coordinator* to be expanded in blocks of 16 bi-directional channels.

Up to 16 CAN 16-I/O Modules may be connected allowing up to 256 I/O channels in addition to the internal channels built-in to the PCI 208 *Motion Coordinator*. Each of the 16 channels in each module is bi-directional and can be used either as an input OR as an output.

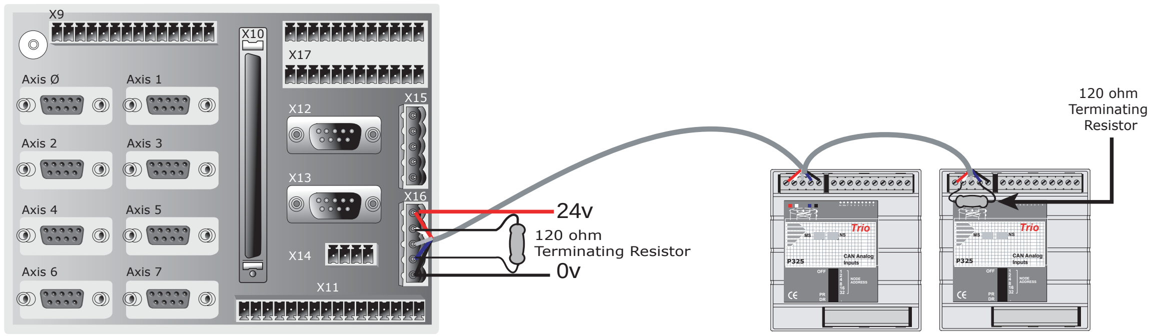
Convenient disconnect terminals are used for all I/O connections.



I/O Connections: The CAN 16-I/O Module has 3 disconnect terminal connectors:

- DeviceNet physical format 5 way CAN connector
- Input/Output Bank 0 - 7 and power supply for bank 0 - 7 on 10 way connector
- Input/Output Bank 8 - 15 and power supply for bank 8 - 15 on 10 way connector.

Bus Wiring The CAN 16-I/O Modules and the *Motion Coordinator* are connected together on a network which matches the physical specification of DeviceNet running at 500kHz. The network is of a linear bus topology. That is the devices are daisy-chained together with spurs from the chain. The total length is allowed to be up to 100m, with drop lines or spurs of up to 6m in length. At both ends of the network, 120 Ohm terminating resistors are required between the CAN_H and CAN_L connections. The resistor should be 1/4 watt, 1% metal film.



The cable required consists of:

Blue/White 24AWG data twisted pair
+ Red/Black 22AWG DC power twisted pair
+ Screen

A suitable type is Belden 3084A.

The CAN 16-I/O modules are powered from the network. The 24 volts supply for the network must be externally connected. The *Motion Coordinator* **does NOT provide the network power.** In many installations the power supply for the *Motion Coordinator* will also provide the network power.

Note: ***It is recommended that you use a separate power supply from that used to power the I/O to power the network as switching noise from the I/O devices may be carried into the network.***

DIP Switch Settings

Address:	Start:	End:
0	32	47
1	48	63
2	64	79
3	80	95
4	96	111
5	112	127
6	128	143
7	144	159
8	160	175
9	176	191
10	192	207
11	208	223
12	224	239
13	240	255
14	256	271
15	272	287

Note that the IO mapping of the PCI 208 starts at 32 rather than 16 which is the usual setting for *Motion Coordinator*'s with 16 or fewer IO's built-in.

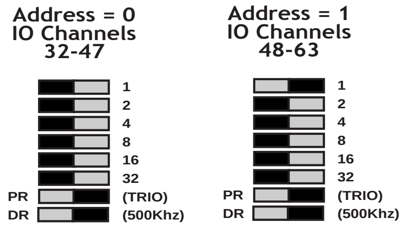
TRIO Protocol

The switch marked PR is set ON to select the standard TRIO protocol.

The top 6 DIP switches on the CAN 16-I/O set the module address. Only addresses 0 - 15 are valid for CAN 16-I/O modules.

The switch marked DR sets the CAN Bus communications rate to 125kHz or 500kHz. Only 500Khz is valid with the TRIO protocol.

The addresses for I/O modules MUST be set in sequence, 0,1,2 etc. Therefore the first two CAN 16-I/O Modules would have switch settings as shown below:



Note: ***The I/O Channels referred to above start at 32. This is because the numbering sequence starts with channels 0 - 31, which are on the PCI 208 master unit itself.***

DUAL IO Mapping:

The PCI 208 has 20 inputs built-in. These are read by the software as inputs 0..19. The I/O module inputs/output start at number 32. There is therefore a set of inputs 20..31 which are unused.

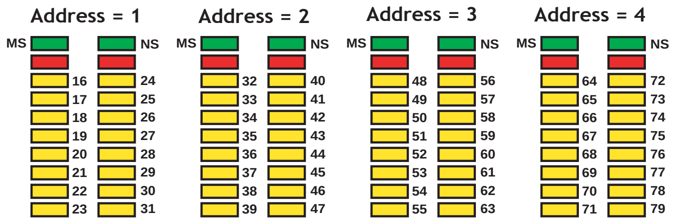
When at least one I/O module is connected to the PCI 208 the first 12 inputs from the I/O module are also mapped into this range automatically by the software.

For example: Input 32 can also be read via input 20, and input 33 can also be read via input 21.

This arrangement allows the CAN I/O module inputs to be used as limit inputs and to be used with the INVERT_IN command.

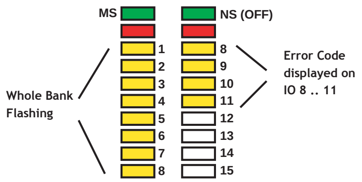
LED Indicators

When NS is ON LEDs marked 0 - 15 represent the input channels 0 - 15 of the module. The actual input as seen by the *Motion Coordinator* software will depends on the I/O modules address:



Error Codes:

When an error occurs on a CAN I/O module, the fault code is represented by a binary number displayed on the leds.



Code	Error Description
1	Invalid Protocol
2	Invalid Module Address
3	Invalid Data Rate
4	Uninitialised
5	Duplicate Address
6	Start Pending
7	System Shutdown
8	Unknown Poll
9	Poll Not Implemented
10	CAN Error
11	Receive Data Timeout

Software Interfacing

The PCI 208 *Motion Coordinator* will automatically detect and allow the use of correctly connected CAN I/O channels. The CAN I/O are accessed with the same `IN` and `OP` commands used to access the built-in I/O on the *Motion Coordinator*. The *Motion Coordinator* sets the system parameter `NIO` which reflects the number of I/O's connected to the system. 3 system parameters are available to facilitate the use of the CAN 16-I/O:

`CANIO_STATUS, CANIO_ADDRESS and CANIO_ENABLE`

When choosing which I/O devices should be connected to which channels the following points need to be considered:

- Inputs 0 - 31 ONLY are available for use with system parameters which specify an input, such as **`FWD_IN`**, **`REV_IN`**, **`DATUM_IN`** etc. See note on page 7 on dual mapping of I/O's
- Outputs 8 - 17 ONLY are available for use with the **`PSWITCH`** command.
- The built-in I/O channels have the fastest operation <1mS
- CAN I/O channels 32 - 79 have the next fastest operation <2mS
- CAN I/O channels 80 - 207 have the next fastest operation <8mS

It is not possible to mix the CAN 16-I/O module which is running the TRIO I/O protocol with DeviceNet equipment on the same network.

Troubleshooting

If the network configuration is incorrect 2 indications will be seen: The CAN 16-I/O module will indicate that it is uninitialised and the *Motion Coordinator* will report the wrong number when questioned:

```
>>? NIO
```

If this is not as expected check:

- Terminating 120 Ohm Network Resistors fitted?
- 24Volt Power to each IO bank required?
- 24Volt Power to Network?
- DIP switches in sequence starting 0,1,2...?
- *Motion Coordinator* CANIO_ADDRESS=32?

P315 Specification

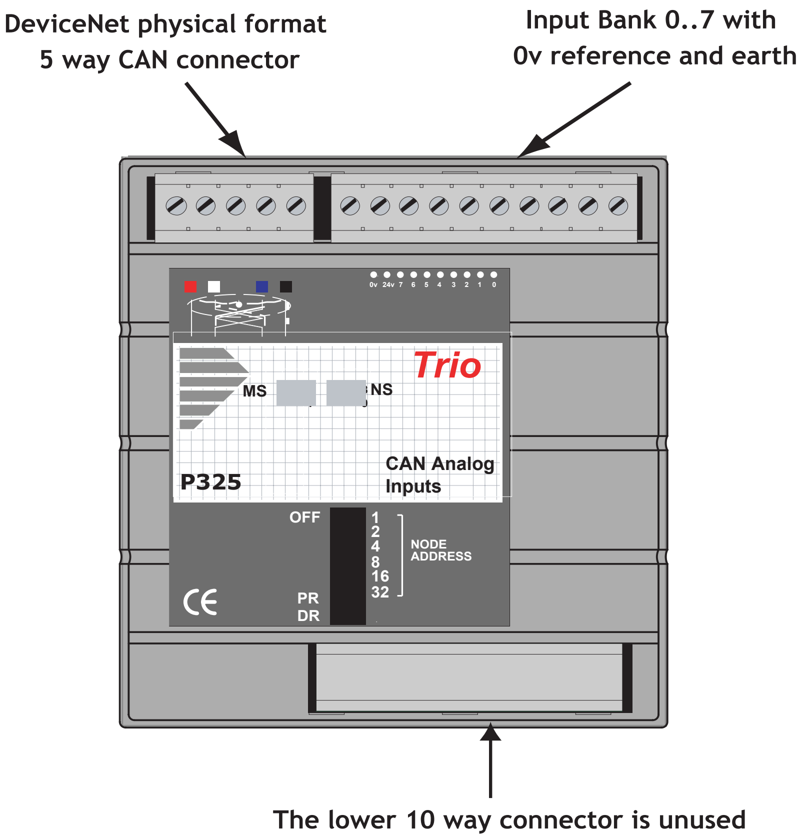
Inputs:	16 24volt inputs channels with 2500v isolation
Outputs:	16 24volt output channels with 2500v isolation
Configuration:	16 bi-directional channels
Output Capacity:	Outputs are rated at 250mA/channel. (1 Amp total/ bank of 8 I/O's)
Protection:	Outputs are overcurrent and over temperature protected
Indicators:	Individual status LED's
Address Setting:	Via DIP switches
Power Supply:	24V dc, Class 2 transformer or power source. 18 ... 29V dc / 1.5W
Mounting:	DIN rail mount
Size:	95mm wide x 45mm deep x 105mm high
Weight:	200g
CAN:	500kHz, Up to 256 expansion I/O channels
EMC:	BSEN50082-2 (1995) Industrial Noise Immunity / BS EN55022 (1995) Class A Industrial Noise Emissions

CAN Analog Inputs Module (P325)

The CAN Analog Input Module allows the PCI 208 *Motion Coordinator* to be expanded with banks of 8 analog input channels. Up to 4 x P325 Modules may be connected allowing up to 32 x 12 bit analog channels in addition to the 4 potential analog inputs if the P184 is fitted. Convenient disconnect terminals are used for the I/O connections.   The input channels are designed for +/-10volt operation. Each bank of 8 channels is opto-isolated from the CAN bus.

I/O Connections

The CAN Analog Input Module has 3 disconnect terminal connectors:



Bus Wiring

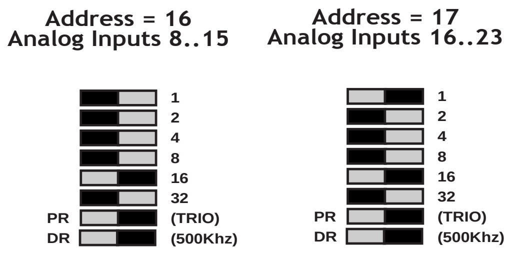
See Can 16 I/O for details

DIP Switch Settings

The switch marked “PR” selects the protocol, but is currently unused as only the TRIO protocol is available.

The switch marked DR sets 125kHz or 500kHz. Only 500Khz is valid with the TRIO protocol.

The addresses for P325 modules MUST be set 16,17,18... in sequence. Therefore the first P325 Module should have the switch setting:



Address:	Start:	End:
16	8	15
17	16	23
18	24	31
19	32	39

Note: P325 modules and P315 (16-I/O) modules may be mixed on the network. The P315 addresses will be 0 to 15 in sequence and the P325 modules will be 16 to 19 in sequence.

LED Indicators

MS	“Module Status”	ON when module powered on OK
NS	“Network Status”	ON when module powered on OK and initialised.

Software Interfacing

The *Motion Coordinator* will automatically detect and allow the use of correctly connected P325 modules. The number of connected analog input channels is reported in the startup message and is also available to the programmer via an additional system parameter “NAIO”.

The analog input resolution is fixed at +10volts to -10volts and will return values -2047 to 2048 to the function AIN(). The first 4 channels are also available as system parameters AIN0, AIN1, AIN2, and AIN3.   This allows these values to be seen using the SCOPE function.

The P325 works “single ended” and does not return differential values.

It is not possible to mix the P325 module which is running the TRIO I/O protocol with DeviceNet equipment on the same network.

Troubleshooting

If the network configuration is incorrect 2 indications will be seen: The P325 module will indicate that it is uninitialised and the *Motion Coordinator* will report the wrong number when questioned:

```
>>? NAIO
```

If this is not as expected check:

- Terminating 120 Ohm Network Resistors fitted?
- 24Volt Power to Network?
- DIP switches in sequence starting 16,17,18...?
- *Motion Coordinator* CANIO_ADDRESS=32?

P325 Specification

Analog Inputs:	8+/-10 volt inputs with 500v isolation from CAN bus
Resolution:	12 bit
Protection:	Inputs are protected against 24v over voltage.
Address Setting:	Via DIP switches
Power Supply:	24V dc, Class 2 transformer or power source. 18 ... 29V dc / 1.5W
Mounting:	DIN rail mount
Size:	95mm wide x 45mm deep x 105mm high
Weight:	200g
CAN:	500kHz, Up to 32 analog input channels
EMC:	BSEN50082-2 (1995) Industrial Noise Immunity / BS EN55022 (1995) Class A Industrial Noise Emissions

CHAPTER

6

SYSTEM SETUP AND DIAGNOSTICS

Preliminary Concepts:

Host Computer	A Windows 2000/XP PC running *Motion* Perfect 2.
Motor	A tuned servo drive / motor configuration for a servo axis or a stepper motor and drive combination
Prompt	When the PCI 208 is ready to receive a new command, the prompt >> will appear on the left hand side of the current line in the "terminal" under the "tools" menu of Motion Perfect
Axis Parameters	Can be written to or read from. For example the proportional gain of a servo axis has the name **`P_GAIN`**. It can be written to: **`P_GAIN=0.5`** or read from: **`PRINT P_GAIN`**. For further information see chapter 9.

System Setup

A control system should be treated with respect as careless or negligent operation may result in damage to machinery or injury to the operator. For this reason the setting up of the system should not be rushed. This section describes a methodical approach to system configuration and is designed to gradually test each aspect of the system in turn, finally resulting in the connection of the motor. If followed cautiously no unexpected situations should arise.

In cases where the setup procedure for servo and stepper systems differ a separate description is provided for each. In multiple axis systems it is advantageous to set up one axis at a time. The following procedure applies both to all *Motion Coordinator* modules.

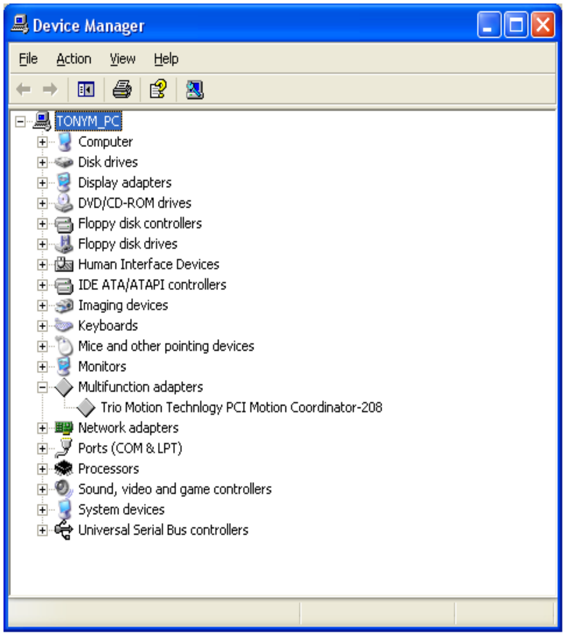
Note:

It is recommended that this section is read in full before attempting to operate the system for the first time.

Preliminary checks

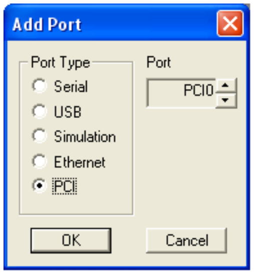
All wiring should be checked for possible misconnection and integrity before any power is applied.

- Install *Motion* Perfect version 2.2.1.10 or higher.
- Install TrioPC Active X version 1.1.0.0 or higher.
- Install the PCI 208 in a free PCI slot with the PC power off.
- Switch on the PC and check PCI 208 is recognised by the PC as "Trio Motion Technology PCI Motion Coordinator 208". The PCI 208 will be found under "Multifunction Adapters" under System Properties->Hardware->Device Manager.



- Run *Motion* Perfect and under the "Options" menu click "Communications". Press "Add" and add a PCI port type.

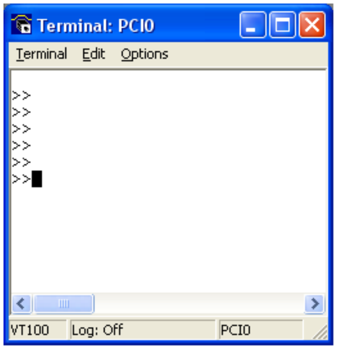
- Delete the default COM1 port type unless you intend to use *Motion* Perfect with serially connected *Motion Coordinator*'s.



Port Type: Serial [ ] USB [ ] Simulation [ ] Ethernet [ ] [✓ PCI]
Port: PCI0

Checking Communications and System Configuration

- Open a terminal window in *Motion* Perfect and PCI 208 should respond with a >> prompt when the "Return" key is pressed.



```
>>
>>
>>
>>
>>
>>
```

VT100 | Log: Off | PCI0

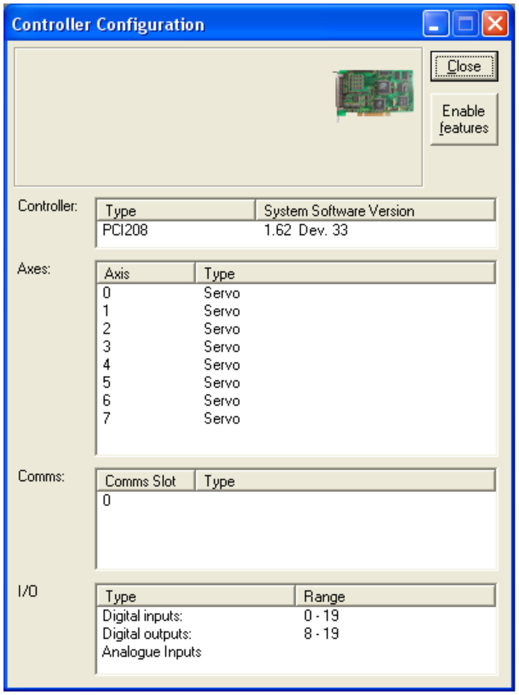
- Click "Connect" under the "Controller" menu.
- If this is the first time you have connected you will need to select the "New Project" option when *Motion* Perfect tries to ensure that your "Project" on the controller matches its copy on disk.
- When the "Project Consistent" message is received in the "Check Project" window you know:

  1 *Motion* Perfect has made a serial connection between your PC and the PCI 208.

  2 *Motion* Perfect has an exact copy of the programs on the PCI 208.

- The controller hardware configuration can now be checked using the “Controller configuration” option under the “Controller” menu. *Motion* Perfect draws a graphical representation of your system, as shown below.

Example:



Controller:

Type	System Software Version
PCI208	1.62 Dev. 33

Axes:

Axis	Type
0	Servo
1	Servo
2	Servo
3	Servo
4	Servo
5	Servo
6	Servo
7	Servo

Comms:

Comms Slot	Type
0	

I/O

Type	Range
Digital inputs:	0 - 19
Digital outputs:	8 - 19
Analogue Inputs	

This message would be produced by a *Motion Coordinator* PCI 208 with the following configuration:

- System Software version 1.62
- Axes 0..7 set to servo operation using feature enable codes.
- No CAN I/O modules are connected.

Check that the system description corresponds with the modules that are actually present. If this is not so, check the CANBus and feature enable codes and the settings of the address switches on any CAN or axis expander modules.

Input/Output Connections:

- Check each of the 24v input connections with a meter then connect them to the controller.
- Test each of the input channels being used for correct operation in turn. These may be easily viewed in the I/O window. Use "IO Status" under the "Tools" menu.
- Switch each output being used in turn for correct operation. These may be easily set with the IO status window.

Connecting a Servo Motor to a P181 Breakout Board

Note: ***This description assumes the motor / drive combination has been already tested and is functioning optimally.***

Each servo axis should be connected in turn to the X1..X8 encoder pins.

- With the servo drive off or inhibited <u>connect the motor encoder only</u> (or the encoder emulation output from the servo drive).
- Check the encoder counts both up and down by looking at the measured axis position **`MPOS`** in the Axis parameter window of *Motion* Perfect ("Axis parameters" under the "Tools" menu) whilst turning the axis by hand.
- Ensure the **`SERVO`** axis parameter is set OFF (0) in the Axis parameter and that the **`DAC`** axis parameter is set to 0. It may be necessary to use the scroll buttons to view these parameters. This will force 0 volts out of the +/-10v output for the axis. Now connect the servo drive to the V+/V- connections.
- Enable the servo drive by clicking the "Drives disabled" button on the control panel. If the axis runs away the motor/drive combination must be re-checked. (Note: clicking "Drives disabled/ Drives enabled" is equivalent to issuing a **`WDOG=ON`** or **`WDOG=OFF`** command).
  The servo motor should now be powered and is likely to be creeping in one direction as the position servo has been switched OFF.
- Set a small positive output voltage by setting **`DAC=25`**. The motor should then move slowly forward - Check the encoder is counting up by looking at the **`MPOS`** axis parameter. If this is correct check that the motor reverses and the encoder counts down when **`DAC=-25`**.
- If the encoder counts down when a positive **`DAC`** voltage is applied. The motor or position feedback needs to be reversed.

  This can be achieved by:

  - Swap A and /A connections on the encoder input, or

- Swap BOTH motor terminals and tacho terminals (DC motors only!) On many digital brushless motors the direction can be reversed by a drive setting, or
- If the drive has differential inputs, reverse the voltage as it enters the drive. (This can cause problems with some servo-drives. The VIO GND pins of the Breakout board/PCI 208 are internally connected inside the *Motion Coordinator* so the axis voltage outputs cannot float relative to each other), or
- set a negative **`PP_STEP`** axis parameter. (This is not possible using SSI encoders)
- Set a negative **`DAC_SCALE`** axis parameter. (Trio recommend wiring motors consistently and not using a negative DAC_SCALE to correct wiring errors)

We are now ready to apply the position servo as described below.

Setting Servo Gains

The servo system controls the motor by constantly adjusting the voltage output which gives a speed demand to the servo drive. The speed demand is worked out by looking at the measured position of the axis from the encoder comparing it with the demand position generated by the *Motion Coordinator*.

The demand position is constantly being changed by the *Motion Coordinator* during a move. The difference between the demand position (Where you want the motor to be) and the measured position (Where it actually is) is called the following error.

The controller checks the following error typically 1000..4000 times per second and updates the voltage output according to the "servo function". The *Motion Coordinator* has 5 gain values which control how the servo function generates the voltage output from the following error.

Default Settings:

Gain	Parameter Name	Value
Proportional Gain	`P_GAIN`	1.0
Integral Gain	`I_GAIN`	0.0
Derivative Gain	`D_GAIN`	0.0
Output Velocity Gain	`OV_GAIN`	0.0
Velocity Feedforward Gain	`VFF_GAIN`	0.0

A simple test program can be used to generate movement to and fro for examination of the motion profile generated on an oscilloscope. The oscilloscope should be connected to the tacho or velocity output from the servo drive.

Example:

```
PRINT "Enter Axis Number ":INPUT VR(0)
BASE(VR(0))
SPEED=20000
ACCEL=200000
DECEL=200000
loop:
  MOVE(1000)
  WAIT IDLE
  WA(100)
  MOVE(-1000)
  WAIT IDLE
  WA(100)
GOTO loop
```

The editor built into *Motion* Perfect may be used to enter the test program. Click on Program, New from the pull down menus and enter a program name, replacing the name **`UNTITLEDx`**. Now click on the EDIT button and an edit window will be opened where the program shown above may be typed in. See the *Motion* Perfect section for more details on how to use the editor. Once the program is entered, it can be run by clicking on the red button next to its name or the RUN button in the Controller Status panel.

The servo gain parameters may be set to achieve the desired response from the servo system. The desired response can vary depending on the type of machine.

Different gain settings can be used to obtain:

Smoothest motor running

This can be achieved by using low proportional gain values, adding output velocity gain adds smoothing damping at the expense of higher following errors.

Low following errors during complete motion cycle

This can be achieved by using velocity feed forward to compensate for following errors together with higher proportional gains.

Exact achievement of end points of moves

This can be achieved by using integral gain in the system together with proportional gain. However overshoot will occur at the end of rapid deceleration.

Typically a combination of the above is required.

Note: ***The system should be set with proportional gain alone firstly starting with the default value of 1.0   The other gains should then be introduced if necessary according to the descriptions which follow.***

Proportional Gain

Description The proportional gain creates an output voltage, Op that is proportional to the following error E.

$$Op = Kp \times E$$

Axis parameter is called `P_GAIN`

Syntax: `P_GAIN=0.8`

Note: ***All practical systems use proportional gain, many use this gain parameter alone.***

Integral Gain

Description The Integral gain creates an output Oi that is proportional to the sum of the errors that have occurred during the system operation.

$$Oi=Ki \times SE$$

Integral gain can cause overshoot and so is usually used only on systems working at constant speed or with a slow acceleration.

Axis parameter is called `I_GAIN`

Syntax: `I_GAIN=0.0125`

Derivative Gain

Description This produces an output Od that is proportional to the change in the following error and speeds up the response to changes in error whilst maintaining the same relative stability.

$$Od = Kd \times DE$$

This gain may create a smoother response. High values may lead to oscillation.

Axis parameter is called `D_GAIN`

Syntax: `D_GAIN=5`

Output Velocity Gain

Description This increases the system damping, creating an output that is proportional to the change in measured position.

$$Oov = Kov \ x \ DPm.$$

This parameter can be useful for smoothing motions but will generate high following errors. Note that a NEGATIVE OV_GAIN is required for damping.

Axis parameter is called `OV_GAIN`

Syntax: `OV_GAIN=-5`

Velocity Feed Forward Gain

Description As movement is created by following errors at high speed the following error can be quite appreciable. To overcome this the Velocity Feed Forward creates an output proportional to the change in demand position so creating movement without the need for a following error.

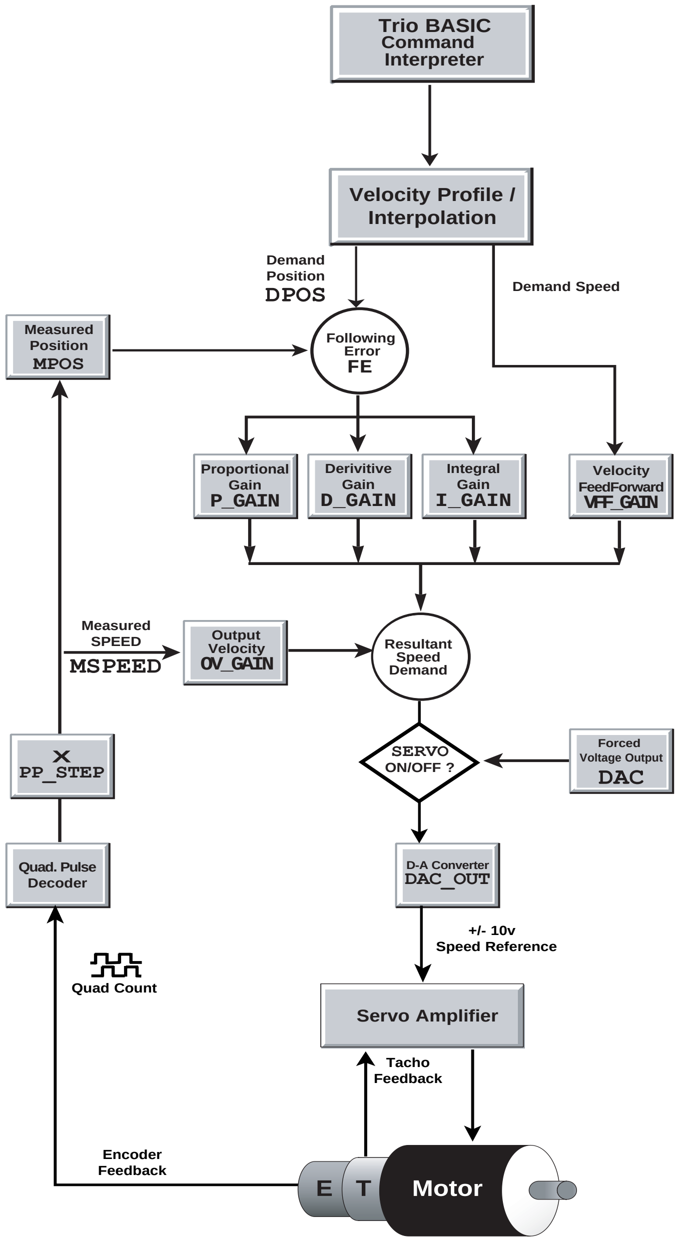
$$Ov = Kvff \ x \ DPd$$

Axis parameter is called `VFF_GAIN`

Syntax: `VFF_GAIN=10`

The VFF_GAIN parameter can be set by minimising the following error at a constant machine speed AFTER the other gains have been set.

Servo Loop Diagram



Diagnostic Checklists

Problem	Potential reasons
Motion Perfect flashes “Enable” button	• Following error on at least one axis. The axis demand position and measured position exceed the programmed limit
Motor runs away without issuing a move command	• tacho/drive polarity • encoder/controller polarity • gains (drive and/or controller)
Motor runs away upon issu-ing a move command	• tacho feedback • encoder feedback • gains (drive and/or controller)
Motor does not move upon issuing a move command	• wiring (enables/inhibits/limits on drive and controller) • check status on all axes • drive power • feedhold applied • speed, acceleration and/or deceleration set to zero • servo set off/watchdog set off • gains (drive and/or controller) • axis is already running a move which has not completed - Check MTYPE and NTYPE
Axis goes out on following error after a time	• speed being requested requires more than 10v - check drive tacho gain and motor/ drive speed characteristics. • drive shutting down on current limit after a time

Problem	Potential reasons
Axis losing position	• encoder coupling • encoder signal (wire length, differential/single ended encoder) • mechanics
Motion Perfect cannot "connect" with the controller	• Controller running a program which transmits on serial channel 0. If this prevents *Motion* Perfect connecting to the controller, open Terminal screen in *Motion* Perfect unconnected mode and type a "halt" command at the command prompt. • *Motion* Perfect is not set to use the PCI port: Set under "options" menu • Check *Motion* Perfect version. The latest version can be downloaded from `www.triomotion.com`

CHAPTER

7

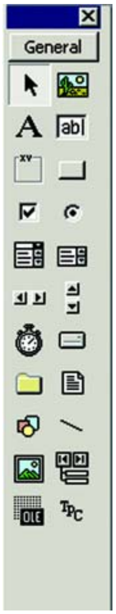
Active X Program Examples

Programming Example:

Simple use of Trio PC Motion ActiveX control in Microsoft Visual BASIC 6.0

Scope This programming example shows how to use the Trio PC Motion ActiveX control in a Microsoft Visual BASIC 6.0 application. It demonstrates how to access system and axis parameters, how to read and write digital I/O and how to perform basic moves.

Configuring Visual BASIC

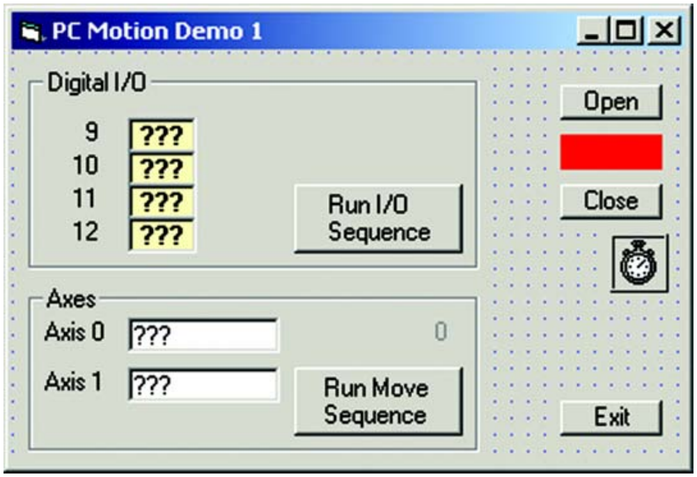


Trio PC Motion ActiveX control must already be installed on the PC.

In order to use the Trio PC Motion ActiveX control Visual BASIC needs to be configured to use the control. To do this, select Project / Components from Visual BASIC's main menu to display the components dialog. The components dialog contains a list of all the controls installed on the PC. Make sure that the check box next to "TrioPC ActiveX Control module" is checked then click on the OK button to close the dialog. The Trio PC Motion ActiveX control icon will appear in the component palette (usually at the bottom, right).

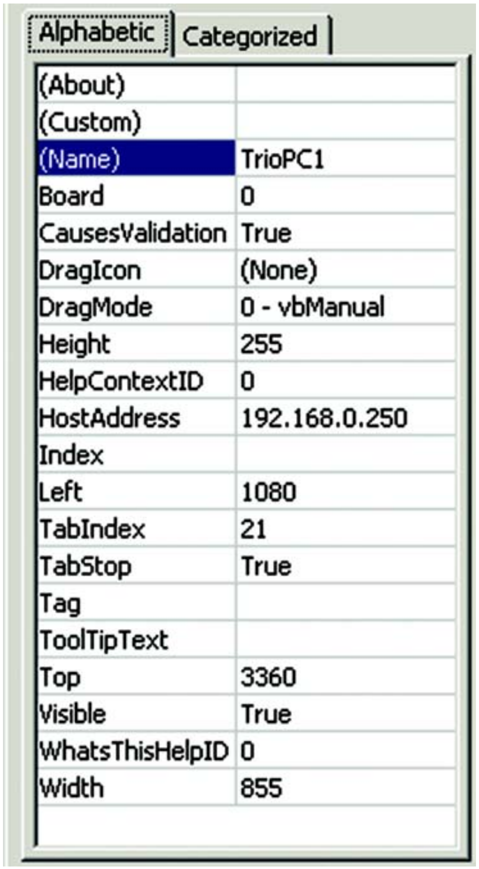
Building the application

Main Form - Visual



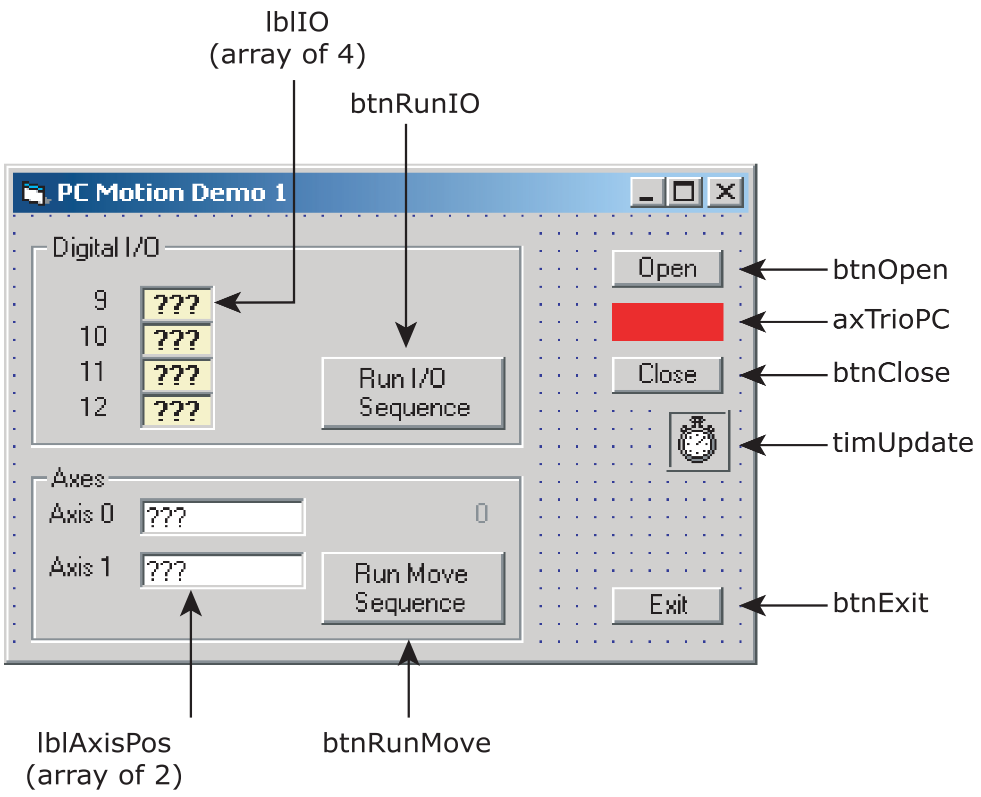
Usually the first stage in building an application is creating the main form (the sample application only has one form). Controls are placed on the form from the component palette the properties of each component being adjusted as appropriate. When the Trio PC Motion ActiveX control is placed on a form it appears as a red, resizable rectangle. The available properties are as shown above.

Only the “Board” and “HostAddress” properties are custom the the control. The “Board” property is only used for a PCI bus connection and the “HostAddress” property is only used for an Ethernet connection. The sample program uses a USB connection so these properties are not used.



(About)	
(Custom)	
(Name)	TrioPC1
Board	0
CausesValidation	True
DragIcon	(None)
DragMode	0 - vbManual
Height	255
HelpContextID	0
HostAddress	192.168.0.250
Index	
Left	1080
TabIndex	21
TabStop	True
Tag	
ToolTipText	
Top	3360
Visible	True
WhatsThisHelpID	0
Width	855

The completed main form for the application is shown below, annotated to show the control names used in the example application.



Main Form - Code

The code is best written in stages in order to make testing easier. The starting point for this application was handlers for the “Exit”, “Open” and “Close” buttons. The operation of the “Open” and “Close” buttons should be tested before any of the code which accesses the Trio PC Motion component. The handlers for other buttons are then added one at a time, together with any helper functions, and the functionality tested for each group of controls as the Digital I/O group is independent from the Axes (moves) group in this application.

Opening and closing the Trio PC Motion component.

Calling the “Open” should open the connection to the controller. A successful call of the “Open” method should cause the visible Trio PC Motion component to change colour to green (remember to refresh it). The open/closed state of the component can be checked using the “IsOpen” method. Calling the “Close” method will close the connection to the controller. After a successful call of the “Close” the colour of the visible Trio PC Motion component will be red.

The connection should always be closed before the application exits. This is done by putting a conditional call to the “Close” method in the “Form_Unload” routine for the main form in the application. To automatically open the connection a call to the “Open” method can be put in the Form_Load routine of the main form in the example application.

Reading and writing Digital I/O

Digital inputs can be read using the “In” method. This can read the states of a range of inputs. The routine “ReadIO” in this application shows how this is done and how to separate out individual bits from the number returned by the “In” method call.

Digital outputs can be written using the “Op” method. The “Op” method only writes to a single digital output. Writing to a range of outputs can be done by using a program loop such as the FOR loop used in the IO section of the “ProcessStateMachine” routine in the example application.

Reading and writing system parameters

System parameters are read using the “GetVariable” method. The example application demonstrates this when the “WDOG” parameter is read at the beginning of the “InitAxes” routine. The value of the parameter is always returned as a double.

System parameters are written using the “SetVariable” method. The example application demonstrates this when the “WDOG” parameter is written at the beginning and at the end of the “InitAxes” routine.

Reading and writing axis parameters

This uses the same “Get/SetVariable” methods as are used for reading and writing system parameters. The difference is that the parameter read or written is the one from the current base axis. To change the base axis the “Base” method must be used. The “Base” method can specify a single axis or several axes (in an array). If more than one axis is specified then parameter reads and writes use the first axis specified (in element 0 of the array). Examples of this can be seen in the “ReadAxisPositions”, the “ProcessStateMachine” and the “InitAxes” routines in the example

Performing Moves

Moves can be specified using the following methods:

CamBox - Cam shape move linked to another axis

Cam - Cam shape move

Connect - Ratio move linked to another axis

Datum - Datuming move sequence

Forward - Continuous position move in positive direction

Reverse - Continuous position move in reverse direction

MoveAbs – absolute

MoveRel – relative (equivalent to MOVE in Trio BASIC)

MoveCirc – circular

MoveHelical – helical

MoveLink – linked to another axis

MoveModify – modify move end position

The example application shows examples of the “MoveAbs” and “MoveRel” methods in the “ProcessStateMachine” routine. Both of these methods specify an array of move positions/distances. The positions/distances are applied to the axes specified in the last call to the “Base” method. If a “Base” method call specifies axes 1, 2 & 5 (in that order) then a call to “MoveRel” with one axis specified would use axis 1, two axes specified would use axis 1 and axis 2, three axes specified would use axis 1, axis 2 and axis 5.

Moves are loaded into a buffering system for each axis. This is capable of containing two moves, the current move and the next move. If a move is written to an axis which already has a next move specified then the move method called will not return until it is possible for the move to be loaded into the next move position in the move buffer. This can have the effect of hanging the GUI of the calling application unless moves are taken to prevent this. Checking that the “NTYPE” axis parameter is zero on all axes involved in the move is one way of doing this although it may have a side effect of introducing pauses in the motion if the moves are short. The example application demonstrates this method in the “ProcessStateMachine” routine.

Timed operations

Visual BASIC has no means of waiting for a time period therefore any timed operations need to be performed in response to events (usually generated by a timer component). This must be done in order to keep the GUI active during ant "Wait" periods. In the example application the "ProcessStateMachine" method is called by the handler for a timer event. This is to allow the I/O sequence to be performed slowly enough for the counting operation to be seen on the GUI and also to allow the display of axis demand position during the Move sequence.

Program Code

The full source code for the program is shown below:

```
' Demo program for Trio PC Motion ActiveX control
'
'   Trio PC Motion Control version 1.1.0.2 or later is required
'
' Program demonstrates opening and closing connection to controller
' as well as reading and writing axis parameters and I/O states.
'
' I/O and Move sequences are controlled using a state machine run from
' a timer.  This is done to make sure that the values displayed on the PC
' are updated on a regular basis and the PC is not held up waiting for a
' command to complete.  It also allows the I/O sequence to run at a slow
' speed.

Option Explicit                ' Forces variables to be declared before being used

' Global Variables
Dim g_bRunningIO As Boolean     ' Flag to control running of I/O sequence
Dim g_bRunningMove As Boolean   ' Flag to control running of Move sequence
Dim g_nIOCount As Integer       ' I/O counter used in I/O sequence
Dim g_nMoveNo As Integer        ' Move counter used in Move sequence

' Define connection defaults
Const gk_sDefaultHostAddress As String = "192.168.0.111"
Const gk_nDefaultPciBoard As Integer = 0
Const gk_nDefaultLink As Integer = 0    ' USB link
Const gk_nDefaultMode As Integer = 0    ' Synchronous (Token) mode
Const gk_nMaxAxes As Integer = 24
Const gk_nAxesInUse As Integer = 2

Private Sub Form_Load()
    ' Initialise global variables
    g_bRunningIO = False
    g_bRunningMove = False
    timUpdate.Enabled = True
    g_nIOCount = 0
    g_nMoveNo = 0
    UpdateButtonStates
End Sub

Private Sub btnClose_Click()
    ' Handler for close button - closes connection to controller
    If axTrioPC.IsOpen(gk_nDefaultMode) Then
        axTrioPC.Close (gk_nDefaultMode)
    End If
    UpdateButtonStates
    Refresh
End Sub

Private Sub btnOpen_Click()
    ' Handler for Open button - opens connection to controller
    Dim bOpen As Boolean

    axTrioPC.HostAddress = gk_sDefaultHostAddress    ' Only needed for ethernet
    axTrioPC.Board = gk_nDefaultPciBoard             ' Only needed for PCI
    bOpen = axTrioPC.Open(gk_nDefaultLink, gk_nDefaultMode)
    UpdateButtonStates
    Refresh

End Sub

Private Sub btnRunIO_Click()
    ' Handler for RunIO button - starts I/O sequence
    g_nIOCount = 0          ' Initialise counter
    ReadIO
    g_bRunningIO = True     ' Indicate I/O sequence running to state machine processor
    UpdateButtonStates
End Sub

Private Sub btnRunMove_Click()
    ' Handler for RunMove button - starts Move sequence
    InitAxes                ' Initialise axis parameters
    g_nMoveNo = 0           ' Initialise counter
    lblMoveNumber.Caption = g_nMoveNo
    lblMoveNumber.Refresh
    ReadAxisPositions
    g_bRunningMove = True   ' Indicate move sequence running to state machine processor
    UpdateButtonStates
End Sub

Private Sub btnExit_Click()
    ' Handler for Exit button
    End
End Sub

Private Sub UpdateButtonStates()
    ' Update button enable states depending on connection state and running sequences
    Dim bOpen As Boolean
    bOpen = axTrioPC.IsOpen(0)

    btnOpen.Enabled = Not bOpen
    btnClose.Enabled = bOpen And Not (g_bRunningIO Or g_bRunningMove)
    btnRunIO.Enabled = bOpen And Not g_bRunningIO
    btnRunMove.Enabled = bOpen And Not g_bRunningMove
End Sub

Private Sub Form_Unload(Cancel As Integer)
    ' Make sure link to controller is closed when form closes
    If axTrioPC.IsOpen(gk_nDefaultMode) Then
        axTrioPC.Close (gk_nDefaultMode)
    End If
End Sub

Private Sub ReadIO()
    ' Read Input values used in I/O sequence
    Dim lIO As Long     ' Used to hold value IO value returned from controller
    Dim lMask As Long   ' Used to mask out individual bit values
    Dim nBit As Integer ' Used to count bits
    Dim nMBR As Integer ' Used as dummy for MessageBox return value

    If axTrioPC.IsOpen(gk_nDefaultMode) Then
        If axTrioPC.In(9, 12, lIO) Then     ' Read bits 9 to 12

            ' Cycle through returned bits (0 to 3 equivalent to inputs 9 to 12)
            ' and display bit values in lblIO control array
            lMask = 1
            For nBit = 0 To 3
                If lIO And lMask Then
                    lblIO(nBit).Caption = "On"
                    lblIO(nBit).ForeColor = &HFF&       ' Red
                Else
                    lblIO(nBit).Caption = "Off"
                    lblIO(nBit).ForeColor = &HFF0000    ' Blue
                End If
                lMask = lMask * 2       ' Move mask to next bit
                lblIO(nBit).Refresh     ' Make sure new value is displayed
            Next nBit

        Else
            nMBR = MsgBox("Error reading IO")
        End If
    Else
        nMBR = MsgBox("Connection not open")
    End If

End Sub
```

```
Private Sub ReadAxisPositions()
    ' Read demand positions of axes used in Move sequence
    Dim nAxis As Integer                 ' Used for axis number
    Dim nAxes(gk_nMaxAxes) As Integer    ' Array used to set Base axis/axes
    Dim dPosition As Double              ' Used for demand position read from controller
    Dim nMBR As Integer                  ' Used as dummy for MessageBox return value

    ' Initialise Axes array to all zero
    For nAxis = 0 To gk_nMaxAxes - 1
        nAxes(nAxis) = 0
    Next nAxis

    If axTrioPC.IsOpen(gk_nDefaultMode) Then
        ' Scan through all axes in use to read DPOS and show value in lblAxisPos
        '   control array
        For nAxis = 0 To gk_nAxesInUse - 1
            nAxes(0) = nAxis        ' Set up Axes array for Base command
                                    '   (single value in element 0)
            If axTrioPC.Base(1, nAxes) Then
                If axTrioPC.GetVariable("DPOS", dPosition) Then
                    lblAxisPos(nAxis).Caption = Int(dPosition)
                    lblAxisPos(nAxis).Refresh   ' Make sure display is updated
                Else
                    nMBR = MsgBox("Error reading axis positions")
                    nAxis = gk_nAxesInUse   ' Set condition to force loop exit
                End If
            Else
                nMBR = MsgBox("Error setting base for reading axis positions")
                nAxis = gk_nAxesInUse   ' Set condition to force loop exit
            End If
        Next nAxis
    Else
        nMBR = MsgBox("Connection not open")
    End If
 End Sub

Private Sub timUpdate_Timer()
    ' Handler for timer
    If axTrioPC.IsOpen(gk_nDefaultMode) Then
        ProcessStateMachine     ' State machine controls running of I/O and
                                '   Move sequences
    End If
End Sub

Private Sub ProcessStateMachine()
    ' State machine controls running of I/O and Move sequences
    ' Called at regular intervals by the timer
    Dim nMask As Integer            ' Used to mask out individual I/O bit values
    Dim bOK As Boolean              ' Used for value returned by Trio PC commands.
    Dim nIO As Integer              ' Used as I/O bit number
    Dim nBases(gk_nMaxAxes) As Integer  ' Array used for storing axis numbers used
                                        '   by the Base command
    Dim nAxis As Integer            ' Axis number
    Dim bOkToMove As Boolean        ' Flag to indicate that it's OK to load moves
    Dim dReadVal As Double          ' Used for value returned by GetVariable command
    Dim dMoveRef As Double          ' Reference value used in calculation of move
                                    '   distances
    Dim dMovePos(gk_nMaxAxes) As Double ' Array used for move distances

    Const kdRefDistance As Double = 500#    ' Fixed value used in calculating moves
    Const knMoves As Integer = 5    ' Number of moves in Move sequence

    ' IO
    If g_bRunningIO Then
        nMask = 1
        ' Set I/O bits 9 to 12 to represent the value in the I/O counter
        For nIO = 0 To 3
            bOK = axTrioPC.Op(nIO + 9, g_nIOCount And nMask)
            nMask = nMask * 2           ' Move mask to next bit
        Next nIO
        ReadIO
        g_nIOCount = g_nIOCount + 1     ' Increment move count
        If g_nIOCount > 15 Then         ' Check for end of move sequence
            g_bRunningIO = False        ' End of sequence so flag it
            UpdateButtonStates
        End If
    End If

    ' Moves
    If g_bRunningMove Then
        ' Scan through NTYPE (next move type) on all axes to make sure that moves
        ' can be loaded into the move buffer without holding up the PC.
        bOkToMove = True
        For nAxis = 0 To gk_nAxesInUse - 1
            nBases(0) = nAxis           ' Set single value in Bases array to select
                                        '   axis to read
            bOK = axTrioPC.Base(1, nBases)
            If bOK Then
                bOK = axTrioPC.GetVariable("NTYPE", dReadVal)
                If bOK Then
                    If dReadVal <> 0# Then
                        ' Next move still loaded on this axis so mark as not OK
                        '   to load
```

```
                        bOkToMove = False
                    End If
                End If
            End If
        Next nAxis

        If bOkToMove Then
            dMoveRef = kdRefDistance * (g_nMoveNo + 1)  ' Move distance is based
                                                        '   on move number

            If g_nMoveNo = 0 Then
                ' First move
                ' Set up Bases for all axes in use and set all moves to zero
                For nAxis = 0 To gk_nAxesInUse - 1
                    nBases(nAxis) = nAxis
                    dMovePos(nAxis) = 0
                Next nAxis
                bOK = axTrioPC.Base(gk_nAxesInUse, nBases)
                ' First move is absolute to reset position to zero on all used axes
                bOK = axTrioPC.MoveAbs(gk_nAxesInUse, dMovePos)
            ElseIf g_nMoveNo > 0 And g_nMoveNo < knMoves Then
                ' Other moves
                ' Set up bases for all axes and set move distances to
                '   calculated values
                For nAxis = 0 To gk_nAxesInUse - 1
                    nBases(nAxis) = nAxis
                    dMovePos(nAxis) = dMoveRef * (nAxis + 1)
                Next nAxis
                bOK = axTrioPC.Base(gk_nAxesInUse, nBases)
                ' All other moves are relative
                bOK = axTrioPC.MoveRel(gk_nAxesInUse, dMovePos)
            Else
                ' Terminate move sequence
                g_bRunningMove = False
                UpdateButtonStates
            End If
            lblMoveNumber.Caption = g_nMoveNo
            lblMoveNumber.Refresh
            g_nMoveNo = g_nMoveNo + 1   ' Increment move number
        End If
        ReadAxisPositions
    End If
End Sub

Private Sub InitAxes()
    ' Performs axis initialisation for all axes in use
    Dim bOK As Boolean      ' Used as return value for TrioPC motion commands
    Dim nAxis As Integer    ' Used as Axis number
    Dim nBases(gk_nMaxAxes) As Integer  ' Array used to store axis numbers for
                                        '   Base command
    Dim dReadVal As Double          ' Used for value returned by GetVariable command

    If axTrioPC.IsOpen(gk_nDefaultMode) Then

        ' Modifications should be done with watchdog off
        bOK = axTrioPC.GetVariable("WDOG", dReadVal)   ' Read value should be 0 or 1
        If dReadVal > 0.5 Then
            bOK = axTrioPC.SetVariable("WDOG", 0#)     ' Turn watchdog off
        End If

        ' Scan through all axes in use and set axis parameters
        For nAxis = 0 To gk_nAxesInUse - 1
            nBases(0) = nAxis       ' Set correct (single) axis for Base command
            bOK = axTrioPC.Base(1, nBases)

            ' Modify parameters
            If bOK Then
                bOK = axTrioPC.SetVariable("SERVO", 0#)
            End If
            If bOK Then
                bOK = axTrioPC.SetVariable("ATYPE", 0#)
            End If
            If bOK Then
                bOK = axTrioPC.SetVariable("UNITS", 100#)
            End If
            If bOK Then
                bOK = axTrioPC.SetVariable("SPEED", 5000#)
            End If
            If bOK Then
                bOK = axTrioPC.SetVariable("ACCEL", 3000#)
            End If
            If bOK Then
                bOK = axTrioPC.SetVariable("DECEL", 7000#)
            End If
            If bOK Then
                bOK = axTrioPC.SetVariable("FELIMIT", 10000#)
            End If
            If bOK Then
                bOK = axTrioPC.SetVariable("SERVO", 1#)
            End If
        Next nAxis
        bOK = axTrioPC.SetVariable("WDOG", 1#)  ' Modifications complete so
                                                '   turn watchdog on
    End If
End Sub
```

CHAPTER

8

Trio BASIC Commands

Motion and Axis Commands

ACC

Type: Axis Command

Syntax: `ACC(rate of acc)`

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

Description: Sets both the acceleration and deceleration rate simultaneously

Parameters:

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

Example: `ACC(100)`

ADD_DAC

Type: Axis Command

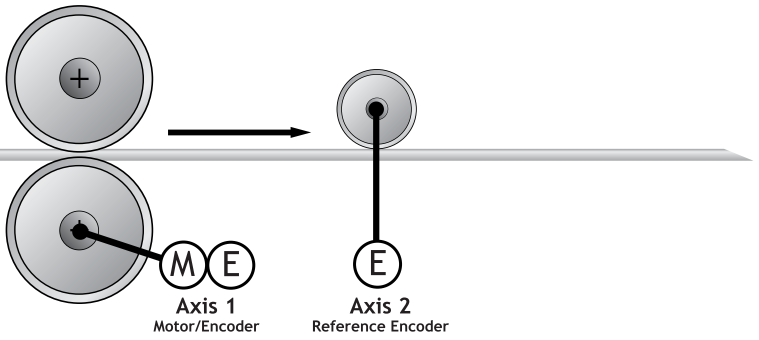
Syntax: `ADD_DAC(axis)`

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

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

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

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



Example 1:

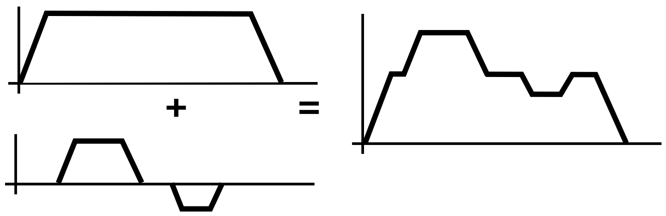
```
ADD_DAC(2) AXIS(1)  ' Combine axis(2) DAC with axis(1)
ADDAX(1) AXIS(2)  ' Superimpose axis 2 profile on axis 1
```

ADDAX

Type: Command

Syntax: **`ADDAX(axis)`**

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



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

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

Parameter: **axis:** Axis to superimpose

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

Example 1:

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

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

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

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

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

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

AXIS

Type: Modifier

Syntax: **AXIS(expression)**

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

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

Parameters:

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

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

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

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

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

See Also: **BASE()**

BASE

Type: Motion Command

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

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

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

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

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

Parameters:

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

Example 1:

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

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

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

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

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

This will set the internal array of 16 axes to:

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

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

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

This example is from an MC206 with 8 axes.

CAM

Type: Axis Command

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

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

Parameters:

`start point:`	The cam table may be used to hold several profiles and/or other information. To allow freedom of use each command specifies where to start in the table.
`end point:`	Specifies end of values in table. Note that 2 or more `CAM()` commands executing simultaneously can use the same values in the table.
`table multiplier:`	The table values are absolute positions from the start of the motion and are normally specified in encoder edges. The table multiplier may be set to any value to scale the values in the table.
`distance:`	The distance factor controls the speed of movement through the table. The time taken to execute the `CAM()` command is dependent on the current axis `SPEED` and this distance (which is in user units).

Say for example the system is being programmed in mm and the speed is set to 10mm/sec. If a distance of 100mm is specified the CAM command will take 10 seconds to execute. The speed may be changed at any time to any value as with other motion commands. The `SPEED` is ramped up to using the current `ACCEL` value. To obtain a `CAM` shape where `ACCEL` has no effect the value should be set to at least 1000 times the `SPEED` value (assuming the default `SERVO_PERIOD` of 1ms).

Example: Motion is required to follow the POSITION equation:

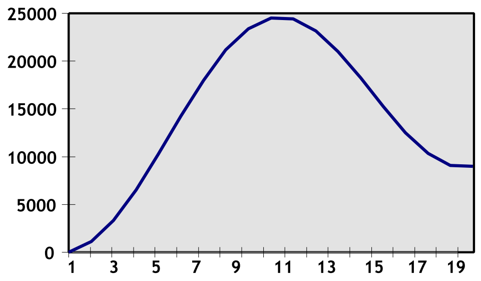
```
t(x) = x*25 + 10000(1-cos(x))
```

Where x is in degrees. This example table provides a simple oscillation superimposed with a constant speed. To load the table and cycle it continuously the program would be:

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

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

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



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

CAMBOX

Type: Axis Command

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

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

Parameters:

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

link options: Bit Values:

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

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

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

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

Note:

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

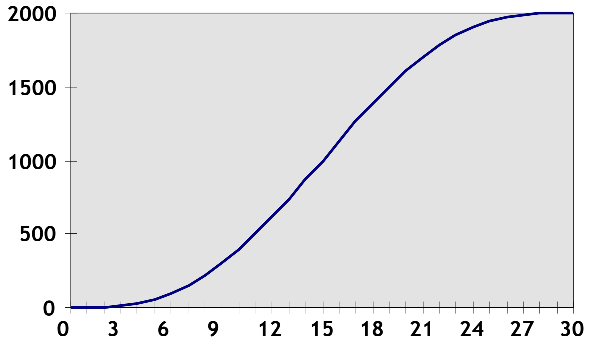
link pos: This parameter is the absolute position where the **CAMBOX** link is to be started when parameter 6 is set to 2.

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

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

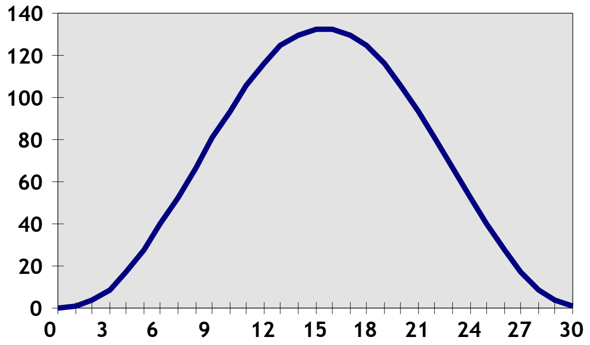
Example 1:

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

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



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

Speed Curve



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

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

Note:

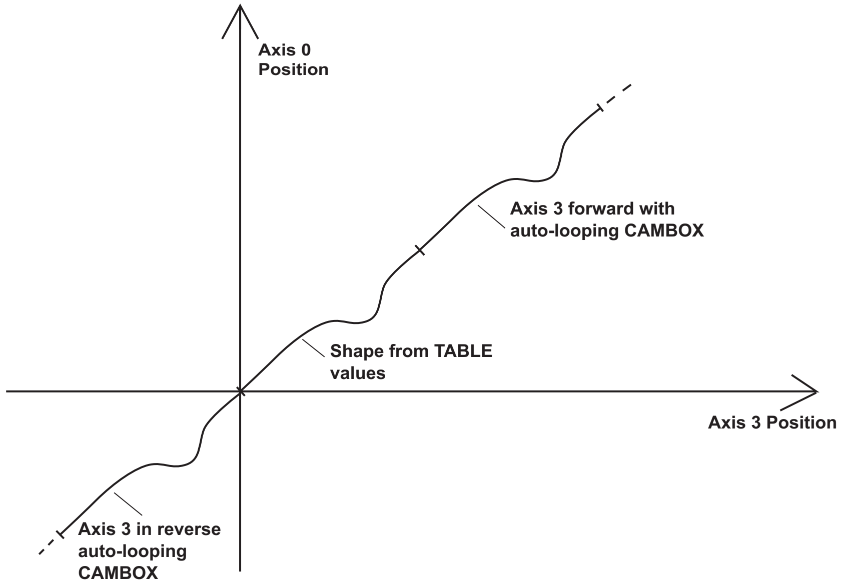
0	The start of the profile shape in the **`TABLE`**
30	The end of the profile shape in the **`TABLE`**

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

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

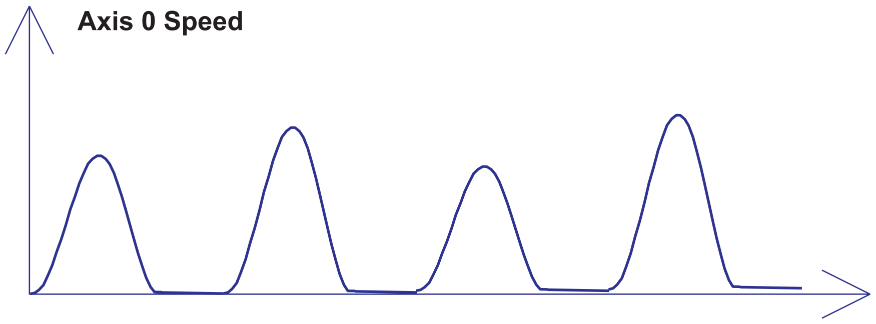
```
CAMBOX(1000,1100,45.68,360,3,4) AXIS(0)
REP_OPTION=2'    cancel repeating mode.
```

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



Example 4: CAMBOX Pattern Mode

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



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

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

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

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

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

CONTROL BLOCK	This is pointed to by the third **CAMBOX** parameter in this options mode only. It is of fixed length (7 table values). It is important to note that the control block is modified during the **CAMBOX** operation. It must therefore be re-initialised prior to each use.
PATTERN BLOCK	The start and end of this are pointed to by 2 of the CONTROL BLOCK values. The pattern sequence is a sequence of scale factors for the SHAPE.

Control Block Parameters

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

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

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

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

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

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

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

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

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

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

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

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

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

Note: Negative motion on link axis:

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

CANCEL

Type: Motion Command

Alternate Format: `CA`

Syntax: `CANCEL / CANCEL(1)`

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

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

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

See also: **RAPIDSTOP.**

Example:

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

Example 2:

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

CONNECT

Type: `Axis Command`

Syntax: `CONNECT(ratio , driving axis)`

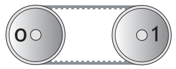
Alternate Format: `CO(ratio, driving axis)`

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

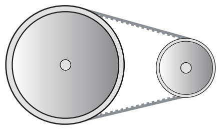
Parameters:

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

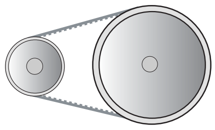
Note: The ratio can be changed at any time by issuing another `CONNECT` command which will automatically update the ratio without the previous `CONNECT` being cancelled. The command can be cancelled with a `CANCEL` or `RAPIDSTOP` command.



`CONNECT(1,1)`



`CONNECT(2,1)`



`CONNECT(0.5,1)`

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

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

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

DATUM

Type: Command

Syntax: `DATUM(sequence no)`

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

Parameter:

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

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

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

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

DEC

Type: Axis Command

Syntax: **`DEC(rate)`**

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

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

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

DEFPOS

Type: Motion Command.

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

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

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

Parameters:

pos1:	Absolute position to set on current base axis in user units.
pos2:	Abs. position to set on the next axis in **BASE** array in user units.
pos3:	Abs. position to set on the next axis in **BASE** array in user units.

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

Example:

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

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

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

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

Example 2:

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

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

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

FORWARD

Type:	Axis Command
Alternate Format:	**FO**
Description:	Sets continuous forward movement.
Note:	The forward motion can only be stopped by issuing a **CANCEL**, **RAPIDSTOP** or by hitting the forward, or datum limits.

Example:

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

MATCH

Type:	Axis Command
Syntax:	**MATCH(count, table address)**
Description:	Instructs the controller to perform a pattern comparison between a stored pattern of registration input transitions and the pattern held following a **REGIST** command.

Parameters:

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

See also: **REGIST** and **RECORD**

Example:

```
dec_dist=SPEED*SPEED*0.5/DECEL
length=10
REP_DIST=100*length
DEFPOS(0)
REGIST(5,length)
MOVE(2*length)
WAIT UNTIL MARK
IF TRANSITIONS>4 AND TRANSITIONS<12 THEN
  MATCH(8,10)
  IF REG_MATCH>0.8 THEN
    IF REG_POS<(MPOS+dec_dist-length) THEN
      MOVEMODIFY(2*length+REG_POS)
    ELSE
      MOVEMODIFY(length+REG_POS)
    ENDIF
  ELSE
    PRINT "marks give too poor fit"
  ENDIF
ELSE
  PRINT "marks not seen"
ENDIF
WAIT IDLE
PRINT REG_POS,ENDMOVE,TRANSITIONS,REG_MATCH
```

MHELICAL

Type: Motion Command.

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

Alternate Format: `MH()`

Description: Performs a helical move.

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

Parameters:

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

direction:	The "direction" is a software switch which determines whether the arc is interpolated in a clockwise or anti- clockwise direction. The parameter is set to 0 or 1. See **MOVECIRC**.
distance3:	The distance to move on the third axis in the **BASE** array axis in user units

MOVE

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

Type: Motion Command

Alternate Format: **MO()**

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

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

Parameters:

distance1:	distance to move on base axis from current position.
distance2:	distance to move on next axis in BASE array from current position.]
[distance3:	distance to move on next axis in BASE array from current position.]
[distance4:	distance to move on next axis in BASE array from current position.]

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

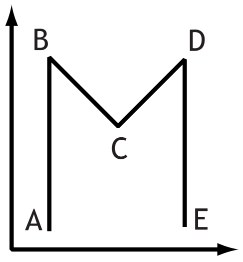
Example 1: A system is working with a unit conversion factor of 1 and has a 1000 line encoder. It is therefore necessary to give the instruction **`MOVE(40000)`** to incrementally move 10 turns on the motor. (A 1000 line encoder gives 4000 edges/turn)

Example 2:

```
MOVE(10) AXIS(5)
MOVE(10) AXIS(4)
MOVE(10) AXIS(3)
```

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

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



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

MOVEABS

Type: Motion Command.

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

Alternate Format: **`MA()`**

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

Parameters:

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

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

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

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

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

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

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

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

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

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

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

MOVECIRC

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

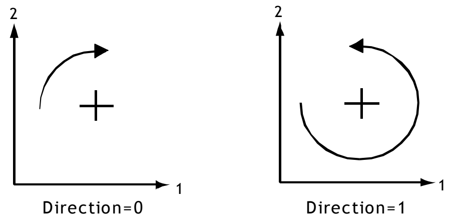
Type: Motion Command.

Alternate Format: `MC()`

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

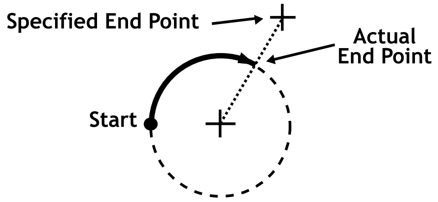
Parameters:

`finish1:`	position on BASE axis to finish at.
`finish2:`	position on next axis in BASE array to finish at.
`centre1:`	position on BASE about which to move.
`centre2:`	position on next axis in **BASE** array about which to move.
`direction:`	The "direction" is a software switch which determines whether the arc is interpolated in a clockwise or anti- clockwise direction.



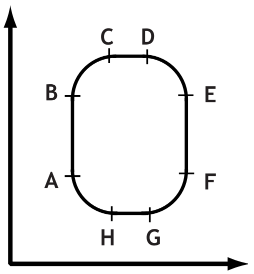
Note: In order for the `MOVECIRC()` command to be correctly executed, the two axes generating the circular arc must have the same number of encoder pulses/linear axis distance. If this is not the case it is possible to adjust the encoder scales in many cases by adjusting with `PP_STEP`.

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



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

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



MOVELINK

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

Type: Motion Command.

Alternate Format: `ML()`

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

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

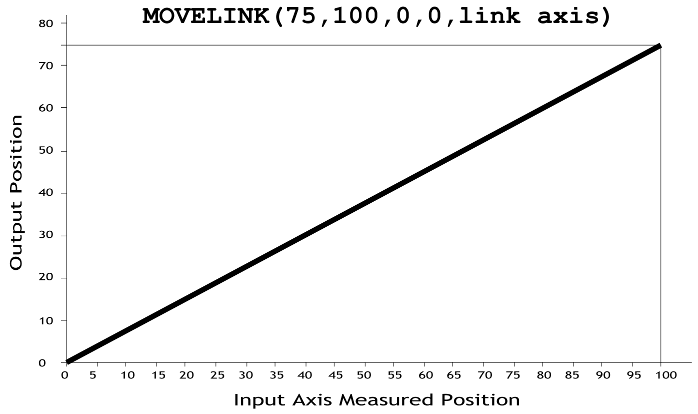
The motion consists of a linear movement with separately variable acceleration and deceleration phases linked via a software gearbox to the MEASURED position (**MPOS**) of another axis.

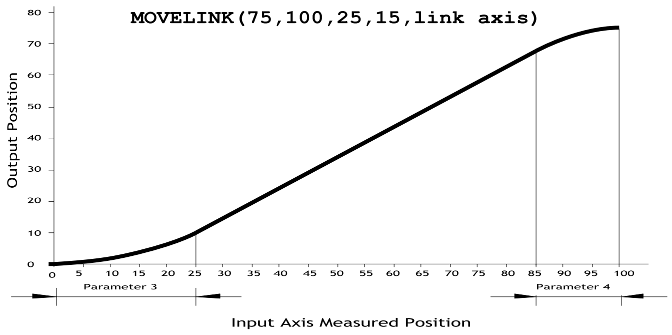
Parameters:

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

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

Note 2: Parameters 6 and 7 are optional.





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

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

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

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

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

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

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

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

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

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

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

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

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

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

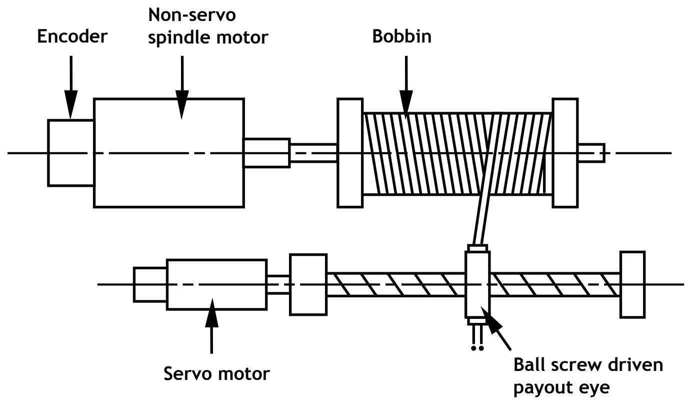
Example 3: **Exact Ratio Gearbox**

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

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

Example 4: Coil Winding

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



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

MOVEMODIFY

Type: Axis Command.

Syntax: **MOVEMODIFY(absolute position)**

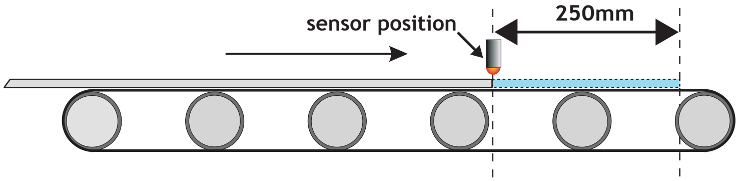
Alternate Format: `MM()`

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

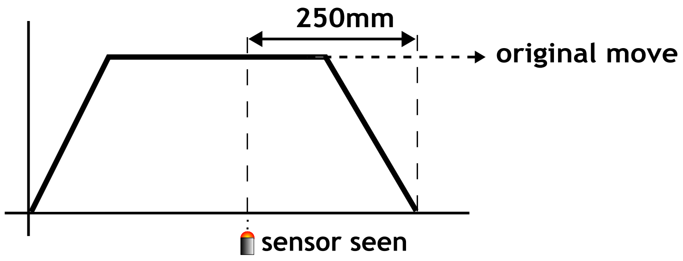
See also: **ENDMOVE**

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

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



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



REGIST

Type: Axis Command

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

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

Parameters:

`mode:` Determines the position to capture:

1 - Absolute position when Z Mark Rising

2 - Absolute position when Z Mark Falling

3 - Absolute position when Registration Input Rising

4 - Absolute position when Registration Input Falling

5 - Sets pattern recognition mode

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

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

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

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

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

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

Note: **Windowing Functions**

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

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

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

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

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

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

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

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

RAPIDSTOP

Type: Motion Command

Alternate Format: **RS**

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

REVERSE

Type: Axis Command

Alternate Format: **RE**

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

Parameters: None.

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

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

Input / Output Commands

AIN

Type: Function

Syntax: `AIN(analog chan)`

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

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

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

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

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

AIN0..7 / AINBI0..7

Type: System Parameter

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

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

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

CURSOR

Type: Command

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

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

4 Line Display as featured on the Membrane Keypad

Example:

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

Example2: The following code

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

would produce the following display:

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

CHR

Type: Command

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

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

DEFKEY

Type: Command

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

```
GET#4,k
```

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

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

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

Parameters:

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

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

FLAG

Type: Command/Function

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

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

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

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

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

FLAGS

Type: Command/Function

Syntax: `FLAGS([value])`

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

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

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

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

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

Example 2: Test if FLAG 3 is set.

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

GET

Type: Command.

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

Example: `GET  k`

GET#

Type: Command

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

Parameters

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

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

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

Example 2: Get a key in a user menu routine

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

HEX

Type: Command

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

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

IN()/IN

Type: Function.

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

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

Parameters:

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

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

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

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

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

Note how the move command is constructed:

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

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

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

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

INPUT

Type: Command.

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

Example1:

```
INPUT num
PRINT "BATCH COUNT=";num[0]
```

On terminal:

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

Example2:

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

This will display on terminal:

```
ENTER LENGTH AND WIDTH ? 1200,1500 <CR>
```

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

INPUTS0 / INPUTS1

Type: System Parameter

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

INVERT_IN

Type: Command.

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

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

Example1:

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

KEY

Type: Function.

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

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

Input device:

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

Example 1:

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

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

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

LINPUT

Type: Command

Syntax: `LINPUT variable`

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

Parameters: None.

Example: `LINPUT VR(0)`

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

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

OP

Type: Command/Function.

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

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

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

Parameters:

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

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

Example 2: `OP (34*256)`

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

Example 3:

```
read_op:
  VR(0)=OP
   'SET OUTPUTS 8..15 ON SIMULTANEOUSLY
  VR(0)=VR(0) AND 65280
  OP(VR(0))
```

Note how this example can also be written:

```
   'SET OUTPUTS 8..15 ON SIMULTANEOUSLY
  OP(OP AND 65280)
```

PRINT

Type: Command.

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

Example 1:

```
PRINT "CAPITALS and lower case CAN BE PRINTED"
```

Suppose VR(1)=6 and variab=1.5: print output will be:

Example 2:

```
>>PRINT 123.45,VR(1)
123.4500 1.5000
>>
```

Note how the comma separator forces the next item to be printed into the next tab column. The width of the field in which a number is printed can be set with the use of [w,x] after the number to be printed. Where w=width of column and x=number of decimal places.

Example 3:

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

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

Example 4:

```
length:
PRINT "DISTANCE=";mpos
DISTANCE=123.0000
```

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

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

Example 5:

```
15 PRINT "ITEM ";total" OF ";limit;CHR(13);
```

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

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

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

PRINT#

Type: Command

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

Parameters:

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

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

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

PSWITCH

Type: Command

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

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

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

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

Parameters:

sw:	The switch number in the range 0..15
en:	Switch enable - 1 or ON to enable software `PSWITCH` 0 or OFF to disable `PSWITCH` 3 to enable hardware `PSWITCH` (hardware PSWITCH can only be used with a P242 daughter board

axis:	Axis number which is to provide the position input in the range 0..number of axes on the controller. For a hardware **`PSWITCH`** it should be set to the axis slot number.
opno:	Selects the physical output to set, should be in range 8..31. For a hardware **`PSWITCH`** it should be set to 0..3.
opst:	Selects the state to set the output to, if 1 then output set **`ON`** else set it **`OFF`**
setpos:	The position at which output is set, in user units
rspos:	The position at which output is reset, in user units

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

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

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

`sw`	The switch number can be any one we chose that is not in use so for the purpose of this example we will use number 0.
`en`	The switch must be enabled to work, therefore this must be set to 1.
`axis`	We are told that the shaft is controlled by axis 0, thus axis is set to 0.
`opno`	We are told that output 11 is the one to fire, so set opno to 11.
`opst`	When the output is set it should be on so set to 1.
`setpos`	The output is to fire at 80° after TDC hence the set position is 80 as we are working in degrees.
`rspos`	The output is to be on for a period of 120° after 80° therefore it goes off at 200°. So the reset position is 200.

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

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

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

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

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

READPACKET

Type: Command

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

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

Parameters:

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

RECORD

Type:	Command	
Syntax:	**`RECORD(count, table address)`**	
Description:	The **`RECORD`** command is part of the pattern recognition system built into the *Motion Coordinator*. Following the recording of a sequence of transitions the **`RECORD`** command is used to: 1 - Reduce the number of transitions to a number defined by the programmer 2 - Store the transition pattern for subsequent comparison with **`MATCH`**	
Parameters:	**`count`**	Number of transitions to record. The actual transitions seen may be greater than this number but the shortest ones are removed so that only the programmed transition count is stored.
	`table address`	This value tells the *Motion Coordinator* where to store the pattern information in the global **`TABLE`** memory. Table used is address to address+24.
Note:	See the **`MATCH`** command for an example of a complete recognition sequence.	

SEND

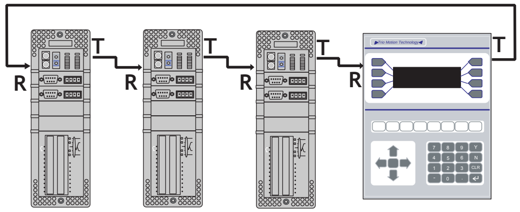
Type:	Command	
Syntax:	**`SEND(n,type,data1[,data2])`**	
Description:	Outputs a fibre-optic network message of a specified type to a given node.	
Parameters:	**`n:`**	Number from 10 to 24 defining the destination node.
	`type:`	Message type: 1 - Direct variable transfer 2 - Keypad offset
	`data1:`	If message type 1, data1 is the numbered variable number, 0..250, on the destination *Motion Coordinator* to modify. If message type 2; data1 is the number of nodes from the keypad that the key characters are to be sent. In the range 10..24, where 10 is the next node and 24 is the fifteenth node away from the keypad.
	`data2:`	Only used if message is type 1. In this case it contains the value to change the specified variable to.

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

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

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

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



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

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

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

SETCOM

Type: Command

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

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

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

Parameters:

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

Example 1:

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

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

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

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

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

Program Loops and Structures

BASICERROR

Type: Program Structure

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

Example:

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

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

ELSE

Type: Program Structure

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

See Also `IF`, `THEN`, `ENDIF`

ELSEIF

Type: Program Structure

Syntax:

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

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

Parameters:

condition(s):	Any logical expression.
commands:	Any valid Trio BASIC commands including further **IF..THEN ..{ELSEIF}..{ELSE} ENDIF** sequences

Example 1:

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

Example 2:

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

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

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

ENDIF

Type: Program Structure

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

Example:

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

See Also **IF, THEN, ELSE**

FOR..TO.. STEP..NEXT

Type: Program Structure

Syntax:

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

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

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

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

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

Parameters:

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

`end:`	A valid Trio BASIC expression.
`increment:`	A valid Trio BASIC expression. (Optional)

Example 1:

```
FOR opnum=10 TO 18
     OP(opnum,ON)
NEXT opnum
```

This loop sets outputs 10 to 18 ON.

Example 2:

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

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

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

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

GOSUB

Type: Program Structure

Syntax: `GOSUB label`

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

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

Example:

```
main:
  GOSUB routine1
  GOSUB routine2
GOTO main

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

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

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

GOTO

Type: Program Structure

Syntax: `GOTO label`

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

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

Example:

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

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

IDLE

Type: Command Modifier

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

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

Example:

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

IF..THEN..ELSE.. ENDIF

Type: Program Structure

Syntax:

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

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

Parameters:

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

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

Example 1:

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

Example 2:

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

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

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

NEXT

Type: Program Structure

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

ON.. GOSUB

Type: Program Structure

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

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

Example:

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

ON.. GOTO

Type: Program Structure

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

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

Example:

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

REPEAT.. UNTIL

Type: Program Structure

Syntax: `REPEAT commands UNTIL condition`

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

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

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

RETURN

Type: Program Structure

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

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

Example:

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

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

STEP

Type: Program Structure

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

Example:

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

STOP

Type: Command

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

Example 1:

```
>>STOP progname
```

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

THEN

Type: Program Structure

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

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

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

TO

Type: Program Structure

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

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

UNTIL

Type: Program Structure

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

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

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

WA

Type: Command

Syntax: `WA(delay time)`

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

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

Example:

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

WAIT IDLE

Type: Command

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

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

Example:

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

WAIT LOADED

Type: Command

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

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

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

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

WAIT UNTIL

Type: Command

Syntax: `WAIT UNTIL` condition

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

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

Example 1:

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

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

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

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

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

WHILE

Type: Program Structure

Syntax: **`WHILE condition`**

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

Execution then continues after the **`WEND`**.

Parameters:

`condition:`	Any valid logical Trio BASIC expression

Example:

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

WEND

Type: Program Structure

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

See also: **`WHILE`**

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

System Parameters and Commands

ADDRESS

Type: System Parameter

Syntax: `ADDRESS=value`

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

Example: `ADDRESS=5`

APPENDPROG

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

Syntax: `APPENDPROG <string>`

Alternate Format: `@ <string>`

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

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

AUTORUN

Type: System Command

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

See Also: `RUNTYPE`.

AXISVALUES

Type: System Command

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

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

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

`<Parameter><type>=<value>`

`<Parameter>`	is the name of the parameter
`<type>`	is the type of the value.
	`i` integer
	`f` float
	`c` float that when changed means that the bank 0 data must be updated
	`s` string
	`c` string of upper and lower case letters, where upper case letters mean an error
`<value>`	an integer, a float or a string depending on the type

CAN

Type: System Function

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

CANIO_ADDRESS

Type: System Parameter (Stored in FLASH Eprom)

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

CANIO_ENABLE

Type: System Parameter

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

CANIO_STATUS

Type: System Parameter

Description: A bitwise system parameter:

`Bit 0`	set indicates an error from the I/O module 0,3,6 or 9
`Bit 1`	set indicates an error from the I/O module 1,4,7 or 10
`Bit 2`	set indicates an error from the I/O module 2,5,8 or 11
`Bit 3`	set indicates an error from the I/O module 12,13,14 or 15
`Bit 4`	should be set to re-initialise the CANIO network
`Bit 5`	is set when initialisation is complete

CHECKSUM

Type: System Parameter (Read Only)

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

CLEAR

Type: System Command

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

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

CLEAR does not alter the program in memory.

Example:

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

On execution this would give an output such as:

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

CLEAR_PARAMS

Type: System Command

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

COMMSERROR

Type: System Parameter

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

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

COMMSTYPE

Type: Slot Parameter

Syntax: `COMMSTYPE SLOT(slot#)`

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

COMPILE

Type: System Command

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

CONTROL

Type: System Parameter (Read Only)

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

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

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

COPY

Type: System Command

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

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

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

DATE

Type: System Parameter (MC216/MC224 Only)

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

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

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

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

DATE$

Type: Command (MC216/MC224 Only)

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

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

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

DAY

Type: System Parameter (MC216/MC224 only)

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

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

DAY$

Type: System Command (MC216/MC224 only)

Description: Prints the current day as a string.

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

DEL

Type: System Command

Alternate Format: **RM**

Syntax: **DEL progname**

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

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

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

DEVICENET

Type:	System Command
Syntax:	**`DEVICENET(slot,func,baud,mac id,poll base,poll inlen,poll outlen)`**
Description:	The command DEVICENET is used to start and stop the DeviceNet slave function which is built into the *Motion Coordinator*. Once the DEVICENET command is running the process can be used to execute other BASIC commands in the usual way. This command is available on MC206 and MC224 only.

Parameters:

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

Example 1:

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

Example 2:

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

DIR

Type:	System Command
Alternate Format:	**`LS`**
Description:	Prints a list of all programs in memory, their size and their **`RUNTYPE`**. The alternative format **`DIR F`** may be used to list the programs stored in the FlashStick if present.
Note:	This command should only be used on the *Motion Coordinator* Command Line

DISPLAY

Type: System Parameter

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

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

The parameter default value is 0.

Parameters:

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

Example:

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

DLINK

Type: System Command

Syntax: **`DLINK(function,…)`**

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

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

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

Read a register on the SLM™ ASIC.

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

Example:

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

Write a register on the SLM™ ASIC.

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

Example:

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

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

Parameters:

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

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

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

Parameters:

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

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

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

Parameters:

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

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

Read an SLM parameter

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

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

Write an SLM parameter

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

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

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

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

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

Read a drive parameter

Parameters:

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

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

Return slot and asic number associated with an axis

Parameters:

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

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

Read an EEPROM parameter

Parameters:

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

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

EDPROG

Type: System Command

Alternate Format: &

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

It has several forms:

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

EDIT

Type: System Command

Syntax: `EDIT [optional line sequence number]`

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

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

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

EPROM

Type: System Command

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

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

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

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

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

ERROR_AXIS

Type: System Parameter (Read Only)

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

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

ETHERNET

Type: System Command

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

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

Parameters:

`read / write:`	Specifies the required action. 0 = Read 1 = Write to Flash EPROM 2 = Write to RAM
`slot number:`	The daughter board slot where the Ethernet port has been installed. On the MC206 this is always slot 0.

`function:`	Function number must be one of the following values. 0 = IP Address 1 = Static(1) or dynamic(0) addressing. (Only static addressing is supported.) 2 = Subnet Mask 3 = MAC address 4 = Default Port Number (initialised to 23) 5 = Token Port Number (initialised to 3240) 6 = Ethernet daughter board firmware version (read only) 7 = Modbus TCP mode. Integer (0) or Floating point (1). (daughter board firmware V1.0.3.1 or later)
`data:`	The optional data is used when changing a parameter value. When writing to the EPROM on the Ethernet daughter board, the new value will only be used after power has been cycled to the controller. Any data written to RAM is used straight away.

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

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

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

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

EX

Type: System Command

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

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

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

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

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

EXECUTE

Type: System Command

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

FEATURE_ENABLE

Type: System Function

Syntax: **`FEATURE_ENABLE(feature number)`**

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

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

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

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

Example 1:

```
>>feature_enable
Security code=17980000000028
Enabled features: 0 1
```

If you require additional features for a controller. These can be enabled by the entry of a password which is unique for each feature and controller security code. To obtain a feature enable code, the feature must be ordered and the security code FAXed to Trio or a distributor.

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

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

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

FLASHVR

Type: System Function

Syntax: `FLASHVR([variable number])`

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

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

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

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

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

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

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

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

FRAME

Type: System Parameter

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

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

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

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

Example:

```
FRAME=1
```

FREE

Type: System Parameter (Read Only)

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

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

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

Parameters: None

Example 1:

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

Example 2:

```
VR(10)=IN AND 255
```

This line requires 21 bytes of storage in the uncompiled version and 19 in the compiled version:

Uncompiled:

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

Compiled version:

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

HALT

Type: System Command

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

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

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

INITIALISE

Type: System Command.

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

LAST_AXIS

Type: System Parameter (Read Only)

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

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

LIST

Type: System Command

Alternate Format: `TYPE`

Description: Prints the current `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.*

LOADSYSTEM

Type: System Command

Description: Loads new version of system software:

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

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

The download sequence:

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

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

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

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

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

LOCK

Type: System Command

Syntax: `LOCK(code)`

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

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

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

The lock code number may be any integer and is held in encoded form. Once `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.***

MOTION_ERROR

Type: System Parameter

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

MPE

Type: System Command

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

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

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

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

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

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

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

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

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

NAIO

Type: System Parameter (Read Only)

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

NETSTAT

Type: System Parameter

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

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

NEW

Type: System Command

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

Note:

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

NIO

Type: System Parameter (Read Only)

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

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

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

PEEK

Type: System Command

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

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

POKE

Type: System Command

Syntax: `POKE(address,value)`

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

POWER_UP

Type: Flash EPROM stored System Parameter

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

Two values are possible:

0	Use the programs in battery backed RAM
1	Copy programs from the controllers Flash Eprom or Memory Stick (if present) into RAM. The EPROM request flag held on the Memory Stick itself controls if the programs loaded are to be written into the internal Flash Eprom after loading (See **EPROM**)

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

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

This value is normally set by *Motion* Perfect.

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

PROCESS

Type: System Command

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

PROFIBUS

Type: System Command

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

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

Parameters:

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

REMOTE

Type: System Command

Syntax: **`REMOTE(slot)`**

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

RENAME

Type: System Command

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

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

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

RUN

Type: System Command

Description: Runs a program on the controller.

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

Note: Execution continues until:

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

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

```
RUN "cycle"
```

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

Example 2: **`RUN “sausage”`** - this will run the named program)

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

RUNTYPE

Type:	System Command
Syntax:	`>>RUNTYPE  progname,autorun[,process#]`
Description:	Sets whether program is run automatically at power up, and which process it is to run on. The current status of each program's **RUNTYPE** is displayed when a **DIR** command is performed. For any program to run automatically on power-up ALL the programs on the controller must compile without errors.
Parameters:	

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

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

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

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

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

See Also: **POWER_UP**

SCOPE

Type: System Command

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

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

Parameters:

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

Example 1:

```
SCOPE(ON,10,0,1000,MPOS AXIS(5), DPOS AXIS(5))
```

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

Example 2:

```
SCOPE(OFF)
```

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

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

SCOPE_POS

Type: System Parameter (Read Only)

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

SELECT

Type: System Command

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

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

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

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

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

SLOT

Type: Slot Modifier

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

SERVO_PERIOD

Type: System Parameter

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

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

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

STEPLINE

Type: System Command

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

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

Parameters:

`Program name:`	This specifies the program to be stepped. All copies of this named program will step unless the process number is also specified. If the program is not running it will step to the first executable line on either the specified process or the next available process if the next parameter is omitted. If the program name is not supplied, either the SELECTed program will step (if command line entry) or the program with the STEPLINE in it will stop running and begin stepping.
`Process number:`	This optional parameter determines which process number the program will use for stepping, or, if multiple copies of the same program exist, it is used to select the required copy for stepping.

Example 1: `>>STEPLINE "conveyor"`

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

STICK_READ

Type: System Command

Syntax: `STICK_READ(sector, table start)`

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

Parameters:

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

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

STICK_WRITE

Type: System Command

Syntax: `STICK_WRITE(sector, table start)`

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

Parameters:

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

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

STORE

Type: System Command

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

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

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

TABLE

Type: System Command

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

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

The command has two forms:

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

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

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

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

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

Parameters:

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

Example 1: **TABLE(100,0,120,250,370,470,530)**

This loads the internal table:

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

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

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

TABLEVALUES

Type: System Command

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

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

Parameters

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

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

TIME

Type: System Parameter (MC216/MC224 only)

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

TIME$

Type: System Command (MC216/MC224 only)

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

Example:

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

TRIGGER

Type: System Command

Description: Starts a previously set up `SCOPE` command

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

TROFF

Type: System Command

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

Example:

```
>>TROFF "lines"
```

TRON

Type: System Command

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

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

Example:

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

TSIZE

Type: System Parameter

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

Example:

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

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

UNLOCK

Type: System Command

Syntax: **`UNLOCK(code)`**

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

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

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

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

Parameters:	**`code`**	Any integer number

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

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

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

USB

Type: System Command

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

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

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

`register:`	The register number to read or write
`value:`	The value to write into a register

USB_STALL

Type: System Parameter

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

VERSION

Type: System Parameter

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

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

VIEW

Type: System Command

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

Example: For the following program:

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

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

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

VR

Type: System Command

Syntax: `VR(expression)`

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

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

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

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

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

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

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

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

```
movep:
  MOVEABS(115) ' MOVE TO FIRST PUT DOWN POSITION:
  FOR VR(0)=101 TO 110
    IF VR(VR(0))=0) THEN GOSUB load
    MOVE(200)‘ 200 IS SPACING BETWEEN POSITIONS
  NEXT VR(0)
  PRINT "All Positions Are Full"
  WAIT UNTIL IN(3)=ON
GOTO movep

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

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

Example 3:

```
‘Assign VR(65) to VR(0) multiplied by Axis 1 measured position
VR(65)=VR(0)*MPOS AXIS(1)
PRINT VR(65)
```

WDOG

Type: System Parameter

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

Example: `WDOG=ON`

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

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

:

Type: Special Character

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

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

Example:
```
start:
```

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

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

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

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

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

'

Type: Special Character

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

Note: The REM command of other BASICs is replaced by '.

Like REM statements ' must be at the beginning of the line or statement or after the executable statement. Comments use memory space and so should be concise in very long programs. Comments have no effect on execution speed since they are not present in the compiled code.

Example:
```
'PROGRAM TO ROTATE WHEEL
turns=10
'turns contains the number of turns required
MOVE(turns)' the movement occurs here
```

#

Type: Special Character

Description: The # symbol is used to specify a communications channel to be used for serial input/output commands.

Note: Communications Channels greater than 3 will only be used when the controller is running in *Motion* Perfect mode (See **MPE** command).

Example 1:
```
PRINT #3,"Membrane Keypad"
PRINT #2,"Port 2"
```

Example 2:
```
' Check membrane keypad on fibre-optic channel
IF KEY #3 THEN GET #3,k
```

$

Type: Special Character

Description: The $ symbol is used to specify that the number that follows is in hexadecimal format.

Note: Only the MC206,EURO205X, PCI208 and MC224 controllers support hexadecimal format.

Example 1:
```
VR(10)=$8F3B
OP($CC00)
```

Process Parameters and Commands

ERROR_LINE

Type: Process Parameter (Read Only)

Description: Stores the number of the line which caused the last Trio BASIC error. This value is only valid when the **BASICERROR** is **TRUE**. This parameter is held independently for each process.

Example:
```
>>PRINT ERROR_LINE PROC(14)
```

INDEVICE

Type: Process Parameter

Description: This parameter specifies the active input device. Specifying an **INDEVICE** for a process allows the channel number for a program to set for all subsequent **GET** and **KEY**, **INPUT** and **LINPUT** statements. (This command is not usually required - Use **GET #** and **KEY #** etc. instead)

Chan	Input device:-
0	Serial port A
1	Serial port B
2	RS485 Port
3	Fibre optic port (value returned defined by DEFKEY)
4	Fibre optic port (returns raw keycode of key pressed)
5	*Motion* Perfect user channel
6	*Motion* Perfect user channel
7	*Motion* Perfect user channel
8	Used for *Motion* Perfect internal operations
9	Used for *Motion* Perfect internal operations
10	Fibre optic network data

Example:
```
INDEVICE=5
  ' Get character on channel 5:
GET k
```

LOOKUP

Type: Process Command

Syntax: `LOOKUP(format,entry) <PROC(process#)>`

Description: The LOOKUP command allows *Motion* Perfect to access the local variables on an executing process. It is not normally required for BASIC programs.

Parameters:

`format:`	0: Prints (in binary) floating point value from an expression 1: Prints (in binary) integer value from an expression 2: Prints (in binary) local variable from a process 3: Returns to BASIC local variable from a process
`entry:`	Either an expression string (format=0 or 1) or the offset number of the local variable into the processes local variable list.

OUTDEVICE

Type: Process Parameter

Description: The value in this parameter determines the serial output device for the **`PRINT`** command for the process. The channel numbers are the same as described in **`INDEVICE`**.

PMOVE

Type: Process Parameter

Modifier: **`PROC`**

Description: Returns 1 if the process move buffer is occupied, and 0 it is empty. When one of the *Motion Coordinator* processes encounters a movement command the process loads the movement requirements into its “process move buffer”. This can hold one movement instruction for any group of axes. When the load into the process move buffer is complete the **`PMOVE`** parameter is set to 1. When the next servo interrupt occurs the motion generation program will load the movement into the “next move buffer” of the required axes if these are available. When this second transfer is complete the **`PMOVE`** parameter is cleared to 0. Each process has its own **`PMOVE`** parameter.

PROC

Type: Process Modifier

Description: Allows a process parameter from a particular process to be read or set.

Example: `WAIT UNTIL PMOVE PROC(14)=0`

PROC_LINE

Type: Process Parameter (Read Only)

Description: Allows the current line number of another program to be obtained with the `PROC(x)` modifier.

Example: `PRINT PROC_LINE PROC(2)`

PROCNUMBER

Type: Process Parameter

Description: Returns the process on which a Trio BASIC program is running. This is normally required when multiple copies of a program are running on different processes.

Example: `MOVE(length) AXIS(PROCNUMBER)`

PROC_STATUS

Type: Process Parameter (Read Only)

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

Example: `WAIT UNTIL PROC_STATUS PROC(12)=0`

Returns

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

RESET

Type: Process Command

Description: Sets the value of all the local named variables of a Trio BASIC process to 0.

RUN_ERROR

Type: Process Parameter

Modifier: `PROC`

Description: Contains the number of the last program error that occurred on the specified process.

Example:

```
>>? RUN_ERROR PROC(5)
  9.0000
>>
```

TICKS

Type: Process Parameter

Description: The current count of the process clock ticks is stored in this parameter. The process parameter is a 32 bit counter which is DECREMENTED on each servo cycle. It can therefore be used to measure cycle times, add time delays, etc. The ticks parameter can be written to and read.

Example:

```
delay:
  TICKS=3000
  OP(9,ON)
test:
  IF TICKS<=0 THEN OP(9,OFF) ELSE GOTO test
```

Note: `TICKS` is held independently for each process.

Mathematical Operations and Commands

+ Add

Type: Arithmetic operation

Syntax `<expression1> + <expression2>`

Description: Adds two expressions

Parameters:

`Expression1:`	Any valid Trio BASIC expression
`Expression2:`	Any valid Trio BASIC expression

Example: `result=10+(2.1*9)`

Trio BASIC evaluates the parentheses first giving the value 18.9 and then adds the two expressions. Therefore result holds the value 28.9

- Subtract

Type: Arithmetic operation

Syntax `<expression1> - <expression2>`

Description: Subtracts expression2 from expression1

Parameters:

`Expression1:`	Any valid Trio BASIC expression
`Expression2:`	Any valid Trio BASIC expression

Example: `VR(0)=10-(2.1*9)`

Trio BASIC evaluates the parentheses first giving the value 18.9 and then subtracts this from 10. Therefore VR(0) holds the value -8.9

* Multiply

Type: Arithmetic operation

Syntax `<expression1> * <expression2>`

Description: Multiplies expression1 by expression2

Parameters:

`Expression1:`	Any valid Trio BASIC expression
`Expression2:`	Any valid Trio BASIC expression

Example:

```
factor=10*(2.1+9)
```

Trio BASIC evaluates the brackets first giving the value 11.1 and then multiplies this by 10. Therefore factor holds the value 111

/ Divide

Type: Arithmetic operation

Syntax `<expression1> / <expression2>`

Description: Divides expression1 by expression2

Parameters:

`Expression1:`	Any valid Trio BASIC expression
`Expression2:`	Any valid Trio BASIC expression

Example:

```
a=10/(2.1+9)
```

Trio BASIC evaluates the parentheses first giving the value 11.1 and then divides 10 by this number

Therefore a holds the value 0.9009

= Equals

Type: Arithmetic Comparison Operation

Syntax `<expression1> = <expression2>`

Description: Returns **`TRUE`** if expression1 is equal to expression2, otherwise returns false.

Note: **`TRUE`** is defined as -1, and **`FALSE`** as 0

Parameters:

`Expression1:`	Any valid Trio BASIC expression
`Expression2:`	Any valid Trio BASIC expression

Example:

```
IF IN(7)=ON THEN GOTO label
```

If input 7 is ON then program execution will continue at line starting “`label:`”

<> Not Equal

Type: Arithmetic Comparison Operation

Syntax `<expression1> <> <expression2>`

Description: Returns **`TRUE`** if expression1 is not equal to expression2, otherwise returns false.

Note: **`TRUE`** is defined as -1, and **`FALSE`** as 0

Parameters:

`Expression1:`	Any valid Trio BASIC expression
`Expression2:`	Any valid Trio BASIC expression

Example: IF MTYPE<>0 THEN GOTO scoop
If axis is not idle (MTYPE=0 indicates axis idle) then goto label “scoop”

> Greater Than

Type: Arithmetic Comparison Operation

Syntax **`<expression1> > <expression2>`**

Description: Returns **`TRUE`** if expression1 is greater than expression2, otherwise returns false.

Note: **`TRUE`** is defined as -1, and **`FALSE`** as 0

Parameters:

`Expression1:`	Any valid Trio BASIC expression
`Expression2:`	Any valid Trio BASIC expression

Example 1: **`WAIT UNTIL MPOS>200`**
The program will wait until the measured position is greater than 200

Example 2: **`VR(0)=1>0`**
1 is greater than 0 and therefore VR(0) holds the value -1

>= Greater Than or Equal

Type: Arithmetic Comparison Operation

Syntax **`<expression1> >= <expression2>`**

Description: Returns **`TRUE`** if expression1 is greater than or equal to expression2, otherwise returns false.

Note: **`TRUE`** is defined as -1, and **`FALSE`** as 0

Parameters:

`Expression1:`	Any valid Trio BASIC expression
`Expression2:`	Any valid Trio BASIC expression

Example: **`IF target>=120 THEN MOVEABS(0)`**
If variable target holds a value greater than or equal to 120 then move to the absolute position of 0.

< Less Than

Type:	Arithmetic Comparison Operation
Syntax	**`<expression1> < <expression2>`**
Description:	Returns **`TRUE`** if expression1 is less than expression2, otherwise returns false.
Note:	**`TRUE`** is defined as -1, and **`FALSE`** as 0
Parameters:	**`Expression1:`** Any valid Trio BASIC expression
	`Expression2:` Any valid Trio BASIC expression
Example:	**`IF AIN(1)<10 THEN GOSUB rollup`** If the value returned from analog input 1 is less than 10 then execute subroutine "**`rollup`**"

<= Less Than or Equal

Type:	Arithmetic Comparison Operation
Syntax	**`<expression1> <= <expression2>`**
Description:	Returns **`TRUE`** if expression1 is less than or equal to expression2, otherwise returns false.
Note:	**`TRUE`** is defined as -1, and **`FALSE`** as 0
Parameters:	**`Expression1:`** Any valid Trio BASIC expression
	`Expression2:` Any valid Trio BASIC expression
Example:	**`maybe=1<=0`** 1 is not less than or equal to 0 and therefore variable **`maybe`** holds the value 0

ABS

Type: Function

Syntax: **`ABS(expression)`**

Description: The **`ABS`** function converts a negative number into its positive equal. Positive numbers are unaltered.

Parameters: **`Expression:`** Any valid Trio BASIC expression

Example:
```
IF ABS(AIN(0))>100 THEN
  PRINT "Analog Input Outside +/-100"
ENDIF
```

ACOS

Type: Function

Syntax: **`ACOS(expression)`**

Description: The **`ACOS`** function returns the arc-cosine of a number which should be in the range 1 to -1. The result in radians is in the range 0..PI

Parameters: **`Expression:`** Any valid Trio BASIC expression.

Example:
```
>>PRINT ACOS(-1)
3.1416
```

AND

Type: Logical and bitwise operator

Syntax **`<expression1> AND <expression2>`**

Description: This performs an AND function between corresponding bits of the integer part of two valid Trio BASIC expressions.

The AND function between two bits is defined as follows:

Parameters: `Expression1:` Any valid Trio BASIC expression

`Expression2:` Any valid Trio BASIC expression

Example 1: `IF (IN(6)=ON) AND (DPOS>100) THEN tap=ON`

Example 2: `VR(0)=10 AND (2.1*9)`

Trio BASIC evaluates the parentheses first giving the value 18.9, but as was specified earlier, only the integer part of the number is used for the operation, therefore this expression is equivalent to:

`VR(0)=10 AND 18`

AND is a bitwise operator and so the binary action taking place is:

	0	1
0	0	0
1	0	1

```
     01010
AND  10010
     -----
     00010
```

Therefore `VR(0)` holds the value 2

Example 3: `IF MPOS AXIS(0)>0 AND MPOS AXIS(1)>0 THEN GOTO cycl`

ASIN

Type: Mathematical Function

Syntax: `ASIN(expression)`

Alternate Format: `ASN(expression)`

Description: The **ASIN** function returns the arc-sine of a number which should be in the range +/-1. The result in radians is in the range -PI/2.. +PI/2 (Numbers outside the +/-1 input range will return zero)

Parameters: `Expression:` Any valid Trio BASIC expression.

Example:

```
>>PRINT ASIN(-1)
-1.5708
```

ATAN

Type: Mathematical Function

Syntax: `ATAN(expression)`

Alternate Format: `ATN(expression)`

Description: The **ATAN** function returns the arc-tangent of a number. The result in radians is in the range -PI/2.. +PI/2

Parameters: `Expression:` Any valid Trio BASIC expression.

Example:

```
>>PRINT ATAN(1)
0.7854
```

ATAN2

Type: Mathematical Function

Syntax: `ATAN2(expression1,expression 2)`

Description: The **ATAN2** function returns the arc-tangent of the ratio expression1/expression 2. The result in radians is in the range -PI.. +PI

Parameters: `Expressions:` Any valid Trio BASIC expression.

Example:

```
>>PRINT ATAN2(0,1)
0.0000
```

CLEAR_BIT

Type:	Command
Syntax:	`CLEAR_BIT(bit#,vr#)`
Description:	`CLEAR_BIT` can be used to clear the value of a single bit within a `VR()` variable.
Example:	`CLEAR_BIT(6,23)` Bit 6 of VR(23) will be cleared (set to 0).

Parameters:

`bit #`	Bit number within the VR. Valid range is 0 to 23
`vr#`	VR() number to use

See also `READ_BIT`, `SET_BIT`

CONSTANT

Type:	System Command
Syntax:	`CONSTANT "name", value`
Description:	Declares the *name* as a constant for use both within the program containing the `CONSTANT` definition and all other programs in the Motion Coordinator project. MC206, EURO205X, PCI208 and MC224 only.

Parameters:

`name:`	Any user-defined name containing lower case alpha, numerical or underscore (_) characters.
`value`	The value assigned to *name*.

Example:

```
CONSTANT "nak",$15
CONSTANT "start_button",5

IF IN(start_button)=ON THEN OP(led1,ON)
IF key_char=nak THEN GOSUB no_ack_received
```

Note: The program containing the `CONSTANT` definition must be run before the name is used in other programs. For fast startup the program should also be the ONLY process running at power-up.

A maximum of 128 `CONSTANTs` can be declared.

COS

Type: Mathematical Function

Syntax: `COS(expression)`

Description: Returns the `COSINE` of an expression. Will work for any value. Input values are in radians.

Parameters:

`Expression:`	Any valid Trio BASIC expression.

Example:

```
>>PRINT COS(0)[3]
1.000
>>
```

EXP

Type: Mathematical Function

Syntax: `EXP(expression)`

Description: Returns the exponential value of the expression.

FRAC

Type: Mathematical Function

Syntax: `FRAC(expression)`

Description: Returns the fractional part of the expression.

Example:

```
>>PRINT FRAC(1.234)
0.2340
```

GLOBAL

Type: System Command

Syntax: `GLOBAL "name", vr_number`

Description: Declares the *name* as a reference to one of the global VR variables. The name can then be used both within the program containing the **GLOBAL** definition and all other programs in the Motion Coordinator project. MC206, EURO205X, PCI208 and MC224 only.

Parameters:

name:	Any user-defined name containing lower case alpha, numerical or underscore (_) characters.
vr_number	The number of the VR to be associated with *name*.

Example:

```
GLOBAL "srew_pitch",12
GLOBAL "ratio1",534

ratio1 = 3.56
screw_pitch = 23.0
PRINT screw_pitch, ratio1
```

Note: The program containing the **GLOBAL** definition must be run before the name is used in other programs. For fast startup the program should also be the ONLY process running at power-up.

In programs that use the defined **GLOBAL**, **name** has the same meaning as **VR(vr_number).** Do not use the syntax: VR(name).

A maximum of 128 **GLOBALs** can be declared.

IEEE_IN

Type: Mathematical Function

Syntax: `IEEE_IN(byte0,byte1,byte2,byte3)`

Description: The **IEEE_IN** function returns the floating point number represented by 4 bytes which typically have been received over a communications link such as Modbus. MC206, EURO205X, PCI208 and MC224 only.

Parameters: **`byte0 - 3:`** Any combination of 8 bit values that represents a valid IEEE floating point number.

Example:

```
VR(20) = IEEE_IN(b0,b1,b2,b3)
```

Note: Byte 0 is the high byte of the 32 bit floating point format.

IEEE_OUT

Type: Mathematical Function

Syntax: **`byte_n = IEEE_OUT(value, n)`**

Description: The **`IEEE_OUT`** function returns a single byte in IEEE format extracted from the floating point value for transmission over a bus sytem. The function will typically be called 4 times to extract each byte in turn. MC206, EURO205X, PCI208 and MC224 only.

Parameters:

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

Example:

```
byte0 = IEEE_OUT(MPOS AXIS(2), 0)
byte1 = IEEE_OUT(MPOS AXIS(2), 1)
byte2 = IEEE_OUT(MPOS AXIS(2), 2)
byte3 = IEEE_OUT(MPOS AXIS(2), 3)
```

Note: Byte 0 is the high byte of the 32 bit IEEE floating point format.

INT

Type: Mathematical Function

Syntax: **`INT(expression)`**

Description: The **`INT`** function returns the integer part of a number.

Parameters: **`expression:`** Any valid Trio BASIC expression.

Example:

```
>>PRINT INT(1.79)
1.0000
>>
```

Note: To round a positive number to the nearest integer value take the **INT** function of the (number + 0.5)

LN

Type: Mathematical Function

Syntax: `LN(expression)`

Description: Returns the natural logarithm of the expression.

Parameter: Any valid Trio BASIC expression.

MOD

Type: Mathematical Function

Syntax: `MOD(expression)`

Description: Returns the integer modulus of an expression.

Example:

```
>>PRINT 122 MOD(13)
5.0000
>>
```

NOT

Type: Mathematical Function

Description: The **NOT** function truncates the number and inverts all the bits of the integer remaining.

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

Example:
```
PRINT 7 AND NOT(1.5)
  6.0000
>>
```

OR

Type: Logical and bitwise operator

Description: This performs an **OR** function between corresponding bits of the integer part of two valid Trio BASIC expressions. The **OR** function between two bits is defined as follows:

OR	0	1
0	0	1
1	1	1

Parameters:

Expression1:	Any valid Trio BASIC expression
Expression2:	Any valid Trio BASIC expression

Example 1:
```
IF KEY OR IN(0)=ON THEN GOTO label
```

Example 2:
```
result=10 OR (2.1*9)
```

Trio BASIC evaluates the parentheses first giving the value 18.9, but as was specified earlier, only the integer part of the number is used for the operation, therefore this expression is equivalent to:

```
result=10 OR 18
```
The OR is a bitwise operator and so the binary action taking place is:

```
    01010
OR  10010
---------
    11010
```

Therefore result holds the value 26

READ_BIT

Type: Command

Syntax: `READ_BIT(bit#,vr#)`

Description: `READ_BIT` can be used to test the value of a single bit within a `VR()` variable.

Example: `res=READ_BIT(4,13)`

Parameters:

`bit #`	Bit number within the VR. Valid range is 0 to 23
`vr#`	VR() number to use

See also `SET_BIT`, `CLEAR_BIT`

SET_BIT

Type: Command

Syntax: `SET_BIT(bit#,vr#)`

Description: `SET_BIT` can be used to set the value of a single bit within a `VR()` variable. All other bits are unchanged.

Parameters:

`bit #`	Bit number within the VR. Valid range is 0 to 23
`vr#`	`VR()` number to use

Example:

```
SET_BIT(3,7)
Will set bit 3 of VR(7) to 1.
```

See also `READ_BIT`, `CLEAR_BIT`

SGN

Type: Mathematical Function

Syntax: `SGN(expression)`

Description: The `SGN` function returns the SIGN of a number.

`1`	Positive non-zero
`0`	Zero
`-1`	Negative

Parameters: `expression:` Any valid Trio BASIC expression.

Example:
```
>>PRINT SGN(-1.2)
-1.0000
>>
```

SIN

Type: Mathematical Function

Syntax: `SIN(expression)`

Description: Returns the SINE of an expression. This is valid for any value in expressed in radians.

Parameters: `expression:` Any valid Trio BASIC expression.

Example:
```
>>PRINT SIN(0)
   0.0000
```

SQR

Type: Mathematical Function

Syntax: `SQR(number)`

Description: Returns the square root of a number.

Parameters: `number:` Any valid Trio BASIC number or variable.

Example:
```
>>PRINT SQR(4)
2.0000
```

```
>>
```

TAN

Type: Mathematical Function

Syntax: `TAN(expression)`

Description: Returns the TANGENT of an expression. This is valid for any value expressed in radians.

Parameters: `Expression:` Any valid Trio BASIC expression.

Example:

```
>>PRINT TAN(0.5)
   0.5463
```

XOR

Type: Logical and bitwise operator

Description: This performs and XOR function between corresponding bits of the integer part of two valid Trio BASIC expressions. It may therefore be used as either a bitwise or logical condition.

The `XOR` function between two bits is defined as follows:

Parameters: `Expression1:` Any valid Trio BASIC expression

`Expression2:` Any valid Trio BASIC expression

Example: `a = 10 XOR (2.1*9)`

Trio BASIC evaluates the parentheses first giving the value 18.9, but as was specified earlier, only the integer part of the number is used for the operation, therefore this expression is equivalent to: `a=10 XOR 18`. The `XOR` is a bitwise operator and so the binary action taking place is:

```
      01010
XOR  10010
----------
      11000
```

The result is therefore 24.

Constants

FALSE

Type: Constant

Description: The constant **FALSE** takes the numerical value of 0.

Example:

```
test:
res=IN(0) OR IN(2)
  IF res=FALSE THEN PRINT "Inputs are off"
ENDIF
```

OFF

Type: Constant

Description: **OFF** returns the value 0

Example:

```
IF IN(56)=OFF THEN GOTO label
'Branch if input 56 is off.
```

ON

Type: Constant

Description: **ON** returns the value 1.

Example:

```
OP(lever,ON)'  This sets the output named lever to ON.
```

TRUE

Type: Constant

Description: The constant `TRUE` takes the numerical value of -1.

Example:

```
t=IN(0)=ON AND IN(2)=ON
IF t=TRUE THEN
  PRINT "Inputs are on"
ENDIF
```

PI

Type: Constant

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

Example:

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

Axis Parameters

ACCEL

Type: Axis Parameter

Syntax: `ACCEL=value`

Description: The **ACCEL** axis parameter may be used to set or read back the acceleration rate of each axis fitted. The acceleration rate is in units/sec/sec.

Example:
```
ACCEL=130:' Set acceleration rate
PRINT " Accel rate:";ACCEL;" mm/sec/sec"
```

ADDAX_AXIS

Type: Axis Parameter (Read Only)

Syntax: `ADDAX_AXIS`

Description: Returns the axis currently linked to with the **ADDAX** command, if none the parameter returns -1.

ATYPE

Type: Axis Parameter

Description: The **ATYPE** axis parameter may be used to test the type of each axis fitted. It will return the values:

ATYPE	Description
0	No axis daughter board fitted
1	Stepper Axis
2	Servo Axis
3	Encoder Reference Axis
4	Stepper with position verification/Differential stepper

ATYPE	Description
5	Resolver Axis
6	Voltage output
7	Absolute SSI Servo
8	CAN daughter board
9	Remote CAN axis
10	PSWITCH Axis
11	Remote DriveLink axis
12	Reserved
13	Embedded axis
14	Encoder Output
15	Reserved
16	Remote SERCOS speed axis
17	Remote SERCOS position axis
18	Remote CanOpen position axis
19	Remote CanOpen speed axis
20	Reserved
21	Remote User Specific CAN axis

The **ATYPE** axis parameter is set by the system software at power up.

Controllers such as the Euro205, MC206 and MC202 an return ATYPE of the equivalent function of the controller. On the MC202 the **ATYPE** parameter can be set to either 1 or 2 to select the axis function. On the Euro205, EURO205X, PCI208 and MC206 the ATYPE can be altered by entering Feature Enable Codes.

Example:

```
>>PRINT ATYPE AXIS(2)
1.0000
```

This would show that an stepper daughter board is fitted in this axis slot.

AXISSTATUS

Type: Axis Parameter (Read Only)

Description: The **AXISSTATUS** axis parameter may be used to check various status bits held for each axis fitted:

Bit	Description	Value	char
0	Unused	1	
1	Following error warning range	2	**w**
2	Communications error to remote drive	4	**a**
3	Remote drive error	8	**m**
4	In forward limit	16	**f**
5	In reverse limit	32	**r**
6	Datuming	64	**d**
7	Feedhold	128	**h**
8	Following error exceeds limit	256	**e**
9	In forward software limit	512	**x**
10	In reverse software limit	1024	**y**
11	Cancelling move	2048	**c**
12	Encoder power supply overload (MC206)	4096	**o**
13	Set on SSI axis after initialisation	8192	

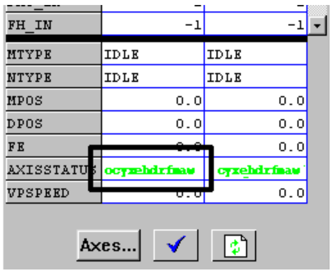
The **AXISSTATUS** axis parameter is set by the system software is read-only..

Example:

```
IF (AXISSTATUS AND 16)>0 THEN
  PRINT "In forward limit"
ENDIF
```

Note: In the *Motion* Perfect parameter screen the **AXISSTATUS** parameter is displayed as a series of characters, **ocyxehdrfmaw**, as listed in the table above.

These characters are displayed in green lowercase letters normally, or red uppercase when set.



FH_IN	-1	-1
MTYPE	IDLE	IDLE
NTYPE	IDLE	IDLE
MPOS	0.0	0.0
DPOS	0.0	0.0
FE	0.0	0.0
AXISSTATUS	ocyxehdrfmaw	cyxehdrfmaw
VPSPEED	0.0	0.0

See Also: **ERRORMASK**

BOOST

Type: Axis Parameter

Syntax: `BOOST=ON / BOOST=OFF`

Description: Sets the boost output on a stepper daughter board. The boost output is a dedicated open collector output on the stepper and stepper encoder daughter boards. The open collector can be switched on or off for each axis using this command.

Example: `BOOST AXIS(11)=ON`

CAN_ADDRESS

Type: Axis Parameter

Description: The `CAN_ADDRESS` axis parameter is used when control is being made of remote servo drives with CAN communications. The `CAN_ADDRESS` holds the address of the remote servo drive.

CAN_ENABLE

Type: Axis Parameter

Description: The `CAN_ENABLE` axis parameter is used when control is being made of the remote servo drives with CAN communications. The `CAN_ENABLE` is used to control the enable on the remote servo drive.

CLOSE_WIN

Type: Axis Parameter

Alternate Format: `CW`

Description: By writing to this parameter the end of the window in which a registration mark is expected can be defined. The value is in user units.

Example: `CLOSE_WIN=10.5`

CLUTCH_RATE

Type: Axis Parameter

Description: This affects operation of **CONNECT** by changing the connection ratio at the specified rate/second.

Default **CLUTCH_RATE** is set very high to ensure compatibility with earlier versions.

Example:

```
CLUTCH_RATE=5
```

CREEP

Type: Axis Parameter

Description: Sets the creep speed on the current base axis. The creep speed is used for the slow part of a DATUM sequence. The creep speed must always be a positive value. When given a **DATUM** move the axis will move at the programmed **SPEED** until the datum input **DATUM_IN** goes low. The axis will then ramp the speed down and start a move in the reversed direction at the **CREEP** speed until the datum input goes high.

The creep speed is entered in units/sec programmed using the unit conversion factor. For example, if the unit conversion factor is set to the number of encoder edges/inch the speed is programmed in INCHES/SEC.

Example:

```
BASE(2)
CREEP=10
SPEED=500
DATUM(4)
CREEP AXIS(1)=10
SPEED AXIS(1)=500
DATUM(4) AXIS(1)
```

DAC

Type: Axis Parameter

Description: Writing to this axis parameter when `SERVO=OFF` allows the user to force a specified voltage on a servo axis. The range of values that a 12 bit DAC can take is:

`DAC=-2048` corresponds to a voltage of 10V

to

`DAC=2047` corresponds to a voltage of -10v

The range of values that a 16 bit DAC (MC206 built-in axes only) can take is:

`DAC=32767` corresponds to a voltage of 10V

to

`DAC=-32768` corresponds to a voltage of -10v

Note: The SERVO DAUGHTER BOARD/MC202 hardware inverts the signal compared to the number.

Example: To force a square wave of amplitude +/-5V and period of approximately 500ms on axis 0.

```
WDOG=ON
SERVO AXIS(0)=OFF
square:
  DAC AXIS(0)=1024
  WA(250)
  DAC AXIS(0)=-1024
  WA(250)
GOTO square
```

DAC_OUT

Type: Axis Parameter (Read Only)

Description: The axis DAC is the electronics hardware used to output +/-10volts to the servo drive when using a servo daughter board. The **DAC_OUT** parameter allows the value being used to be read back. The value put on the DAC comes from 2 potential sources:

If the axis parameter **SERVO** is set **OFF** then the axis parameter DAC is written to the axis hardware. If the **SERVO** parameter is **ON** then a value calculated using the servo algorithm is placed on the DAC. Either case can be read back using **DAC_OUT**. Values returned will be in the range -2048 to 2047.

Example:

```
>>PRINT DAC_OUT AXIS(8)
288.0000
>>
```

DAC_SCALE

Type: Axis Parameter

Description: The **DAC_SCALE** axis parameter is only available on the MC206, MC224, EURO205X and PCI208. The parameter has 2 purposes: It is set to value 16 on power up on the built-in axes of the MC206. This scales the values applied to the higher resolution DAC so that the gains required on the axis are similar to those required on the other controllers. **DAC_SCALE** may be set negative (-16) to reverse the polarity of the DAC output signal. When the servo is off the magnitude of **DAC_SCALE** is not important as the voltage applied is controlled by the DAC parameter. The polarity is still reversed however by **DAC_SCALE**. When a Servo Daughter Board is used in an MC206 the default **DAC_SCALE** parameter for that axis will be 1.

Example:

```
>>DAC_SCALE AXIS(3)=-16
>>
```

DATUM_IN

Type: Axis Parameter

Alternate Format: `DAT_IN`

Description: This parameter holds a digital input channel to be used as a datum input. The input can be in the range 0..31. If `DATUM_IN` is set to -1 (default) then no input is used as a datum.

The same input may also be used as a limit input if required.

Example: `DATUM_IN AXIS(0)=28`

Note: Feedhold, forward, reverse, datum and jog inputs are ACTIVE LOW.

DECEL

Type: Axis Parameter

Syntax: `DECEL=value`

Description: The `DECEL` axis parameter may be used to set or read back the deceleration rate of each axis fitted. The deceleration rate will be returned in units/sec/sec.

Example:

```
DECEL=100' Set deceleration rate
PRINT " Decel is ";DECEL;" mm/sec/sec"
```

DEMAND_EDGES

Type: Axis Parameter (Read Only)

Description: Allows the user to read back the current `DPOS` in encoder edges.

Example: `>>PRINT DEMAND_EDGES AXIS(4)`

DPOS

Type: Axis Parameter (Read Only)

Description: The demand position **`DPOS`** is the demanded axis position generated by the move commands. Its value may also be adjusted without doing a move by using the **`DEFPOS()`** or **`OFFPOS`** commands. It is reset to 0 on power up or software reset. The demand position must never be written to directly although a value can be forced to create a step change in position by writing to the **`ENDMOVE`** parameter if no moves are currently in progress on the axis.

Example:

```
>>? DPOS AXIS(10)
```

This will return the demand position in user units.

DRIVE_STATUS

Type: Axis Parameter

Alternate Format: **`AMP_STATUS`**

Description: Returns the status register of a drive with digital communications capability connected to the *Motion Coordinator*.

In the case of an SLM axis it returns the SLM and drive status:
Bits 0..7 return bits 0..7 of register 0x8000 on the drive. Bits 8..23 return register 0xD000 on the SLM.

Example:

```
>>PRINT DRIVE_STATUS AXIS(8)
0.0000
>>
```

D_GAIN

Type: Axis Parameter

Syntax: **`D_GAIN=value`**

Description: The derivative gain is a constant which is multiplied by the change in following error.

Adding derivative gain to a system is likely to produce a smoother response and allow the use of a higher proportional gain than could otherwise be used.

High values may lead to oscillation. For a derivative term *Kd* and a change in following error *Đe* the contribution to the output signal is:

$$Od = Kd \times Đe$$

Example: `D_GAIN=0.25`

Note: Servo gains have no effect in stepper motor axes.

ENCODER

Type: Axis Parameter (Read Only)

Description: The **ENCODER** axis parameter holds a raw copy of the encoder or resolver hardware register. On Servo daughter boards, for example, this is a 12 bit (Modulo 4096) number. On SSI absolute daughter boards the **ENCODER** register holds a value of the numbers of bits programmed with **SSI_BITS**. The **MPOS** axis measured position is calculated from the **ENCODER** value automatically allowing for overflows and offsets. On MC202 and the built-in axes of a Euro205 or MC206 the **ENCODER** register is a 14 bit register.

ENDMOVE

Type: Axis Parameter

Description: This parameter holds the position of the end of the current move in user units. It is normally only read back although may be written to if required provided that **SERVO=ON** and no move is in progress. This will produce a step change in **DPOS**. Making step changes in **DPOS** can easily lead to "Following error exceeds limit" errors unless the steps are small or the **FE_LIMIT** is high.

ERRORMASK

Type: Axis Parameter

Description: The value held in this parameter is bitwise **AND**ed with the **AXISSTATUS** parameter by every axis on every servo cycle to determine if a runtime error should switch off the enable (**WDOG**) relay. If the result of the **AND** operation is not zero the enable relay is switched OFF.

On the MC202, Euro 205 and MC216 the default setting is 256. This will trip the enable relay only if a following error condition occurs.

For the MC206, the default value is 268 which is set to also trap critical errors with digital drive communications.

See Also: **AXISSTATUS**

FAST_JOG

Type: Axis Parameter

Description: This parameter holds the input number to be used as the fast jog input. The input can be in the range 0..31. If **FAST_JOG** is set to -1 (default) then no input is used for the fast jog. If the **FAST_JOG** is asserted then the jog inputs use the axis **SPEED** for the jog functions, otherwise the **JOGSPEED** will be used.

Note: Feedhold, forward, reverse, datum and jog inputs are ACTIVE LOW.

FASTDEC

Type: Axis Parameter

Description: The **FASTDEC** axis parameter is a reserved word, but is not currently used in the motion generator program.

FE

Type: Axis Parameter (Read Only)

Description: This parameter is the position error, which is equal to the demand position(**DPOS**)-measured position(**MPOS**). The parameter is returned in user units.

FE_LIMIT

Type: Axis Parameter

Alternate Format: **FELIMIT**

Syntax: **FE_LIMIT=value**

Description: This is the maximum allowable following error. When exceeded the controller will generate a run time error and always resets the enable (**WDOG**) relay thus disabling further motor operation. This limit may be used to guard against fault conditions such as mechanical lock-up, loss of encoder feedback, etc. It is returned in USER UNITS.

The default value is 2000 encoder edges.

FERANGE

Type: Axis Parameter

Syntax: `FERANGE=value`

Description: Following error report range. When the following error exceeds this value on a servo axis the axis has bit 1 in the **AXISSTATUS** axis parameter set.

FEGRAD

Type: Axis Parameter

Syntax: `FEGRAD=value`

Description: Following error limit gradient. Specifies the allowable increase in following error per unit increase in velocity profile speed. The parameter is not currently used in the motion generator program.

FEMIN

Type: Axis Parameter

Syntax: `FEMIN=value`

Description: Following error limit at zero speed. The parameter is not currently used in the motion generator program.

FHOLD_IN

Type: Axis Parameter

Alternate Format: `FH_IN`

Syntax: `FHOLD_IN=value`

Description: This parameter holds the input number to be used as a feedhold input. The input can be in the range 0..31. If **`FHOLD_IN`** is set to -1 (default) then no input is used as a feedhold. When the feedhold input is set motion on the specified axis has its speed overridden to the Feedhold speed (**`FHSPEED`**) WITHOUT CANCELLING THE MOVE IN PROGRESS. This speed is usually zero. When the input is reset any move in progress when the input was set will go back to the programmed speed. Moves which are not speed controlled E.G. **`CONNECT`**, **`CAMBOX`**, **`MOVELINK`** are not affected.

Note: Feedhold, forward, reverse, datum and jog inputs are ACTIVE LOW.

FHSPEED

Type: Axis Parameter

Syntax: **`FHSPEED=value`**

Description: When the feedhold input is set motion is usually ramped down to zero speed as the feedhold speed is set to its default zero value. In some cases it may be desirable for the axis to ramp to a known constant speed when the feedhold input is set. To do this the **`FHSPEED`** parameter is set to a non zero value. The value is in user units/sec.

FS_LIMIT

Type: Axis Parameter

Alternate Format: **`FSLIMIT`**

Description: An end of travel limit may be set up in software thus allowing the program control of the working envelope of the machine. This parameter holds the absolute position of the forward travel limit in user units. When the limit is hit the controller will ramp down the speed to zero then cancel the move. Bit 9 of the **`AXISSTATUS`** register is set when the axis position is greater than the **`FS_LIMIT`**.

FWD_IN

Type: Axis Parameter

Description: This parameter holds the input number to be used as a forward limit input. The input can be in the range 0..31. If **`FWD_IN`** is set to -1 (default) then no input is used as a forward limit. When the forward limit input is asserted any forward motion on that axis is stopped.

Example: **`FWD_IN=19`**

Note: Feedhold, jog forward, reverse and datum inputs are ACTIVE LOW.

FWD_JOG

Type: Axis Parameter

Description: This parameter holds the input number to be used as a jog forward input. The input can be in the range 0..31. If **`FWD_JOG`** is set to -1 (default) then no input is used as a forward jog.

Example: **`FWD_JOG=7`**

Note: Feedhold, forward, reverse, datum and jog inputs are ACTIVE LOW.

INVERT_STEP

Type: Axis Parameter

Description: **`INVERT_STEP`** is used to switch a hardware inverter into the stepper pulse output circuit. This can be necessary in for connecting to some stepper drives. The electronic logic inside the *Motion Coordinator* stepper pulse generation assumes that the FALLING edge of the step output is the active edge which results in motor movement. This is suitable for the majority of stepper drives. Setting **`INVERT_STEP=ON`** effectively makes the RISING edge of the step signal the active edge. **`INVERT_STEP`** should be set if required prior to enabling the controller with **`WDOG=ON`**. Default=OFF.

Note: If the setting is incorrect. A stepper motor may lose position by one step when changing direction.

I_GAIN

Type: Axis Parameter

Description: The integral gain is a constant which is multiplied by the sum of following errors of all the previous samples. This term may often be set to 0 (Default). Adding integral gain to a servo system reduces position error when at rest or moving steadily but it will produce or increase overshoot and may lead to oscillation.

For an integral gain *Ki* and a sum of position errors $\oplus e$, the contribution to the output signal is:

$$Oi = Ki \times \oplus e$$

Note: Servo gains have no effect on stepper motor axes.

JOGSPEED

Type: Axis Parameter

Description: Sets the slow jog speed in user units for an axis to run at when performing a slow jog. A slow jog will be performed when a jog input for an axis has been declared and that input is low. The jog will be at the **JOGSPEED** provided the **FAST_JOG** input has not be declared and is set low. Two separate jog inputs are available for each axis **FWD_JOG** and **REV_JOG**.

LINKAX

Type: Axis Parameter (Read Only)

Description: Returns the axis number that the axis is linked to during any linked moves. Linked moves are where the demand position is a function of another axis. E.G. **CONNECT, CAMBOX, MOVELINK**

MARK

Type: Axis Parameter (Read Only)

Description: Returns **TRUE** when a registration event has occurred. This is set to **FALSE** by the **REGIST** command and set to true when the registration event occurs. When **TRUE** the **REG_POS** is valid.

Example:

```
loop:
  WAIT UNTIL IN(punch_clr)=ON
  MOVE(index_length)
  REGIST(3)' rising edge of R
  WAIT UNTIL MARK  MOVEMODIFY(REG_POS + offset)
  WAIT IDLE
GOTO loop
```

MARKB

Type: Axis Parameter (Read Only)

Description: Only usable on MC206 built-in axes. MARKB returns **TRUE** when the Z registration position has been latched. This is set to **FALSE** by the **REGIST** command and set to true when the registration event occurs. When **TRUE** the **REG_POSB** is valid. See REGIST() and REG_POSB.

MERGE

Type: Axis Parameter

Syntax: **MERGE=ON / MERGE=OFF**

Description: This is a software switch which can be used to enable or disable the merging of consecutive moves. With merging enabled, if the next move is already in the buffer the axis will not ramp down to zero speed but load up the following move allowing them to be seamlessly merged. Note that it is up to the programmer to ensure that the merging is sensible. For example merging a forward move with a reverse move will cause an attempted instantaneous change of direction.

MERGE will only function if:

1) The next move is loaded
2) Axis group does not change on multi-axis moves
3) Velocity profiled moves (**MOVE**, **MOVEABS**, **MOVECIRC**, **MHELICAL**, **REVERSE**, **FORWARD**) cannot be merged with linked moves (**CONNECT**,**MOVELINK**,**CAMBOX**)

Note: When merging multi-axis moves only the base axis **MERGE** flag needs to be set.

If the moves are short a high deceleration rate must be set to avoid the controller ramping the speed down in anticipation of the end of the buffered move

Example:

```
MERGE=OFF'   Decelerate at the end of each move
MERGE=ON'    Moves will be merged if possible
```

MICROSTEP

Type: Axis Parameter

Description: Sets microstepping mode when using a stepper daughter board, MC202 or Euro205. On these controllers the stepper pulse circuit contains a circuit which places the step pulses more evenly in time by dividing the pulse rate by 2 or 16:

MICROSTEP=OFF (DEFAULT)	62.5 kHz Maximum
MICROSTEP=ON	500 kHz Maximum

(On the MC206, PCI208 and EURO205X a different pulse generation circuit is used which always divides the pulse rate by 16 and is NOT affected by the MICROSTEP parameter. This circuit can generate pulses up to 2Mhz) The stepper daughter board can generate pulses at up to 62500 Hz with **MICROSTEP=OFF** (This is the default setting and should be used when the pulse rate does not exceed 62500 Hz even if the motor is microstepping) With **MICROSTEP=ON** the stepper board can generate pulses at up to 500,000 Hz although the pulses are not so evenly spaced in time.

With **MICROSTEP=OFF** the **UNITS** parameter should be set to 16 times the number of pulses in a distance parameter. With **MICROSTEP=ON** the **UNITS** should be set to 2 times the number.

Example:

```
UNITS AXIS(2)=180*2' 180 pulses/rev * 2
MICROSTEP AXIS(2)=ON
```

MPOS

Type: Axis Parameter (Read Only)

Description: This parameter is the position of the axis as measured by the encoder or resolver. It is reset to 0 (unless a resolver is fitted) on power up or software reset. The value is adjusted using the **DEFPOS()** command or **OFFPOS** axis parameter to shift the datum position or when the **REP_DIST** is in operation. The position is reported in user units.

Example:

```
WAIT UNTIL MPOS>=1250
SPEED=2.5
```

MSPEED

Type: Axis Parameter (Read Only)

Description: The **MSPEED** represents the change in measured position in user units (per second) in the last servo period. The **SERVO_PERIOD** defaults to 1mSec. It therefore can be used to represent the speed measured. This value represents a snapshot of the speed and significant fluctuations can occur, particularly at low speeds. It can be worthwhile to average several readings if a stable value is required at low speeds.

MTYPE

Type: Axis Parameter (Read Only)

Description: This parameter holds the type of move currently being executed.

MTYPE	Move Type
0	Idle (No move)
1	MOVE
2	MOVEABS
3	MHELICAL
4	MOVECIRC
5	MOVEMODIFY
10	FORWARD
11	REVERSE

MTYPE	Move Type
12	DATUMING
13	CAM
14	Forward Jog
15	Reverse Jog
20	CAMBOX
21	CONNECT
22	MOVELINK

This parameter may be interrogated to determine whether a move has finished or if a transition from one move type to another has taken place.

A non-idle move type does not necessarily mean that the axis is actually moving. It may be at zero speed part way along a move or interpolating with another axis without moving itself.

Note that testing MTYPE=0 is not equivilent to WAIT IDLE. WAIT IDLE is equivilent to WAIT UNTIL (MTYPE=0) AND (NTYPE=0) AND (PMOVE=0).

Example: Load a move and print its type. Note how because the MOVE becomes active on the next servo cycle the programmer MUST wait for the move to become active, prior to printing the type:

```
MOVE(1000)
WAIT LOADED
PRINT MTYPE
```

NTYPE

Type: Axis Parameter (Read Only)

Description: This parameter holds the type of the next buffered move. The values held are as for **MTYPE**. If no move is buffered zero will be returned. The **NTYPE** parameter is read only but the **NTYPE** can be cleared using **CANCEL(1)**

OFFPOS

Type: Axis Parameter

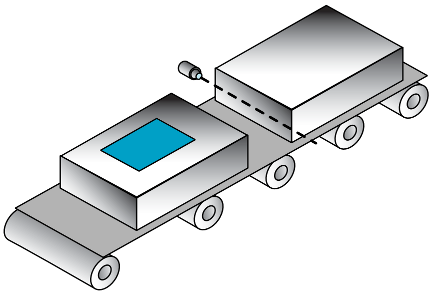
Description: The **OFFPOS** parameter allows the axis position to be offset by any value without affecting motion. **OFFPOS** can therefore be used to effectively datum a system at full speed.Values loaded into the **OFFPOS** axis parameter are reset to 0 by the system software as the axis position is changed.

Example: Define the current demand position as zero:

```
OFFPOS=-DPOS
WAIT UNTIL OFFPOS=0'   wait until applied
```

This is equivalent to **DEFPOS(0)**

Example 2: A conveyor is used to transport boxes onto which labels must be applied.



Using the **REGIST()** function, we can capture the position at which the leading edge of the box is seen, then by using **OFFPOS** we can adjust the measured position of the axis to be zero at that point. Therefore, after the registration event has occurred, the measured position (seen in **MPOS**) will actually reflect the absolute distance from the start of the box, the mechanism which applies the label can take advantage of the absolute position start mode of the **MOVELINK** or **CAMBOX** commands to apply the label.

```
BASE(conv)
REGIST(3)
WAIT UNTIL MARK
OFFPOS = -REG_POS ‘ Leading edge of box is now zero
```

Note: The **OFFPOS** adjustment is executed on the next servo period. Several Trio BASIC instructions may occur prior to the next servo period. Care must be taken to ensure these instructions do not assume the position shift has occurred.

OPEN_WIN

Type: Axis Parameter

Alternate Format: **OW**

Description: This parameter defines the position before which registration marks will be ignored if windowing is specified by the **REGIST()** command.

OUTLIMIT

Type: Axis Parameter

Description: The output limit restricts the voltage output from a servo axis to a lower value than the maximum. The value required varies depending on whether the axis has a 12 bit or 16 bit DAC. If the voltage output is generated by a 12 bit DAC values an OUTLIMIT of 2047 will produce the full +/-10v range. If the voltage output is generated by a 16 bit DAC values an OUTLIMIT of 32767 will produce the full +/-10v range. Only the MC206 internal axes have 16 bit DAC's

Example:
```
OUTLIMIT AXIS(0)=1023
```

The above will limit the voltage output to a ±5V output range on a servo daughter board axis. This will apply to the **DAC** command if **SERVO=OFF** or to the voltage output by the servo if **SERVO=ON**.

OV_GAIN

Type: Axis Parameter

Description: The output velocity gain is a gain constant which is multiplied by the change in measured position. The result is summed with all the other gain terms and applied to the servo DAC. Default value is 0. Adding NEGATIVE output velocity gain to a system is mechanically equivalent to adding damping. It is likely to produce a smoother response and allow the use of a higher proportional gain than could otherwise be used, but at the expense of higher following errors. High values may lead to oscillation and produce high following errors. For an output velocity term Kov and change in position ΔPm, the contribution to the output signal is:

$$Oov=Kov \times \Delta Pm$$

Note: Negative values are normally required. Servo gains have no effect on stepper motor axes.

PP_STEP

Type: Axis parameter

Description: This parameter allows the incoming raw encoder counts to be multiplied by an integer value in the range -1024 to 1023. This can be used to match encoders to high resolution microstepping motors for position verification or for moving along circular arcs on machines where the number of encoder edges/distance do not match on the axes. Using a negative number will reverse the encoder count.

Example: A microstepping motor has 20000 steps/rev. The *Motion Coordinator* is working in **MICROSTEP=ON** mode so will internally process 40000 counts/rev. A 2500 pulse encoder is to be connected. This will generate 10000 edge counts/rev. A multiplication factor of 4 is therefore is required to convert the 10000 counts/rev to match the 40000 counts/rev of the motor.

```
PP_STEP AXIS(3)=4
```

Example 2: An X-Y machine has encoders which give 50 edges/mm in the X axis (Axis 0) and 75 edges/mm in the Y axis (Axis 1). Circular arc interpolation is required between the axes. This requires that the interpolating axes have the same number of encoder counts/distance. It is not possible to multiply the X axis counts by 1.5 as the **PP_STEP** parameter must be an integer. Both X and Y axes must therefore be set to give 150 edges/mm:

```
PP_STEP AXIS(0)=3
PP_STEP AXIS(1)=2
UNITS AXIS(0)=150
UNITS AXIS(1)=150
```

Note: In a servo loop, increasing PP_STEP will require a proportionate decrease of loop gains.

P_GAIN

Type: Axis Parameter

Description: The proportional gain sets the 'stiffness' of the servo response. Values that are too high will produce oscillation. Values that are too low will produce large following errors.
For a proportional gain *Kp* and position error *e*, its contribution to the output signal is:

$$Op = Kp \times e$$

Note: `P_GAIN` may be fractional values. The default value is 1.0. Servo gains have no effect on stepper motor axes.

Example: `P_GAIN AXIS(11)=0.25`

REG_MATCH

Type: Axis Parameter (Read Only)

Description: Indicates to a programmer the quality of the fit of a **RECORD** / **MATCH** sequence. A value of 1 is returned if the fit is 100%.

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

REG_POS

Type: Axis Parameter (Read Only)

Alternate Format: **RPOS**

Description: Stores the position at which a registration mark was seen on each axis in user units. See **REGIST()** for more details.

Example: A paper cutting machine uses a **CAM** profile shape to quickly draw paper through servo driven rollers then stop it whilst it is cut. The paper is printed with a registration mark. This mark is detected and the length of the next sheet is adjusted by scaling the CAM profile with the third parameter of the **CAM** command:

```
'  Example Registration Program using CAM stretching:
' Set window open and close:
  length=200
  OPEN_WIN=10
  CLOSE_WIN=length-10
  GOSUB Initial
Loop:
  TICKS=0' Set millisecond counter to 0
  IF MARK THEN
    offset=REG_POS
  '  This next line makes offset -ve if at end of sheet:
    IF ABS(offset-length)<offset THEN offset=offset-length
```

```
        PRINT "Mark seen at:"offset[5.1]
    ELSE
        offset=0
      PRINT "Mark not seen"
      ENDIF

    ' Reset registration prior to each move:
    DEFPOS(0)
      REGIST(3+768)' Allow mark at first 10mm/last 10mm of sheet
      CAM(0,50,(length+offset*0.5)*cf,1000)
    WAIT UNTIL TICKS<-500
    GOTO Loop
```

(variable “cf” is a constant which would be calculated depending on the machine draw length per encoder edge)

REG_POSB

Type: Axis Parameter (Read Only)

Description: Useable only on MC206 built-in axes. REG_POSB returns the position at which a registration Z mark was seen on an axis. See **REGIST()** for more details.

REMAIN

Type: Axis Parameter (Read Only)

Description: This is the distance remaining to the end of the current move. It may be tested to see what amount of the move has been completed. The units are user distance units.

Example: To change the speed to a slower value 5mm from the end of a move.

```
start:
  SPEED=10
  MOVE(45)
  WAIT UNTIL REMAIN<5
  SPEED=1
  WAIT IDLE
```

REMOTE_ERROR

Type: Axis Parameter

Description: Returns the number of errors on a drive’s digital communication link.

Example:

```
>>PRINT REMOTE_ERROR
1.0000
>>
```

REPDIST

Type: Axis Parameter

Description: The repeat distance contains the allowable range of movement for an axis before the position count overflows or underflows. For example, when an axis executes a **FORWARD** move the demand and measured position will continually increase. When the measured position reaches the **REPDIST** twice that distance is subtracted to ensure that the axis always stays in the range **-REPEAT DISTANCE** to **+REPEAT DISTANCE** (Assuming **REP_OPTION=OFF**). The *Motion Coordinator* will adjust its absolute position without affecting the move in progress or the servo algorithm.

REP_OPTION

Type: Axis Parameter

Description: Bit 0 of the **REP_OPTION** parameter controls the way the REPDIST is applied. In the default setting (**REP_OPTION bit 0=0**) REPDIST operation is selected in the range **-REPEAT DISTANCE** to **+REPEAT DISTANCE**. In some circumstances it more convenient for the axis positions to be specified from 0 to **+REPEAT DISTANCE**. (**REP_OPTION bit 0=1**)

REP_OPTION bit 1 is set **ON** to switch OFF the automatic repeat option of the **CAMBOX** or MOVELINK function. When the system software has set the option OFF it automatically clears bit 1 of **REP_OPTION**.

REV_IN

Type: Axis Parameter

Description: This parameter holds the input number to be used as a reverse limit input. The input should be in the range 0..31. If **`REV_IN`** is set to -1 (default) then no input is used as a reverse limit. When the reverse limit input is asserted moves going in the reverse direction will be cancelled. The axis status bit 5 will also be set.

Note: Feedhold, forward, reverse and datum inputs are ACTIVE LOW.

REV_JOG

Type: Axis Parameter

Description: This parameter holds the input number to be used as a reverse jog input. The input should be in the range 0..31. If **`REV_JOG`** is set to -1 (default) then no input is used as a reverse jog. When the input is asserted then the axis is moved forward at the **`JOGSPEED`** or axis **`SPEED`** depending on the status of the **`FAST_JOG`** input.

Note: Feedhold, forward, reverse and datum inputs are ACTIVE LOW.

RS_LIMIT

Type: Axis Parameter

Alternate Format: **`RSLIMIT`**

Description: An end of travel software limit may be set up in software thus allowing the program control of the working envelope of the machine. This parameter holds the absolute position of the reverse travel limit in user units. When the limit is hit the controller will ramp down the speed to zero then cancel the move. Bit 10 in the axis status parameter is set when the axis is in the **`RS_LIMIT`**

SERVO

Type: Axis Parameter

Description: On a servo axis this parameter determines whether the axis runs under servo control or open loop. When **SERVO=OFF** the axis hardware will output a voltage dependent on the DAC parameter. When **SERVO=ON** the axis hardware will output a voltage dependent on the gain settings and the following error.

SERVO is also used on stepper axes with position verification. If **SERVO=ON** the system software will compare the difference between the **DPOS** and **MPOS (FE)** on the axis with the **FE_LIMIT**. If the difference exceeds the limit the following error bit is set in the **AXISSTATUS** register, the enable relay is forced OFF and the servo is set OFF. If the **SERVO=OFF** on a stepper verification axis the **FE** is not compared with the **FE_LIMIT**.

Example:

```
SERVO AXIS(0)=ON'   Axis 0 is under servo control
SERVO AXIS(1)=OFF'  Axis 1 is run open loop
```

Note: Stepper axes with position verification need consideration also of **VERIFY** and **PP_STEP**.

SP

Type: Axis Command - Use the **SPEED** axis parameter for new applications.

Syntax: **SP(speed)**

Description: This format is only provided to simplify compatibility with earlier controllers. Sets demand speed of the current or base axis.

SPEED

Type: Axis Parameter

Description: The **SPEED** axis parameter can be used to set/read back the demand speed axis parameter. The speed is returned in units/s. The demand speed is the speed ramped up to during the movement commands **MOVE**, **MOVEABS**, **MOVECIRC**, **FORWARD**, **REVERSE**, **MHELICAL and MOVEMODIFY.**

Example:
```
SPEED=1000
PRINT “Speed Set=”;SPEED
```

SRAMP

Type: Axis Parameter

Description: This parameter stores the s-ramp factor. This controls the amount of rounding applied to trapezoidal profiles. 0 sets no rounding. 10 maximum rounding. Using S ramps increases the time required for the movement to complete. **SRAMP** can be used with **MOVE**, **MOVEABS**, **MOVECIRC**, **MHELICAL**, **FORWARD**, **REVERSE** and **MOVEMODIFY** move types.

Note: The **SRAMP** factor should not be changed while a move is in progress.

SSI_BITS

Type: Axis Parameter

Description: This parameter is only used with the SSI Absolute daughter board. It is used to set the number of data bits to be clocked out of the encoder by the axis hardware. The maximum permitted value is 24. The default value is 0 which will cause no data to be clocked from the SSI encoder, users MUST therefore set a value depending on the encoder type.

If the number of SSI_BITS is to be changed, the parameter must first be set to zero before entering the new value.

Example:
```
SSI_BITS AXIS(3)=12
SSI_BITS AXIS(7)=21
```

Example2:

```
're-initialise MPOS using absolute value from encoder
SERVO=OFF
SSI_BITS =0
SSI_BITS =databits
```

TRANS_DPOS

Type: Axis Parameter (Read Only)

Description: Axis demand position at output of frame transformation. **TRANS_DPOS** is normally equal to **DPOS** on each axis. The frame transformation is therefore equivalent to 1:1 for each axis. For some machinery configurations it can be useful to install a frame transformation which is not 1:1, these are typically machines such as robotic arms or machines with parasitic motions on the axes. Frame transformations have to be specially written in the “C” language and downloaded into the controller. It is essential to contact Trio if you want to install frame transformations.

Note: See also **FRAME**

TRANSITIONS

Type: Axis Parameter

Description: Records the number of register input transitions in a **REGIST** sequence

See also **REGIST**, **RECORD**, **MATCH**

UNITS

Type: Axis Parameter

Description: The unit conversion factor sets the number of encoder edges/stepper pulses in a user unit. The motion commands to set speeds, acceleration and moves use the **UNITS** parameter to allow values to be entered in more convenient units e.g.: mm for a move or mm/sec for a speed.

Note: Units may be any positive value but it is recommended to design systems with an integer number of encoder pulses/user unit.

Example: A leadscrew arrangement has a 5mm pitch and a 1000 pulse/rev encoder. The units should be set to allow moves to be specified in mm. The 1000 pulses/rev will generate 1000 x 4=4000 edges/rev. One rev is equal to 5mm therefore there are 4000/5=800 edges/mm so:

```
>>UNITS=1000*4/5
```

Example 2: A stepper motor has 180 pulses/rev and is being used with **MICROSTEP=OFF**

To program in revolutions the unit conversion factor will be:

```
>>UNITS=180*16
```

Note: Users with stepper axes should also read the **MICROSTEP** command when choosing **UNITS**.

VERIFY

Type: Axis Parameter

Description: The verify axis parameter is used to select different modes of operation on a stepper encoder axis.

VERIFY=OFF
Encoder count circuit is connected to the **STEP** and **DIRECTION** hardware signals so that these are counted as if they were encoder signals. This is particularly useful for registration as the registration circuit can therefore function on a stepper axis.

VERIFY=ON
Encoder circuit is connected to external A,B, Z signal

Note: On the MC202 and the Euro205 when **VERIFY=OFF**, the encoder counting circuit is configured to accept **STEP** and **DIRECTION** signals hard wired to the encoder A and B inputs. If **VERIFY=ON**, the encoder circuit is configured for the usual quadrature input.

Take care that the encoder inputs do not exceed 5 volts.

Example:

```
VERIFY AXIS(3)=ON
```

VFF_GAIN

Type: Axis Parameter

Description: The velocity feed forward gain is a constant which is multiplied by the change in demand position. Adding velocity feed forward gain to a system decreases the following error during a move by increasing the output proportionally with the speed. For a velocity feed forward term *Kvff* and change in position ΔPd, the contribution to the output signal is:

$$Ovff=Kvff \times \Delta Pd$$

Note: Servo gains have no effect on stepper motor axes.

VP_SPEED

Type: Axis Parameter (Read Only)

Alternate Format: `VPSPEED`

Description: The velocity profile speed is an internal speed which is ramped up and down as the movement is velocity profiled. It is reported in user units/sec.

Example: Wait until command speed is achieved:

```
MOVE(100)
WAIT UNTIL SPEED=VP_SPEED
```

CHAPTER

9

TRIO BASIC PROGRAMMING EXAMPLES

Example Programs

1 - Fetching an Integer Value from the Membrane Keypad

The subroutine “getnum” fetches an integer value from the membrane keypad in variable “num”. The routine prints the number on the display bottom line at cursor position 70, although this can be set to other values. Only the number keys, the “CLR” key and the ENTER key are used. Other keys are ignored.

```
' Demonstrate integer number entry via Membrane Keypad:
getnum: pos=70
  num=0
  PRINT#4,CHR(20);
  REPEAT
    PRINT#4,CURSOR(pos);num[6,0];
    GET#4,k
    IF k=69 THEN GOTO getnum
    IF k>=59 AND k<=61 THEN k=k-7
    IF k>=66 AND k<=68 THEN k=k-17
    IF k=71 THEN k=48
    IF k>47 AND k<58 THEN
      k=k-48
      num=num*10+k
    ENDIF
  UNTIL k=73
RETURN
```

Example 2 - Fetching a Real Value from the Membrane Keypad

This similar routine also fetches a number from the membrane keypad, but this number can have up to 2 decimal places. Note how this example uses the emulated keypad from *Motion* Perfect.

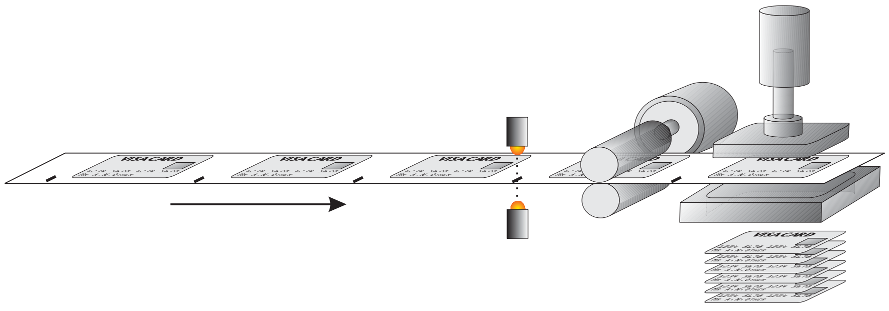
```
getnum:
  pos=40
  dpoint=0
  num=0
  negative=1
  PRINT#5,CHR(20);
  REPEAT
    PRINT#5,CURSOR(pos);num*negative[8,2];
```

```
        GET#5,k
        IF k=72 AND dpoint=0 THEN dpoint=1
        IF k=70 THEN negative=-negative
        IF k=69 THEN GOTO getnum
        IF k>=59 AND k<=61 THEN k=k-7
        IF k>=66 AND k<=68 THEN k=k-17
        IF k=71 THEN k=48
        IF k>47 AND k<58 THEN
          k=k-48
          IF dpoint>0 THEN
            dpoint=dpoint/10
            IF dpoint>=0.01 THEN num=num+k*dpoint
          ELSE
            num=num*10+k
          ENDIF
        ENDIF
      UNTIL k=73
      num=num*negative
    RETURN
```

Example 3 - ATM Card Production

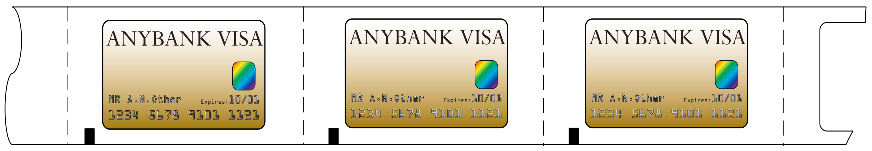
Key Features Used: `REGIST, MOVEMODIFY`

An automated die-cutting machine, is designed to punch out pre-printed plastic cards for use in ATM machines etc.



There is one servo axis which is connected to the draw rollers which feed the card into the machine. A printed registration mark appears once per card and is sensed by an optical sensor connected to the Registration input of the MC2xx's Servo Daughter Board.

The operation of the machine is quite simple, the cards are printed at a known fixed-pitch. Each cycle, the draw rolls must feed the card into position, an output is then fired to operate the punch. An input signals that the punch is clear of the cards and the cycle can repeat.



In an ideal situation we would simply datum the first card and then move a fixed pitch every cycle,

```
loop:
  MOVE(card_pitch)
  WAIT IDLE
  OP(punch,ON)
  WAIT UNTIL IN(punch_clear)=OFF
  WAIT UNTIL IN(punch_clear)=ON
  OP(punch,OFF)
GOTO loop
```

In the real world we must allow for mechanical slippage and any inconsistencies which may occur in the printing. Therefore we will use the registration mark to synchronise the position of the draw each cycle

```
loop:
  DEFPOS(0)
  REGIST(3)
  MOVE(card_pitch)
  WAIT UNTIL MARK
  MOVEMODIFY(REG_POS+20)
  WAIT IDLE
  OP(punch,ON)
  WAIT UNTIL IN(punch_clear)=OFF
  WAIT UNTIL IN(punch_clear)=ON
  OP(punch,OFF)
GOTO loop
```

The above example shows only the simplest form of the main loop. It allows for a fixed offset value of 20, but there is no provision for error handling etc. An example where the code might be expanded to check for registration errors would be:

```
loop:
  DEFPOS(0)
  REGIST(3)
  MOVE(card_pitch)
  WAIT UNTIL MARK OR MTYPE=0
  IF MTYPE=0 THEN
    ' Indicate error to user
    PRINT #3,"Registration Error!"
    errors=errors+1
    if errors>max_errors then GOTO reg_failed
    OP(error_lamp,ON)
  ELSE
    OP(error_lamp,OFF)
    MOVEMODIFY(reg_pos+20)
    WAIT IDLE
  ENDIF
  ' Rest of loop as before
GOTO loop

reg_failed:
  PRINT #3,CURSOR(00);"Too many reg errors!"
  PRINT #3,CURSOR(20);"Press any key...    "
  GET #3,k
GOTO start
```

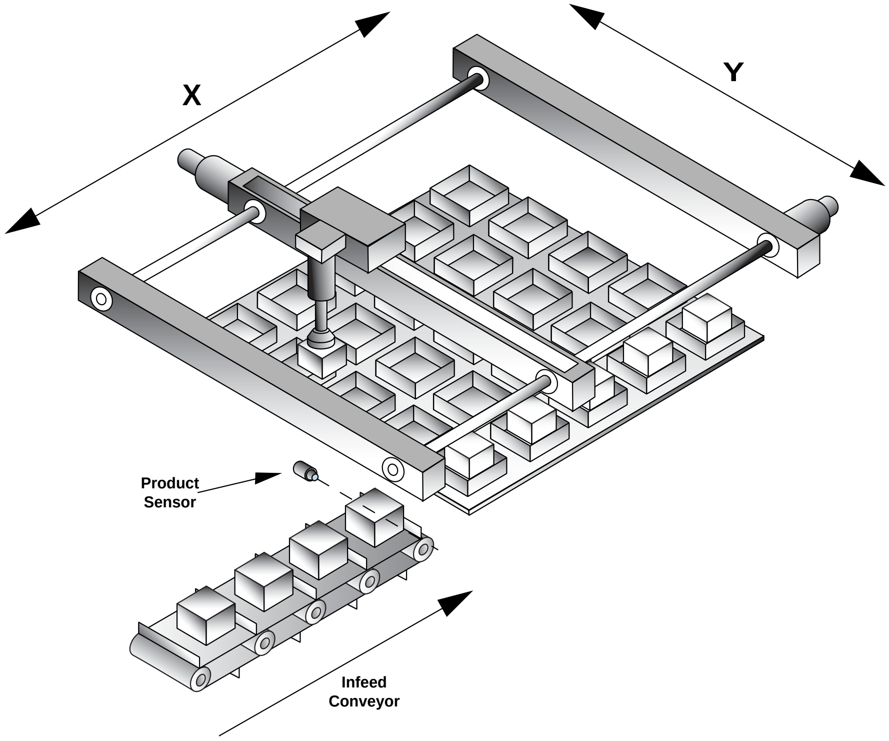
Example 4 - 2 Axis Pick & Place System

Overview

A square palette has sides 1200mm long.

It must be divided into a grid, each of these positions on the palette contains a box into which a widget must be placed:

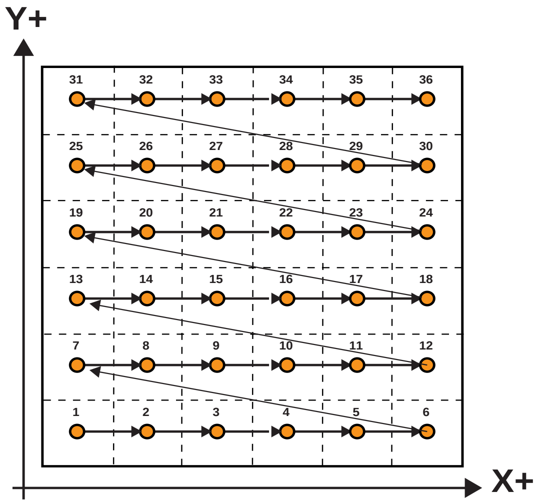
A vacuum operated pick-up mechanism collects objects from a conveyor and fills each of the boxes in turn.



Additional Information:

Whist the palette size is fixed at the maximum size of 1.2m square, the program should be flexible enough to allow for a user-defined number of boxes on the palette. The grid could contain up to 10 divisions in each direction and they may be combined in any ratio, i.e. 2x4, 6x3, 1x10, 9x4 etc.

The illustration below shows a sample palette with a 6x6 grid of boxes. The numbers/arrows show the order in which the boxes are filled. Note that we step through the rows (Y axis) in turn, filling each box (move along X axis) before moving onto the next row.



Structuring the program

This example can be solved with a very simple structure using two nested `FOR..NEXT` loops.

Firstly we create a loop to step through each row (Y) in turn:

```
FOR y=0 to ydiv-1

NEXT y
```

'ydiv' is the number of rows, and the -1 is because we start counting at 0 rather than 1.

Now, within this loop we create another for the 'X' direction:

```
FOR y=0 to ydiv-1
  FOR x=0 to xdiv-1
    '
  NEXT x
NEXT y
```

Calculating the box positions

So now we have a sequence which steps sequentially through each row, and then through each position on that row in turn. We can use the absolute move (**MOVEABS**) command to position the axes at an absolute position in our X/Y coordinate system in the form

```
MOVEABS(x,y)
```

The x and y variables with the **FOR..NEXT** loop are simply logical box coordinates and therefore need to be scaled to the correct positions.

If we know the palette size (1200) and the number of divisions in each direction (xdiv/ydiv) then we can simply calculate an appropriate scaling, thus for the x axis:

```
xscale = 1200/xdiv
```

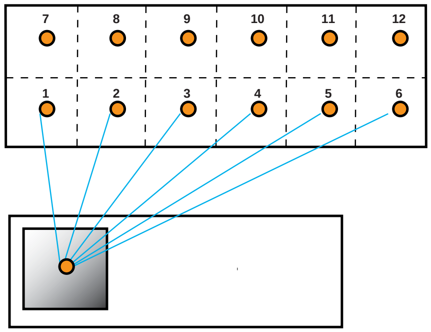
The actual position (of the box's corner) would therefore be (x*xscale), we could adjust this for the centre of the box by adding half the box size (xscale/2). So the position would be:

```
(x*xscale)+(xscale/2)
```

Our final consideration is that for each box we first have to move to the preset pick-up position, fetch the product and then move to the appropriate empty box to place the product in.

The pick up point is at a known absolute position and so we can simply use a pair of constants (pick_x and pick_y) to reference this point.

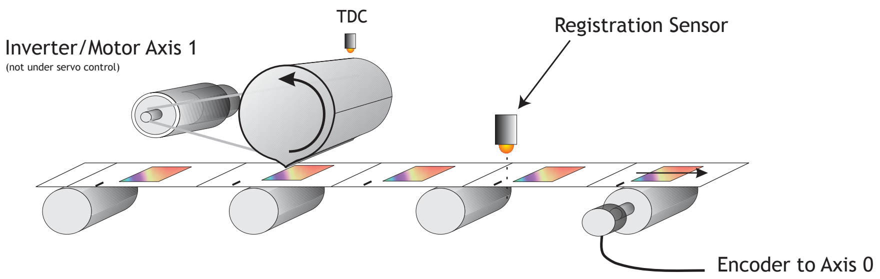
```
MOVEABS(pick_x, pick_y)
```



```
constants:
  nozzle=8 `    output - nozzle raise/lower
  vacuum=9 `    output - vacuum on / off

  xdiv=6
  ydiv=6

start:
  xscale=1200/xdiv
  xmid=xscale/2
  yscale=1200/ydiv
  ymid=yscale/2

  FOR y=0 TO xdiv-1
    FOR y=0 TO ydiv-1
    GOSUB pick
      MOVEABS((x*xscale)+xmid,(y*yscale)+ymid)
      WAIT IDLE
      GOSUB place
    NEXT x
  NEXT y
GOTO start

pick:
  MOVEABS(pick_x, pick_y)
  WAIT IDLE

  OP(nozzle,ON)
  OP(vacuum,ON)
  wa(500)
  OP(nozzle,OFF)
  wa(500)
RETURN

place:
  OP(nozzle,ON)
  OP(vacuum,OFF)
  wa(250)
  OP(nozzle,OFF)
  wa(500)
RETURN
```

Example 5 - Rotating Print Head with Registration

Description A rotating print head prints a number on a conveyor mounted product. During printing the print head must be synchronized with the conveyor. The print position must be registered to be relative to a registration mark.



The program achieves the motion profile by:

1) Making a synchronization gearbox connection between the conveyor and the print head with CONNECT. This will let the printer make a print on the conveyor. The distance between the prints will be the peripheral distance of the print head rotation.
2) Datuming and setting the repeat distance REPDIST for the print head rotation so that it has an absolute position of zero every time it touches the conveyor.
3) Superimposing a movement onto the print head. This movement has two functions: To adjust the printer position to keep the system in register and to adjust by the difference between the peripheral distance of the printer and the registration marks on the conveyor.
4) The superimposed movement is run on axis 3 of the controller (an “imaginary” axis) and the movement summed with the ADDAX command. The move is performed when the printer is going “over the top” of its stroke.

Program Listing

```
'  Rotating Head with Registration:
start: GOSUB initial

loop:
  WAIT UNTIL MPOS>100' Wait 100mm past print

  IF MARK THEN
    ’ mark seen during last cycle:
    ’ Limit adjust to 10mm
    r_adj=REG_POS*0.5' Apply 50% of error
    IF ABS(r_adj)>10 THEN r_adj=SGN(r_adj)*10
    OP(8,OFF)
  ELSE
    ’ mark not seen last cycle: Set Zero adjust
    r_adj=0
    OP(8,ON)' light "no register" warning lamp
  ENDIF

  BASE(3)' Correction on axis 3
  MOVELINK(75-r_adj,150,25,25,1)
  ‘ 75 is the diff. between the mark spacing and
  ‘ the print head circumference
  ' Move linked to conveyor
  WAIT IDLE
  BASE(0)
  REGIST(3)
GOTO loop

initial:
  BASE(0)'   Setup axis 0
  UNITS=20'  Edges/mm
  P_GAIN=0.5
  REP_DIST=200' 200mm=180 degrees
  SERVO=ON

  BASE(1)'  Setup axis 1
  SERVO=OFF
  UNITS=15'Edges/mm on conveyor
  BASE(3)' Setup axis 3
  UNITS=20' Match axis 0 units

  ' Datum axis 0 keeping sync with paper:
  WDOG=ON
  BASE(0)
  CONNECT(20/15,1)
  WAIT UNTIL IN(0)=ON' Wait for prox
  DEFPOS(-150)
  ADDAX(3)'  Add moves on axis 3
RETURN
```

Example 6 - *Motion Coordinator* programs sharing data

Description These two programs run multi-tasking on a *Motion Coordinator*. The Motion Cycle program performs a movement. The Operator Interface program communicates with a membrane keypad to control the Motion Cycle program. In this simple example of multi-tasking the two tasks communicate via two global variables.

`VR(start)`	This holds the start/stop signal
`VR(length)`	This holds the movement length

Operator Interface Program

```
' Motion Coordinator Demonstration Program

start: GOSUB initial

loop:
  PRINT #3,CHR(20);CHR(14);
  PRINT #3,CURSOR(0);">LENGTH:";in_length[0];
  IF VR(start)=ON THEN
    PRINT #3,CURSOR(15);"STOP<";
  ELSE
    PRINT #3,CURSOR(15);" RUN<";
  ENDIF

  GET #3,kp
  PRINT kp
  IF kp=53 THEN GOSUB input_length
  IF kp=54 THEN VR(start)=1-VR(start)
GOTO loop

input_length:
  PRINT #3,CURSOR(60);"New Length:";
  GOSUB getnum
  IF num>=smallest_len THEN
    in_length=num
  ELSE
    PRINT #3,CURSOR(60);"Min. Length=200mm";
    WA(1000)
  ENDIF
RETURN
getnum:
  pos=40
  num=0
  PRINT#4,CHR(20);
  REPEAT
    PRINT#4,CURSOR(pos);num[6,0];
    GET#4,k
    IF k=69 THEN GOTO getnum
      IF k>=59 AND k<=61 THEN k=k-7
      IF k>=66 AND k<=68 THEN k=k-17
      IF k=71 THEN k=48
      IF k>47 AND k<58 THEN
        k=k-48
        num=num*10+k
      ENDIF
  UNTIL k=73
RETURN

initial:
  PRINT #3,CHR(20);CHR(14);
  PRINT #3,CURSOR(0);
  PRINT #3," Demonstration of  "
  PRINT #3,"Motion Coordinator "
  WA(3000)

' Set Global Variable Pointers:
  start=0
  length=1

' Set any none zero local variables:
  in_length=VR(length)
  VR(start)=0
  RUN "cycle"
RETURN
```

Motion Cycle Program

```
'
' Motion Cycle demonstration program:
'
  GOSUB setvar
  GOSUB initial
```

```
loop:
  WAIT UNTIL VR(start)=ON
  MOVE(VR(length))
  MOVEABS(0)
GOTO loop

initial:
  WDOG=ON
  WA(100)

  BASE(0)
  P_GAIN=0.8
  FE_LIMIT=1000
  SERVO=ON
  ACCEL=1000000
  DECEL=100000
  SPEED=10000
RETURN

setvar:
  ' Define Global Variable Pointers:
  start=0
  length=1
RETURN
```

Example 7 - Handling Axis Errors

The *Motion Coordinator* controllers are designed to trap error conditions in hardware, and if required to automatically open the drive enable relay (watchdog) and to disable the output to the drives.

As this mechanism happens automatically, it may not be immediately apparent that an error has occurred and therefore we need a mechanism in the software to recognise it, and to set up the type of errors which will cause the controller to disable the drive / output.

The relevant parameters are:

- **AXISSTATUS**
- **ERRORMASK**
- **MOTION_ERROR**
- **ERROR_AXIS**

```
start:
  ‘ Monitor constantly until axis error occurs
  ‘ Set ERRORMASK so that Following Errors and Fwd/Rev limit
  ‘ switches will automatically trip the watchdog relay
  ERRORMASK = 256+16+32

REPEAT
  IF MOTION_ERROR<>0 THEN
    ax = ERROR_AXIS
    BASE(ax)
    PRINT #3,CURSOR(0);”Error on Axis “;ax[0]
    IF (AXISSTATUS AND 256)>0 THEN PRINT #3,”Fol. Error”;
    IF (AXISSTATUS AND (16+32))>0 THEN PRINT #3,”H/W Limit”;
    IF (AXISSTATUS AND 512+1024)>0 THEN PRINT #3,”S/W Limit”;
  ENDIF

  WAIT UNTIL KEY#3
  GET #3,k
GOTO start
```

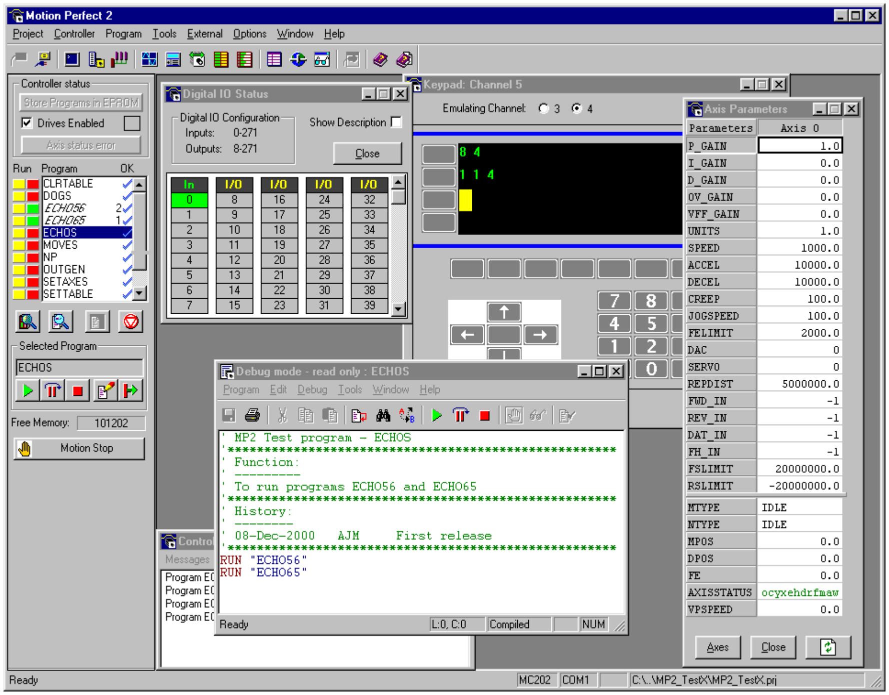
CHAPTER

10

SUPPORT SOFTWARE

Motion Perfect 2

Motion Perfect 2 is an application for the PC, designed to be used in conjunction with the *Motion Coordinator* range of multi tasking motion controllers.



Free Memory: 101202

Run	Program	OK
	CLRTABLE	✓
	DOGS	✓
	ECHO56	2 ✓
	ECHO65	1 ✓
	ECHOS	✓
	MOVES	✓
	NP	✓
	OUTGEN	✓
	SETAXES	✓
	SETTABLE	✓

Selected Program: ECHOS

Digital IO Configuration
Inputs: 0-271
Outputs: 8-271

In	I/O	I/O	I/O	I/O
0	8	16	24	32
1	9	17	25	33
2	10	18	26	34
3	11	19	27	35
4	12	20	28	36
5	13	21	29	37
6	14	22	30	38
7	15	23	31	39

Emulating Channel: 3 [✓ 4]

```
8 4
1 1 4
```

```
' MP2 Test program - ECHOS
'*****************************************************
' Function:
' ---------
' To run programs ECHO56 and ECHO65
'*****************************************************
' History:
' --------
' 08-Dec-2000   AJM     First release
'*****************************************************
RUN "ECHO56"
RUN "ECHO65"
```

Ready L:0, C:0 Compiled NUM

Program EC
Program EC
Program EC
Program EC

Parameters	Axis 0
P_GAIN	1.0
I_GAIN	0.0
D_GAIN	0.0
OV_GAIN	0.0
VFF_GAIN	0.0
UNITS	1.0
SPEED	1000.0
ACCEL	10000.0
DECEL	10000.0
CREEP	100.0
JOGSPEED	100.0
FELIMIT	2000.0
DAC	0
SERVO	0
REPDIST	5000000.0
FWD_IN	-1
REV_IN	-1
DAT_IN	-1
FH_IN	-1
FSLIMIT	20000000.0
RSLIMIT	-20000000.0
MTYPE	IDLE
NTYPE	IDLE
MPOS	0.0
DPOS	0.0
FE	0.0
AXISSTATUS	ocyxehdrfmaw
VPSPEED	0.0

Ready MC202 COM1 C:\...\MP2_TestX\MP2_TestX.prj

Motion Perfect provides the user with an easy to use Windows based interface for controller configuration, rapid application development, and run-time diagnostics of processes running on the *Motion Coordinator*.

System Requirements:

The following equipment is required to use *Motion* Perfect 2.

PC

	Minimum Specification	Recommended
CPU	Pentium class processor, operating at 75MHz	Pentium class processor, operating at 200MHz
RAM	16 Mb	32Mb RAM (64Mb for Windows 2000)
Hard disk space	10 Mb	10 Mb
	For Serial Operation: Windows 98 or Windows NT v4 **For USB/ETHERNET Communications:** Windows 98, Windows ME, or Windows 2000 **PCI Bus Communications:** Windows 2000/XP	Windows 98 SE, Windows ME, Windows 2000 or Windows XP.
Display	800 x 600 256 colours	1024 x 768 16-bit colour
Communications	Single RS232 Serial Port/ PCI slot for PCI 208	RS232 serial port, USB port
Other	Mouse or similar pointing device	

Motion Coordinator

- *Motion Coordinator* controller or compatible controllers.

  Compatible controllers include:
  MC2, MC202, MC204, MC402e, Euro 205, MC206, MC216, MC224 and PCI 208

In order to use the Packet Communications mode, system software version 1.49 or higher is required.

Note: *You should always try to use the most recent version of* ***Motion Perfect****. Updates are available from your local distributor or you can download the latest version from the Trio Web site:* **WWW.TRIOMOTION.COM**

New Features for *Motion* Perfect 1 Users

If you have previously used the first version of *Motion* Perfect you will find version 2 very familiar. We have maintained a core functionality and working environment as close to the original as possible whilst adding a number of new and enhanced features.

New in Version 2

- Full Windows 32 bit Application
- Designed for 32 Bit Windows (Windows 95,98, ME, and Windows 2000)
- Full Support for Latest Trio Products
- Supports controller locking for project security
- Supports Feature Enable Codes for the Euro 205 and MC206
- New Editor Features:
  Syntax highlighting
  Error highlighting
  Offline program editing
  Windows restored after connection
- Links to External Applications
- Integrated support for CAD2Motion, DocMaker
- Enhanced Communications Features
  Protected RS-232 communications mode
  USB support for high speed communications
  Motion Coordinator simulation
- New/Enhanced Tools
- TABLE Editor
- Variable Editor
- Load / Save TABLE files
- Resizable Oscilloscope

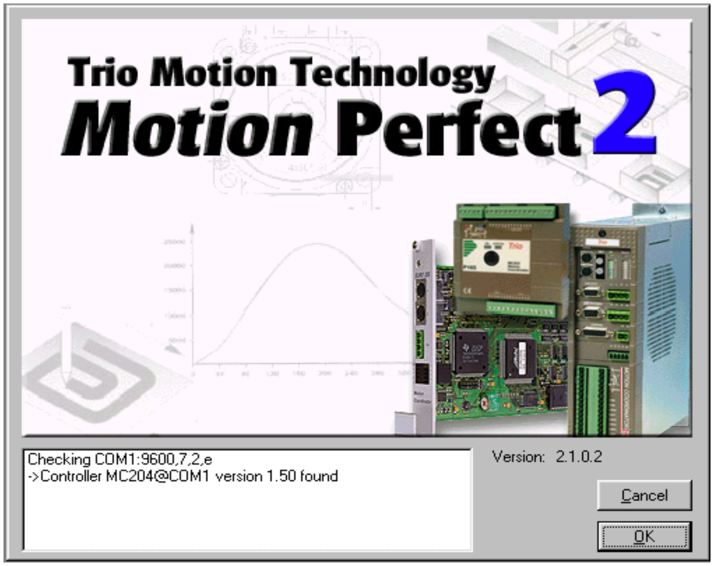
Connecting *Motion* Perfect to a controller

Motion Perfect can be connected to the *Motion Coordinator* using a serial connection, USB, Ethernet, or via the PCI bus if using the PCI 208. A suitable serial cable can be supplied by Trio Motion Technology. *Motion* Perfect may use either of the standard serial ports on a PC, COM1 or COM2, but it will connect quicker to your system if COM1 is used as this is the first port it tries. The communication method(s) can be selected under the menu Options->Communications

If you wish to edit a project but do not have a controller connected to your PC, it is possible to edit off-line by connecting to a ‘virtual’ controller running on your PC. See the information on the MCSimulation at the end of this chapter.

Running *Motion* Perfect 2 for the First time

Turn your PC on and enter Windows. Make sure the *Motion Coordinator* is turned on and then launch *Motion* Perfect. During initialisation you will see a splash screen such as the one below.



```
Checking COM1:9600,7,2,e
->Controller MC204@COM1 version 1.50 found
```

Figure 1: Splash screen

The splash screen features a small messages window (bottom left) which is used to display the status of the connection process. In this example *Motion* Perfect is connected to an MC204 controller via serial port COM1.

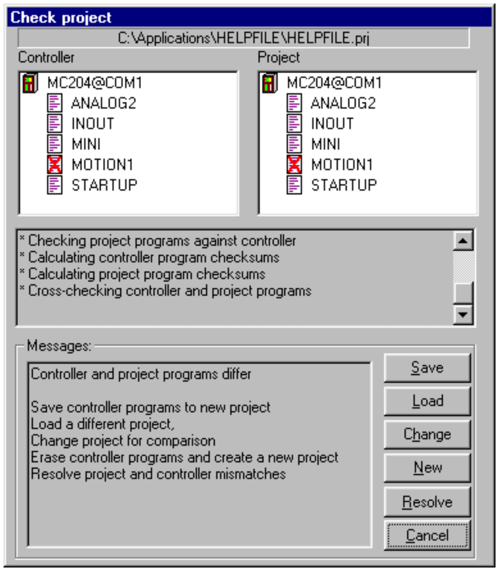
Motion Perfect 2 Projects

One of the keys to using *Motion* Perfect is to understand its concept of a "Project". The project facilitates the application design and development process, by providing a disk based copy of the multiple controller programs, parameters and data which may be used for a single motion application. Once the user has defined a project, *Motion* Perfect works behind the scenes automatically maintaining consistency between the programs on the controller and the files on the PC. When creating or editing programs on the controller they are automatically duplicated on the PC which means you do not have to worry about loading or saving programs and you can be confident that next time you connect to the controller you will have the correct information on your PC .

The Project Check Window

Whenever you connect to the controller, *Motion* Perfect will perform a **project check** to compare the programs on the controller with those defined in the current project on the pc. During the project check a window similar to the one below will be displayed. If the projects match then you will see a "project checked ok" message and an OK button to continue. If however there is any inconsistency between the controller and the PC, the display will feature a number of addition options, shown below.

You can force *Motion* Perfect to perform a project check at any time with the "Check Project" option from the project menu. (Ctrl+Alt+P)



* Checking project programs against controller
* Calculating controller program checksums
* Calculating project program checksums
* Cross-checking controller and project programs

Controller and project programs differ

Save controller programs to new project
Load a different project,
Change project for comparison
Erase controller programs and create a new project
Resolve project and controller mismatches

Project Check Options

`Save` **Save the controller contents to disk.**

If you have never connected with this controller before, and therefore do not have the project on your PC, or if there in an inconsistency in the project check and you are sure that the project on the controller is the correct version, then select SAVE to copy the programs on the controller to disk.

Note: This will of course overwrite any programs already in the PC copy of the project. If you are unsure which is the correct versiom, you should save the project with a new name to avoid overwriting any existing project programs on the PC.

`Load` **Load the PC files onto the controller**

If you are uploading a complete project from the PC to the controller, or the project check fails and you are sure the version on your PC is correct, then you should use this option to upload the entire project from the PC to the controller.

Note: The entire contents of the programs on the controller will be erased. If you are unsure, SAVE the controller contents first!

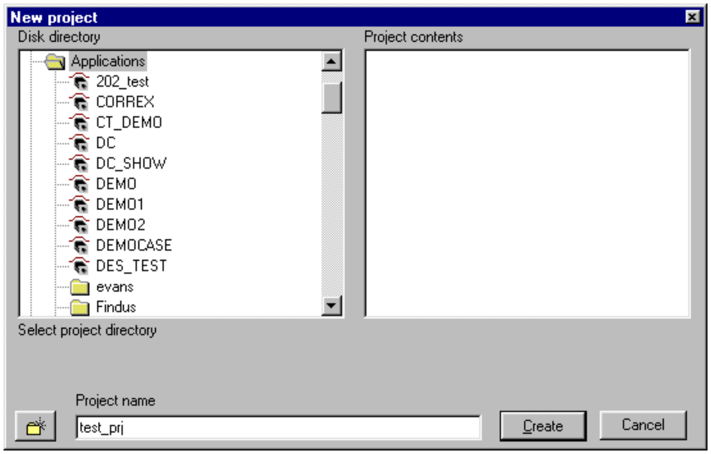
`Change` **Change the project on the PC to compare with..**

If you have been working on more than one project, the project on the controller may not match the 'last project' remembered by *Motion* Perfect. If this is the case you can use this option to select another project on the PC. Once you select an alternative, *Motion* Perfect will perform a fresh project check and the above process will be repeated.

New **Create a new project**

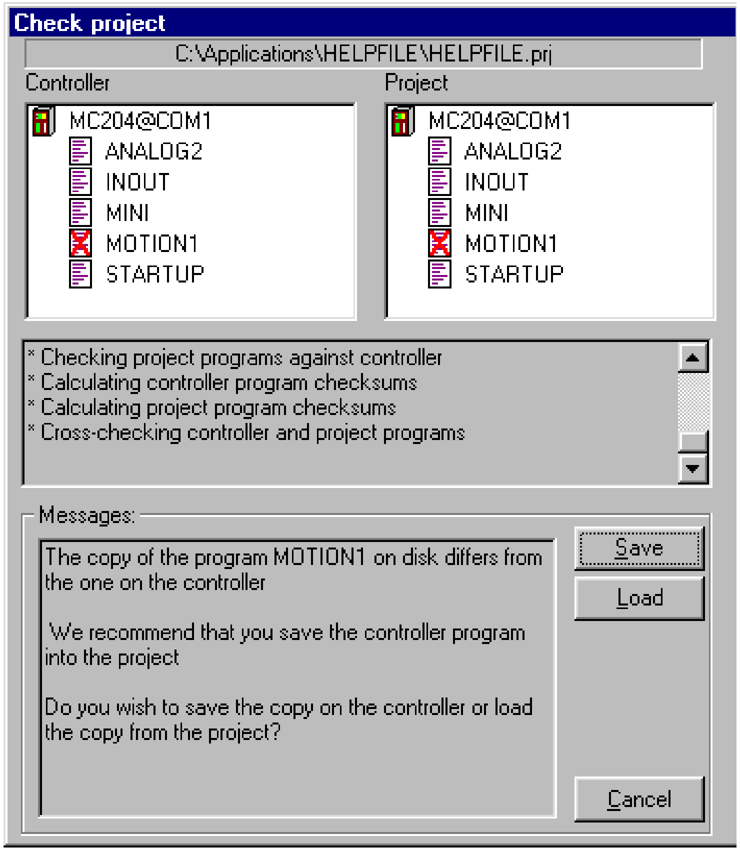
The controller contents will be erased and a new project created on the PC. You will be prompted to select a directory and project name.

When you create a new project, *Motion* Perfect will make a new directory with the project name, and within that directory a project file with the same name (the .PRJ extension is added to the filename).



Project name: test_prj

`Resolve` This option should be used when you have the correct project selected, but one or more of the files differ between the controller and PC version, or do not exist in one of the copies.



Controller	Project
MC204@COM1	MC204@COM1
ANALOG2	ANALOG2
INOUT	INOUT
MINI	MINI
MOTION1	MOTION1
STARTUP	STARTUP

```
* Checking project programs against controller
* Calculating controller program checksums
* Calculating project program checksums
* Cross-checking controller and project programs
```

The copy of the program MOTION1 on disk differs from the one on the controller

We recommend that you save the controller program into the project

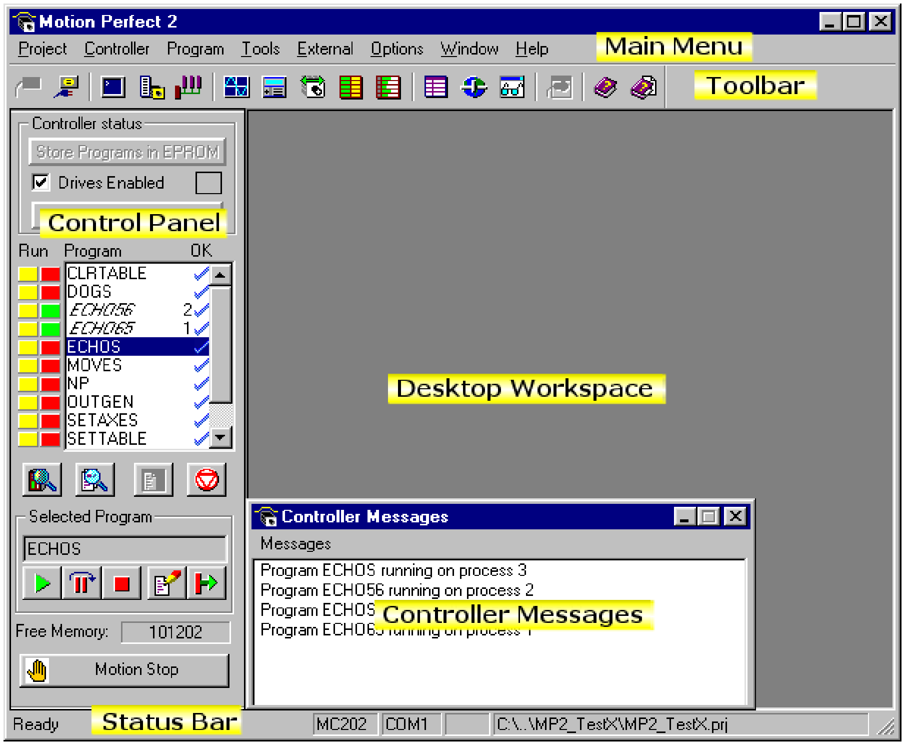
Do you wish to save the copy on the controller or load the copy from the project?

You will need to use your judgment to decide whether the disk or controller version is correct. Typically, if you are recovering the project after a comms failure or PC crash then the version on the controller should be saved. If you have modified the disk based copy of the program then you will need to load this version onto the controller.

`Cancel` Cancels the connection process and starts *Motion* Perfect in disconnected mode.

Once the project has been checked and is consistent then a backup copy of the PC project will be created.

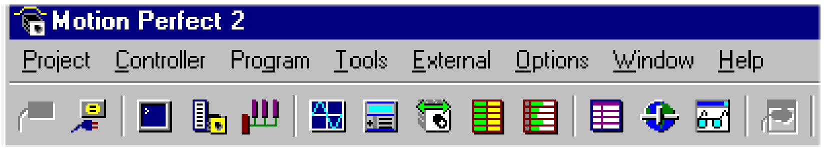
The *Motion* Perfect Desktop



Program ECHOS running on process 3
Program ECHO56 running on process 2
Program ECHOS
Program ECHO65 running on process 1

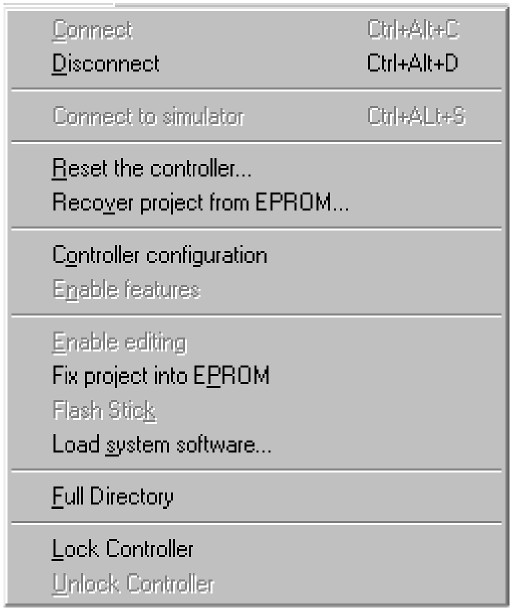
Main Menu	Standard Windows menu to access all features of the *Motion* Perfect application.
Toolbar	Shortcut buttons to access the *Motion* Perfect tools
Control Panel	Displays the current controller contents and provides controls for interrogating the controller status, running / editing programs
Desktop Workspace	This area is used to display the user windows and tools
Controller Messages	Status and error messages reported by the controller
Status Bar	Information about the current project and controller connection.

Main Menu


Motion Perfect 2
Project Controller Program Tools External Options Window Help

`Project`	Options for Creating, Loading & Saving *Motion* Perfect Projects, Loading/Saving program files and Table data
`Controller`	Options relating to the controller hardware, including connecting/disconnecting and checking configuration information.
`Program`	Program specific options, including creating, editing and running controller tasks.
`Tools`	Access to the main *Motion* Perfect tools. These options are also available from the Toolbar
`Options`	Configure the *Motion* Perfect Environment. Includes options to setup the communications ports and to customise the editor display.
`Window`	Control the appearance of the *Motion* Perfect desktop.
`Help`	Access the help files and version information.

Controller Menu

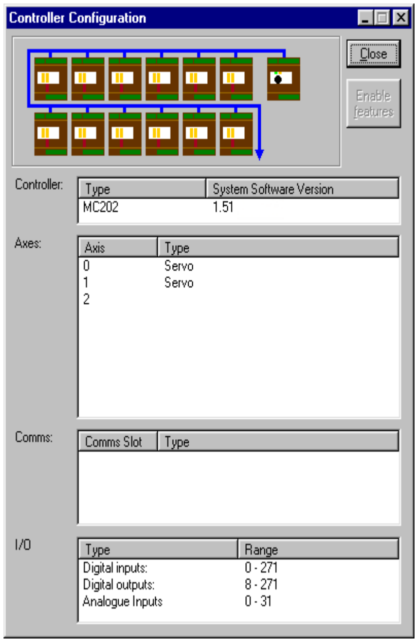


The controller menu contains the following items:

Connect	Connect using serial communications to the controller and start the project manager. This is only available if *Motion* Perfect is currently disconnected from the controller.
Disconnect	Disconnect the serial communications, and stop using the project tools. Only available if *Motion* Perfect 2 is currently connected to the controller.
Reset Controller	Perform a software-reset (EX) on the controller. This will cause *Motion* Perfect 2 to disconnect from the controller
Recover Project from EPROM	Reset the controller and restore the programs which were previously stored in the EPROM

Controller Configuration

This screen interrogates the hardware and displays the configuration information reported back by the controller.



Controller:

Type	System Software Version
MC202	1.51

Axes:

Axis	Type
0	Servo
1	Servo
2	

Comms:

Comms Slot	Type

I/O

Type	Range
Digital inputs:	0 - 271
Digital outputs:	8 - 271
Analogue Inputs	0 - 31

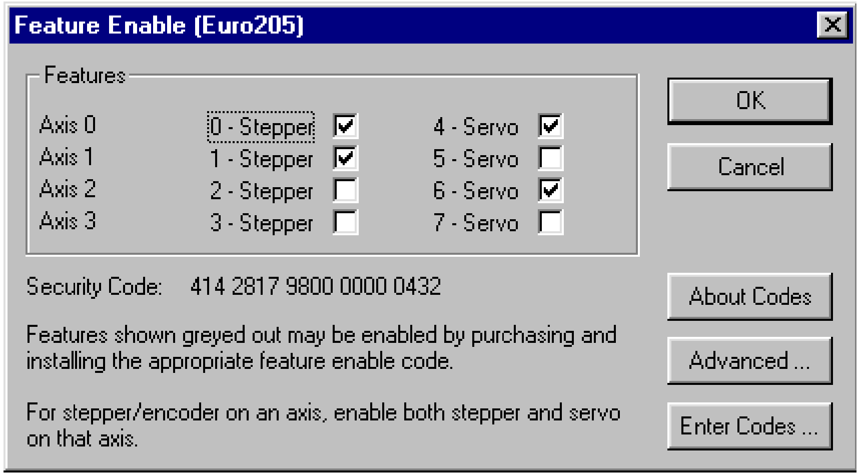
Looking at the example screen shown here from top to bottom:

Controller:	We are connected to a *Motion Coordinator* MC202
Software Version:	The controller is running version 1.51 of the system software.
Axis Types	A list of the types of all available axes.
Comms Boards	If the controller is fitted with any of the extended / communications daughter boards, that capability will be indicated here.
I/O	The channel range available for each type of I/O. Remember that on many *Motion Coordinator*s the channels are shared, i.e. if Output 15 is available, then it implies that Input 15 is also available and shares the same connector.

Feature Enable

Certain *Motion Coordinators*, such as the EURO205 and MC206, have the ability to unlock additional axes by entering a "Feature Enable Code".

When you access the Feature Enable dialog, you will be presented with a display similar to the following:


Feature Enable (Euro205)

Features

Axis 0	0 - Stepper	✓	4 - Servo	✓
Axis 1	1 - Stepper	✓	5 - Servo	☐
Axis 2	2 - Stepper	☐	6 - Servo	✓
Axis 3	3 - Stepper	☐	7 - Servo	☐

Security Code: 414 2817 9800 0000 0432

Features shown greyed out may be enabled by purchasing and installing the appropriate feature enable code.

For stepper/encoder on an axis, enable both stepper and servo on that axis.

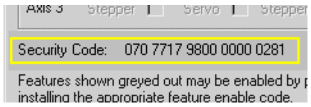
This display illustrates the feature which are currently available. If the codes for additional features have been purchased, the relevant boxes will be available for checking.

Enabling Additional Features

To enable a feature you must enter a Feature Enable Code, which is unique to each controller and feature.   To obtain a Feature Enable Code, you will need to specify the feature required and the security code for the specific controller to be updated. The order for the required codes should be FAXed to Trio or an authorised Trio distributor.

Security Code

Controllers with features which can be enabled each have a unique security code number which is implanted when the unit is manufactured. This security code number is displayed on the above screen (as highlighted right).

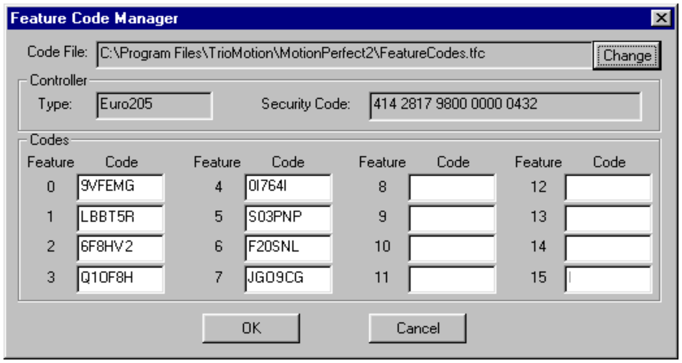

Axis 3 Stepper Servo Stepper

Security Code: 070 7717 9800 0000 0281

Features shown greyed out may be enabled by
installing the appropriate feature enable code.

Once you have the required codes, select the button.

A dialog similar to the following example will appear.



Feature Code Manager

Code File: C:\Program Files\TrioMotion\MotionPerfect2\FeatureCodes.tfc

Controller

Type: Euro205 — Security Code: 414 2817 9800 0000 0432

Codes

Feature	Code	Feature	Code	Feature	Code	Feature	Code
0	9VFEMG	4	0I764I	8		12	
1	LBBT5R	5	S03PNP	9		13	
2	6F8HV2	6	F20SNL	10		14	
3	Q10F8H	7	JG09CG	11		15	

Each feature requested has a feature number. Enter the relevant code for each feature number, being careful to 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 the letter I.

Feature Code File

Motion Perfect stores all of the Feature Enable Codes of which it is aware in a file called "`FeatureCodes.TFC`". By default this file is located in the same directory as the *Motion* Perfect 2 executable file.

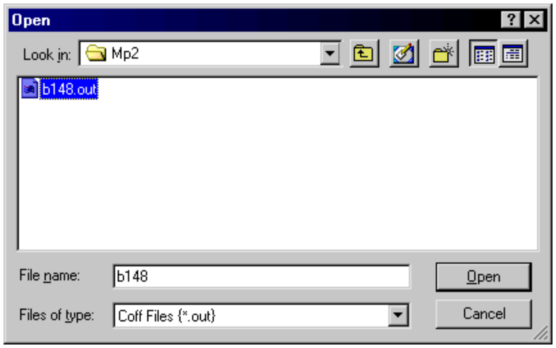
`Enable Editing`	Restore the power-up state of a controller currently starting from EPROM to run from RAM and allow editing.
`Fix Project into EPROM`	Store the programs in RAM into the controllers flash-eprom memory. The startup state for each program will not be changed.
`Full Directory`	Display a complete listing of all files on the controller, details of memory used and the run status of each program.

Loading New System Software

*Motion Coordinator*s feature a flash eprom for storage of both user programs and the system software. From *Motion* Perfect 2 it is possible to upgrade the software to a newer version using a system file supplied by Trio.

NOTE: ***We do not advise that you load a new version of the system software unless you are specifically advised to do so by your distributor or by Trio.***

When you select the 'Load System Software' option from the controller menu, you will first be presented with a warning dialog to ensure you have saved your project and are sure you wish to continue. When you press OK you will be presented with the standard Windows file selector to choose the file you wish to load.



Each *Motion Coordinator* controller has its own system file, identified by the first letter of the file name.

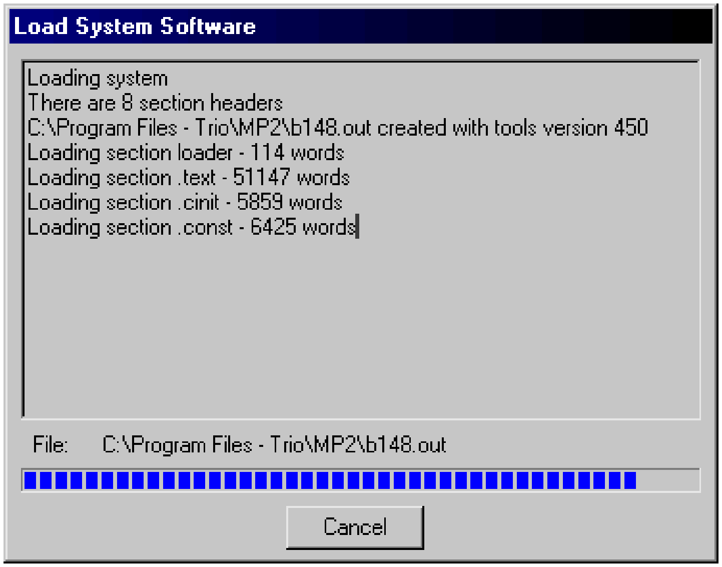
System Software File Prefix Codes:

Filename	Controller Type
`Annn.OUT`	MC2
`Bnnn.OUT`	MC204
`Cnnn.OUT`	MC216
`Dnnn.OUT`	MC402
`Ennn.OUT`	Euro205
`Fnnn.OUT`	MC202 / Motion Module
`Gnnn.OUT`	MCW202
`Hnnn.OUT`	MC402e
`Jnnn.OUT`	MC206

You must ensure that you load only software designed for your specific controller, other versions will not work

When you have chosen the appropriate file you will be prompted once again to check that you wish to continue. Press OK to start the download process.

Downloading will take several minutes, depending on the speed of your PC. During the download, you should see the progress of each section updated as follows:-



```
Loading system
There are 8 section headers
C:\Program Files - Trio\MP2\b148.out created with tools version 450
Loading section loader - 114 words
Loading section .text - 51147 words
Loading section .cinit - 5859 words
Loading section .const - 6425 words
```

File: C:\Program Files - Trio\MP2\b148.out

When the download is complete, a checksum is performed to ensure that the download process was successful. If it saw you will be presented with a confirmation screen and asked if you wish to store the software into Eprom.

When you press Yes, the controller will take a few moments to fix the project into the eprom and you can then continue as normal.

At this point you can check the controller configuration to confirm the new software version.

Flashstick support

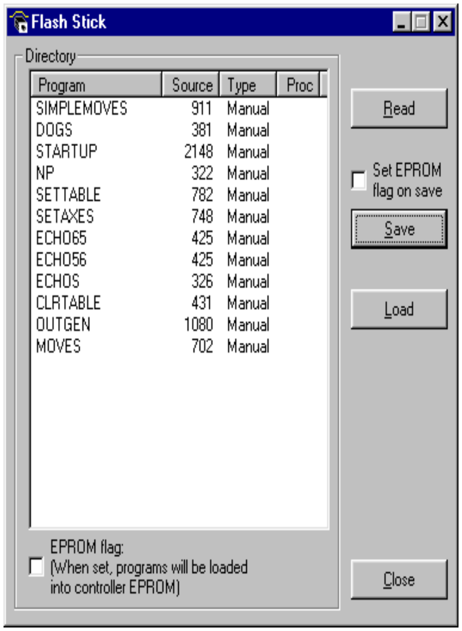
This only applies to controllers which are fitted with a flashstick socket (eg MC206).

When a controller with flashstick support is powered on with a flashstick inserted, then the controller will automatically load the programs from the flashstick into the controller RAM.

The **Read** button will read the directory of the flashstick and display it. As the directory is automatically read when the tool window is created, this button only needs to be used when the flashstick is changed.

The **Save** button will save the programs on the controller to the flashstick eassing any programs already stored on the flashstick. If the **set EPROM fag on save** box is checked then a flag is set on the flashstick which makes the controller store the programs on the flashstick in controller EPROM as well as in controller RAM at power-up.

The **Load** button will load the programs from the flashstick onto the controller. This is done by resetting the controller.

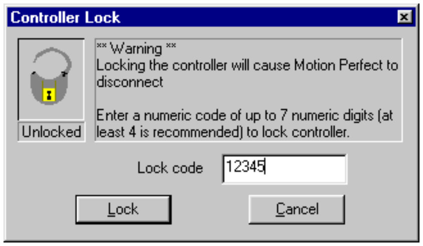


Program	Source	Type	Proc
SIMPLEMOVES	911	Manual	
DOGS	381	Manual	
STARTUP	2148	Manual	
NP	322	Manual	
SETTABLE	782	Manual	
SETAXES	748	Manual	
ECHO65	425	Manual	
ECHO56	425	Manual	
ECHOS	326	Manual	
CLRTABLE	431	Manual	
OUTGEN	1080	Manual	
MOVES	702	Manual	

EPROM flag:
(When set, programs will be loaded into controller EPROM)

Lock / Unlock

Lock Controller


Controller Lock

** Warning **
Locking the controller will cause Motion Perfect to disconnect

Enter a numeric code of up to 7 numeric digits (at least 4 is recommended) to lock controller.

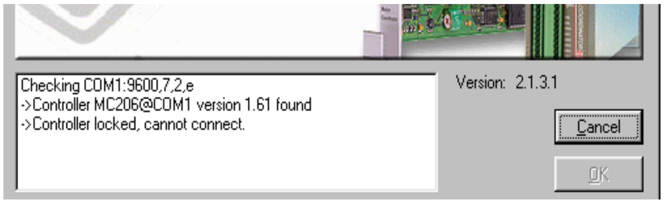
Lock code 12345

Locking the controller will prevent any unauthorised user from viewing or modifying the programs in memory.

You simply need to enter a numeric code (up to 7 digits). This value will be encoded by the system and used to lock the directory structure. The lock code is held in encrypted form in the flash memory of the *Motion Coordinator*.

This can also be achieved by issuing the LOCK(lock_code) command from the controller's command line.

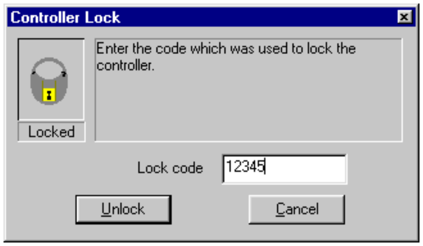
Once the *Motion Coordinator* is locked it is not possible to list, edit or save any of the controller programs. You cannot connect to the controller with *Motion* Perfect 2, although the terminal screen and unlock dialog will still be available.


```
Checking COM1:9600,7,2,e
->Controller MC206@COM1 version 1.61 found
->Controller locked, cannot connect.
```

Version: 2.1.3.1

WARNING: ***If you forget the lock code there is no way to unlock the controller. You will need to return it to Trio or a distributor to have the lock removed.***

Unlock Controller


Controller Lock

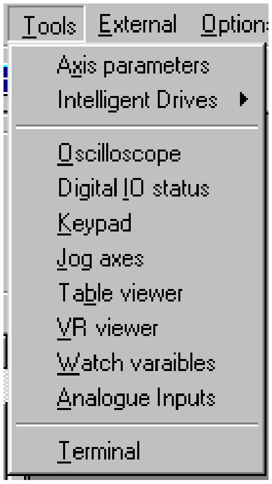
Enter the code which was used to lock the controller.

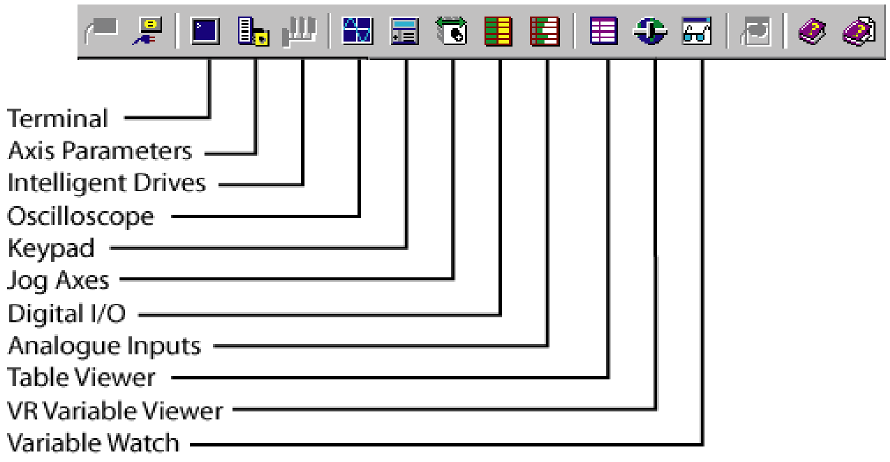
Lock code 12345

In order to unlock the controller you need to enter the same numeric code which was used to lock it. Once the unlock code is entered it will be possible to gain full access to the programs in memory.

Motion Perfect Tools

The *Motion* Perfect tools can be accessed from either the Tools Menu or the Toolbar buttons



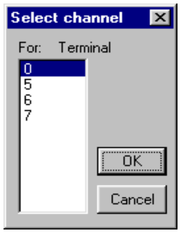


Terminal

The terminal window provides a direct connection to the *Motion Coordinator*. Most of the functions that must be performed during the installation, programming and commissioning of a system with a *Motion Coordinator* have been automated by the options available in the *Motion* Perfect menu options. However, if direct intervention is required the terminal window may be used.

Selecting a communications channel

If *Motion* Perfect is connected to the controller, you will be prompted to select from any of the active user communications channels:

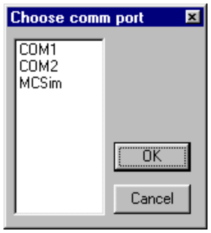


For this example we will select channel 0 which is used for the *Motion Coordinator* Command Line interface.

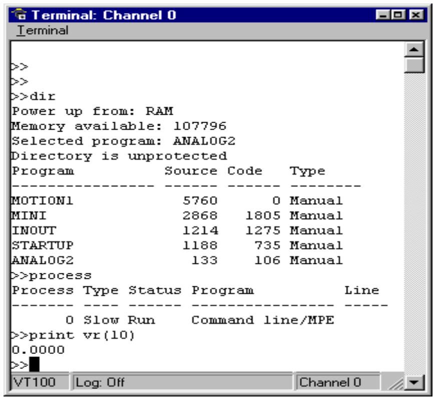
Using the terminal without a *Motion* Perfect Connection

If *Motion* Perfect is not connected to the controller, then the terminal window may be used to talk to the controller in a “No *Motion* Perfect connection”. In this mode the terminal window has complete control over the serial link.

Once connected to the controller, the terminal window opens a communications channel to the controller. The user must select the required channel.

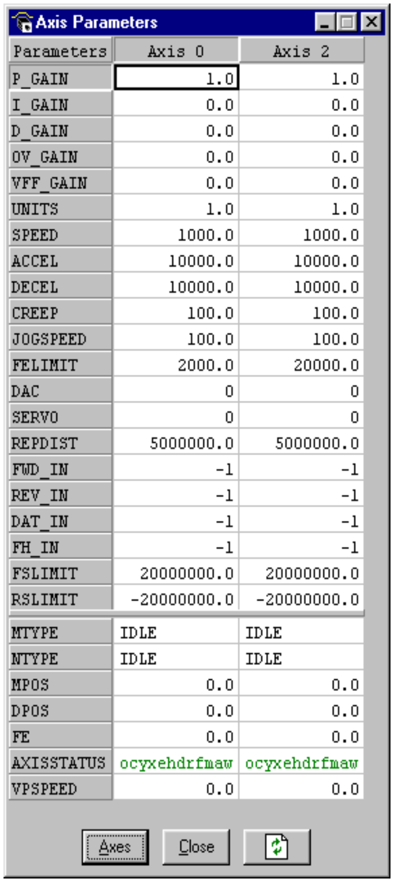


If this channel is already in use, either by another terminal window or a keypad emulation then access will be denied.



```
>>
>>
>>dir
Power up from: RAM
Memory available: 107796
Selected program: ANALOG2
Directory is unprotected
Program          Source Code   Type
---------------- ------ ------ --------
MOTION1            5760      0 Manual
MINI               2868   1805 Manual
INOUT              1214   1275 Manual
STARTUP            1188    735 Manual
ANALOG2             133    106 Manual
>>process
Process Type Status Program          Line
-------- ---- ------ ---------------- -----
      0 Slow Run    Command line/MPE
>>print vr(10)
0.0000
>>
```

Axis Parameters



Parameters	Axis 0	Axis 2
P_GAIN	1.0	1.0
I_GAIN	0.0	0.0
D_GAIN	0.0	0.0
OV_GAIN	0.0	0.0
VFF_GAIN	0.0	0.0
UNITS	1.0	1.0
SPEED	1000.0	1000.0
ACCEL	10000.0	10000.0
DECEL	10000.0	10000.0
CREEP	100.0	100.0
JOGSPEED	100.0	100.0
FELIMIT	2000.0	20000.0
DAC	0	0
SERVO	0	0
REPDIST	5000000.0	5000000.0
FWD_IN	-1	-1
REV_IN	-1	-1
DAT_IN	-1	-1
FH_IN	-1	-1
FSLIMIT	20000000.0	20000000.0
RSLIMIT	-20000000.0	-20000000.0
MTYPE	IDLE	IDLE
NTYPE	IDLE	IDLE
MPOS	0.0	0.0
DPOS	0.0	0.0
FE	0.0	0.0
AXISSTATUS	ocyxehdrfmaw	ocyxehdrfmaw
VPSPEED	0.0	0.0

The Axis Parameters window enables you to monitor and change the motion parameters for any axis on the controller.

The window is made up of a number of cells, separated into two banks, bank 1 at the top and bank 2 at the bottom:

Bank 1. contains the values of parameters that may be changed by the user.

Bank 2. contains the values of parameters that cannot be changed by the user, as these values are set by the system software of the *Motion Coordinator* as it processes the Trio BASIC motion commands and monitors the status of the external inputs.

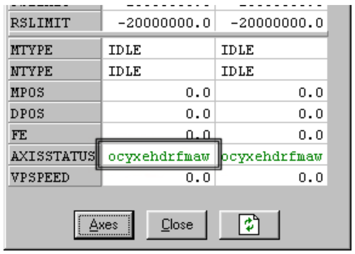
The black dividing bar that separates the two banks may be repositioned using the mouse to redistribute the space occupied by the different banks, for example to allow the user to shrink the window and view other windows whilst still watching the bank 2 information.

When there are more parameters in a bank that can be shown in the window a scroll bar will appear beside that bank so that the user can scroll up and down the parameter list to see the required values.

The user can select different parameters using the cursor keys or using the mouse. Multiple items may be selected by pressing the shift key and then using the cursor keys or the clicking the mouse to select a different cell, or by pressing the left mouse button in the start cell and the moving the mouse to select the last cell in the selection. Functions may be implemented in the future that work on a selection of multiples cells.

When the user changes the **UNITS** parameter for any axis, all the data for this axis is re-read as many of the parameters, such as the **SPEED**, **ACCEL**, **MPOS**, etc., are adjusted by this factor to be shown in user units.

In the *Motion* Perfect parameter screen the **AXISSTATUS** parameter is displayed as a series of characters, **ocyxehdrfmaw**.

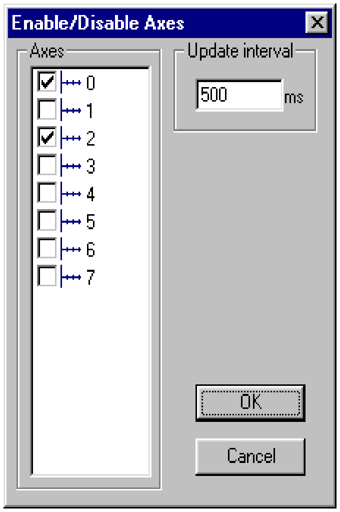


RSLIMIT	-20000000.0	-20000000.0
MTYPE	IDLE	IDLE
NTYPE	IDLE	IDLE
MPOS	0.0	0.0
DPOS	0.0	0.0
FE	0.0	0.0
AXISSTATUS	ocyxehdrfmaw	ocyxehdrfmaw
VPSPEED	0.0	0.0

These characters represent **AXISSTATUS** bits in order, as follows:-

char	status bit
w	Warning FE Range
a	Drive Comms Error
m	Remote Drive Error
f	Forward Limit
r	Reverse Limit
d	Datum Input
h	Feed Hold Input
e	Following Error
x	Forward Soft Limit
y	Reverse Soft Limit
c	Cancelling Move
o	Encoder Overcurrent

Parameter Screen Options



Enable/Disable Axes

Axes: [✓ 0] 1 [✓ 2] 3 4 5 6 7

Update interval: 500 ms

Select Axes

This shows a dialog that allows the user to select the axes for which the data will be displayed.

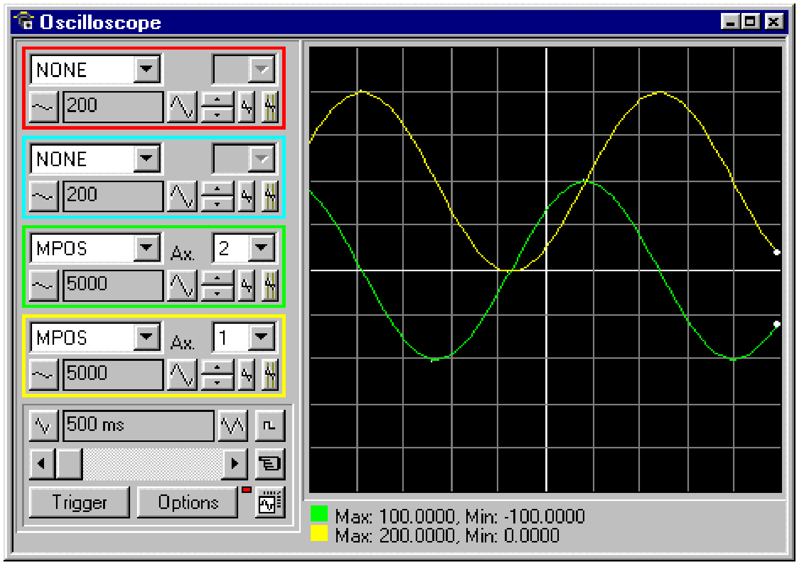
The axes set by the last Create Startup, Jog Axes window or Axes Parameters window will be displayed by default.

Refresh Display

In order to minimise the load placed upon the controller communications, the parameters in the bank 1 section are only read when the screen is first displayed or the parameter is edited by the user. It is possible that if a parameter is changed in a user program then value displayed may be incorrect. The refresh button will force *Motion* Perfect to read the whole selection again.

Note *If there is any possibility that a program has changed any of the parameters then you should ensure that your refresh the display before making changes.*

Oscilloscope



🟩 Max: 100.0000, Min: -100.0000
🟨 Max: 200.0000, Min: 0.0000

The software oscilloscope can be used to trace axis and motion parameters, aiding program development and machine commissioning.

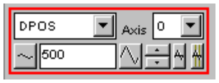
There are four channels, each capable of recording at up to 1000 samples/sec, with manual cycling or program linked triggering.

The controller records the data at the selected frequency, and then uploads the information to the oscilloscope to be displayed. If a larger time base value is used, the data is retrieved in sections, and the trace is seen to be plotted in sections across the display. Exactly when the controller starts to record the required data depends upon whether it is in manual or program trigger mode. In program mode, it starts to record data when it encounters a **TRIGGER** instruction in a program running on the controller. However, in manual mode it starts recording data immediately.

Controls

The oscilloscope controls are organised into four channel specific control blocks, followed by the oscilloscope general controls.

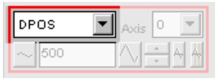
Oscilloscope Channel Controls



Each oscilloscope channel has the following channel specific controls organised in each of four 'channel control blocks' surrounded by a coloured border which indicates the colour of this channels trace on the display.

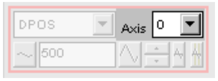
There are parameter list box / axis list box / vertical scale up-down buttons/ vertical offset scrollbar/ vertical offset reset button and cursor bars on-off button controls per scope channel.

Parameter



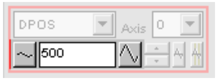
The parameters which the oscilloscope can record and display are selected using the pull-down list box in the upper left hand corner of each channel control block. Depending upon the parameter chosen, the next label switches between 'axis' or 'ch' (channel). This leads to the second pull-down list box which enables the user to select the required axis for a motion parameter, or channel for a digital input/output or analog input parameter. It is also possible to plot the points held in the controller table directly, by selecting the 'TABLE' parameter, followed by the number of a channel whose first/last points have been configured using the advanced options dialog. If the channel is not required then 'NONE' should be selected in the parameter list box.

Axis / Channel Number



A pull-down list box which enables the user to select the required axis for a motion parameter, or channel for a digital input/output or analog input parameter. The list box label switches between being blank if the oscilloscope channel is not in use, 'axis' if an axis parameter has been selected, or 'ch' if a channel parameter has been selected.

Vertical Scaling



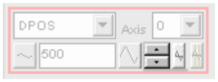
The vertical scale (units per grid division on the display) are selected per channel, and these can be configured in either automatic or manual mode.

In automatic mode the oscilloscope calculates the most appropriate scale when it has finished running, prior to displaying the trace. Hence if the oscilloscope is running with continuous triggering, it will initially be unable to select a suitable vertical scale. It must be halted and re-started, or used in the manual scaling mode.

In manual mode the user selects the scale per grid division.

The vertical scale is changed by pressing the up/down scale buttons either side of the current scale text box (left hand side button decreases the scale, and the right hand side button increases the scale value.) To return to the automatic scaling mode, continue pressing the left hand side button (decreasing the scale value) until the word 'AUTO' appears in the current scale text box.

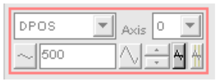
Channel Trace Vertical Offset



The vertical offset buttons are used to move a trace vertically on the display. This control is of particular use when two or more traces are identical, in which case they overlay each other and only the uppermost trace will be seen on the display.

The offset value remains for a channel until the vertical offset reset button is pressed, or the scrollbar is used to return the trace to its original position.

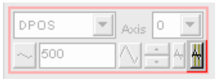
Vertical Offset Reset



The vertical offset value applied using the vertical offset scroll bars can be cleared by pressing this vertical offset reset button.

The button latches on/off. When the button is latched ON then the vertical offset will automatically rescale each time the oscilloscope display is redrawn.

Cursor Bars



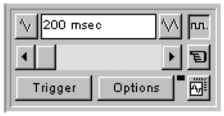
After the oscilloscope has finished running, and has displayed a trace, cursor bars can be enabled. These are displayed as two vertical bars, of the same colour as the channel trace, and initially located at the maximum and minimum trace location points. The values these represent are shown below the oscilloscope display, and again the text is of the same colour as the channel values represented.

The cursor bars are enabled/disabled by pressing the cursor button which toggles alternately displaying and removing the cursor bars. The bars can then be moved by positioning the mouse cursor over the required bar, holding down the left mouse button, and dragging the bar to the required position. The respective maximum or minimum value shown below the display is updated as the bar is dragged along with the value of the trace at the current bar position.

When the cursor bars are disabled, the maximum and minimum points are indicated by a single white pixel on the trace.

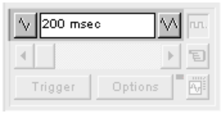
Oscilloscope General Controls

The oscilloscope general controls appear at the bottom left of the oscilloscope window. From here you can control such aspect as the time base, triggering modes and memory used for the captured data.



Time Base

The required time base is selected using the up/down scale buttons either side of the current time base scale text box (left side button decreases the scale, and the right side button increases the scale value.) The value selected is the time per grid division on the display.

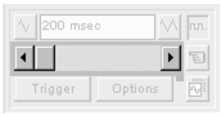


If the time base is greater than a predefined value, then the data is retrieved from the controller in sections (as opposed to retrieving a compete trace of data at one time.) These sections of data are plotted on the display as they are received, and the last point plotted is seen as a white spot.

After the oscilloscope has finished running and a trace has been displayed, the time base scale may be changed to view the trace with respect to different horizontal time scales. If the time base scale is reduced, a section of the trace can be viewed in greater detail, with access provided to the complete trace by moving the horizontal scrollbar.

Horizontal scrollbar

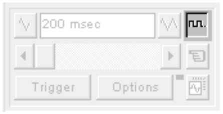
Once the oscilloscope has finished running and displayed the trace of the recorded data, if the time base is changed to a faster value, only part of the trace is displayed. The remainder can be viewed by moving the thumb box on the horizontal scrollbar.



Additionally, if the oscilloscope is configured to record both motion parameters and plot table data, then the number of points plotted across the display can be determined by the motion parameter. If there are additional table points not visible, these can be brought into view by scrolling the table trace using the horizontal scrollbar. The motion parameter trace does not move.

One Shot / Continuous Trigger Mode

Button Raised = One Shot Trigger Mode:

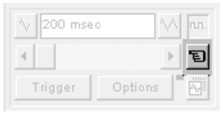


In one-shot mode, the oscilloscope runs until it has been triggered and one set of data recorded by the controller, retrieved and displayed.

Button Pressed = Continuous Trigger Mode:

In continuous mode the oscilloscope continues running and retrieving data from the controller each time it is re-triggered and new data is recorded. The oscilloscope continues to run until the trigger button is pressed for a second time.

Manual/Program Trigger Mode



The manual/program trigger mode button toggles between these two modes. When pressed, the oscilloscope is set to trigger in the program mode, and two program listings can be seen on the button. When raised, the oscilloscope is set to the manual trigger mode, and a pointing hand can be seen on the button.

Button Raised = Manual Trigger Mode:

In manual mode, the controller is triggered, and starts to record data immediately the oscilloscope trigger button is pressed.

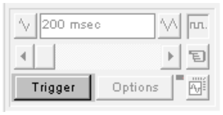
Button Depressed = Program Trigger Mode:

In program mode the oscilloscope starts running when the trigger button is pressed, but the controller does not start to record data until a TRIGGER instruction is executed by a program running on the controller. After the trigger instruction is executed by the program, and the controller has recorded the required data. The required data is retrieved by the oscilloscope and displayed.

The oscilloscope stops running if in one-shot mode, or it waits for the next trigger on the controller if in continuous mode

Trigger Button

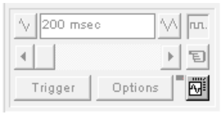
When the trigger button is pressed the oscilloscope is enabled. If it is manual mode the controller immediately commences recording data. If it is in program mode then it waits until it encounters a trigger command in a running program.



After the trigger button has been pressed, the text on the button changes to ‘Halt’ whilst the oscilloscope is running. If the oscilloscope is in the one-shot mode, then after the data has been recorded and plotted on the display, the trigger button text returns to ‘Trigger’, indicating that the operation has been completed.The oscilloscope can be halted at any time when it is running, and the trigger button is displaying the ‘Halt’ text, by pressing this button.

Reset Oscilloscope Configuration

The current scope configuration (the state of all the controls) is saved when the scope window is closed, and retrieved when the scope window is next opened. This removes the need to re-set each individual control every time the scope window is opened.



The configuration reset button (located at the bottom right hand side of the scope control panel) can be pressed to reset the scope configuration, clearing all controls to their default values.

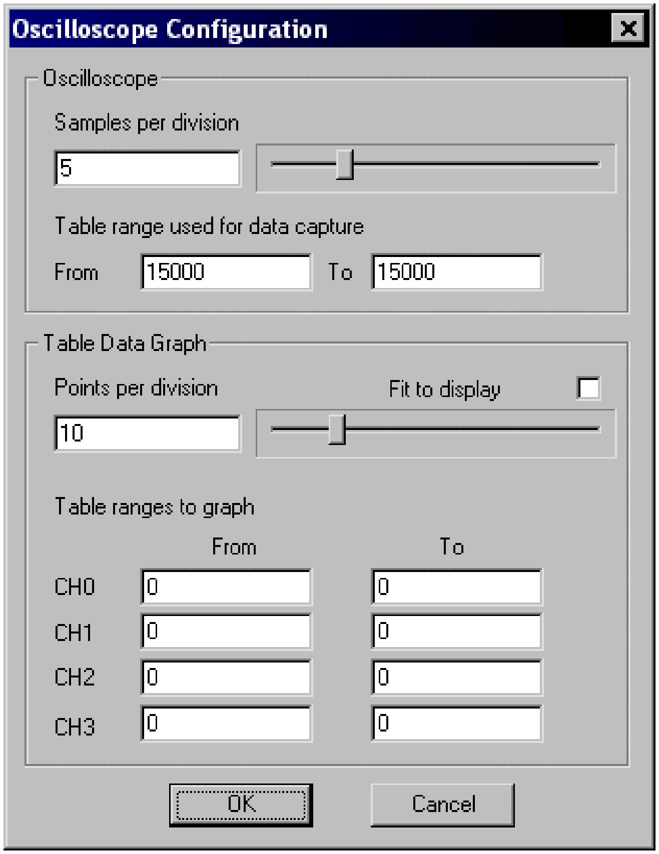
Status Indicator

The status indicator is located in between the options and configuration reset buttons. This lamp changes colour according to the current status of the scope, as follows:

`Red`	oscilloscope stopped.
`Black`	polling controller waiting for it to complete recording the required data.
`Yellow`	retrieving data from the controller.

Advanced Oscilloscope Configuration Options

When the options button is pressed the advanced oscilloscope configuration settings dialog is displayed, as shown below. Click the mouse button over the various controls to reveal further information.


Oscilloscope Configuration

Oscilloscope

Samples per division: 5

Table range used for data capture

From 15000 To 15000

Table Data Graph

Points per division: 10

Fit to display: ☐

Table ranges to graph

	From	To
CH0	0	0
CH1	0	0
CH2	0	0
CH3	0	0

Samples per division

The oscilloscope defaults to recording five points per horizontal (time base) grid division. This value can be adjusted using the adjacent scrollbar.

To achieve the fastest possible sample rate it is necessary to reduce the number of samples per grid division to 1, and increase the time base scale to its fastest value (1 servo period per grid division).

It should be noted that the trace might not be plotted completely to the right hand side of the display, depending upon the time base scale and number of samples per grid division.

Oscilloscope Table Values

The controller records the required parameter data values in the controller as table data prior to uploading these values to the scope. By default, the lowest oscilloscope table value used is zero. However, if this conflicts with programs running on the controller which might also require this section of the table, then the lower table value can be reset.

The lower table value is adjusted by setting focus to this text box and typing in the new value. The upper oscilloscope table value is subsequently automatically updated (this value cannot be changed by the user), based on the number of channels in use and the number of samples per grid division. If an attempt is made to enter a lower table value which causes the upper table value to exceed the maximum permitted value on the controller, then the original value is used by the oscilloscope.

Table Data Graph

It is possible to plot controller table values directly, in which case the table limit text boxes enable the user to enter up to four sets of first/last table indices.

Parameter Checks

If analog inputs are being recorded, then the fastest oscilloscope resolution (sample rate) is the number of analog channels in msec (ie 2 analog inputs infers the fastest sample rate is 2msec). The resolution is calculated by dividing the time base scale value by the number of samples per grid division.

It is not possible to enter table channel vales in excess of the controllers maximum TABLE size, nor to enter a lower oscilloscope table value. Increasing the samples per grid division to a value which causes the upper oscilloscope table value to exceed the controller maximum table value is also not permitted.

If the number of samples per grid division is increased, and subsequently the time base scale is set to a faster value which causes an unobtainable resolution, the oscilloscope automatically resets the number of samples per grid division.

General Oscilloscope Information

Displaying Controller Table Points -

If the oscilloscope is configured for both table and motion parameters, then the number of points plotted across the display is determined by the time base (and samples per division). If the number of points to be plotted for the table parameter is greater than the number of points for the motion parameter, the additional table points are not displayed, but can be viewed by scrolling the table trace using the horizontal scrollbar. The motion parameter trace does not move.

Data Upload from the controller to the oscilloscope -

If the overall time base is greater than a predefined value, then the data is retrieved from the controller in blocks, hence the display can be seen to be updated in sections. The last point plotted in the current section is seen as a white spot.

If the oscilloscope is configured to record both motion parameters, and also to plot table data, then the table data is read back in one complete block, and then the motion parameters are read either continuously or in blocks (depending upon the time base).

Even if the oscilloscope is in continuous mode, the table data is not re-read, only the motion parameters are continuously read back from the controller.

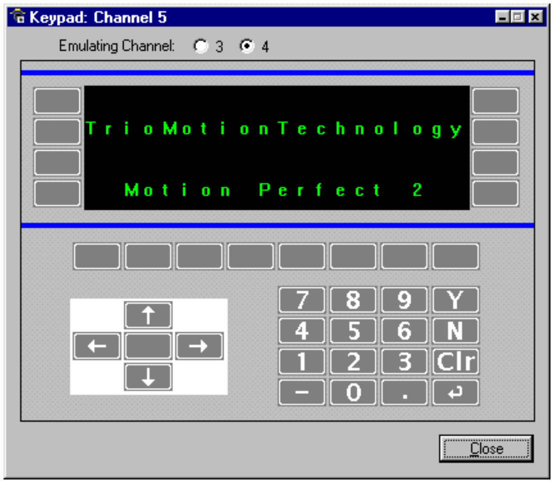
Enabling/Disabling of oscilloscope controls -

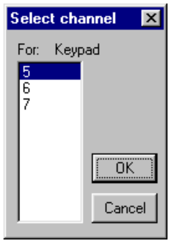
Whilst the oscilloscope is running all the oscilloscope controls except the trigger button are disabled. Hence, if it is necessary to change the time base or vertical scale, the oscilloscope must be halted and re-started.

Display accuracy -

The controller records the parameter values at the required sample rate in the table, and then passes the information to the oscilloscope. Hence the trace displayed is accurate with respect to the selected time base. However, there is a delay between when the data is recorded by the controller and when it is displayed on the oscilloscope due to the time taken to upload the data via the serial link.

Keypad Emulation





The keypad requires one of the user communications channels, and so you will be prompted for the channel to use.

If the specified channel is already in use, either by another keypad or a terminal window, the window will not open. Once a channel has been reserved then the keypad will be shown.

In the Trio BASIC program the channel definition for the commands that are associated with the Keypad must be changed from 3 (or 4) to the channel that corresponds with the channel selected for the emulation. We recommend that the channel assignment be made through a variable, so when time comes to run the program on the real machine, only one program change will be required.

example:
```
kpd=5
PRINT #kpd, "Press any key.."
```

Emulating Channel

The normal operation of the keypad emulation returns the characters as if they were read from channel #3 with the **DEFKEY** translation. Alternatively, the *Motion Coordinator* can read the characters returned directly from the Keypad using channel 4. If the emulate #4 codes is selected then the keypad emulation will return the raw characters.

Note: It is only possible to emulate the default **DEFKEY** table.

Key Functions

menu keys	This is a keypad menu key. Normally it is associated with a message on the display. This button can only be pressed by clicking the mouse over it.
function keys 1-8	This is the keypad function key 1. Normally it has an associated user label. This button can be pressed by clicking the mouse over it or using the ‘1’ - '8' keys in the QWERTY area of the PC keyboard.
number keys	This is a keypad number key. It can be pressed by clinking the mouse over it or using the corresponding number in the numerical keypad of your PC keyboard.
Y/N keys	This is the keypad ‘Y’ and 'N' keys. This is usually used to respond YES or NO to some question on the display. It can be pressed by clicking the mouse over it or using the ‘Y’/'N' keys in the QWERTY area of the PC keyboard.
CLR key	This is the keypad ‘CLR’ key. This is usually used to perform some form of CANCEL operation. It can be pressed by clicking the mouse over it or using the ‘ESC’ in the QWERTY area of the PC keyboard.
Return key	This is the keypad Return key. This is usually used to perform some form of ACCEPT operation. It can be pressed by clicking the mouse over it or using the ‘Enter’ in the QWERTY area or numerical keypad of the PC keyboard.
- key	This is the keypad ‘-’ key. This is usually used for entering negative numbers. It can be pressed by clicking the mouse over it or using the ‘-’ in the QWERTY area or numerical keypad of the PC keyboard.
. key	This is the keypad ‘.’ key. This is usually used for entering fractional numbers. It can be pressed by clicking the mouse over it or using the ‘.’ in the QWERTY area or numerical keypad of the PC keyboard.
arrow keys	This is the keypad up arrow key. This is usually used to select between options on the display. It can be pressed by clicking the mouse over it or using the appropriate arrow key of the PC keyboard.
centre button	This is the keypad centre key. It can only be pressed by clicking the mouse over it.

Table / VR Editor

The Table and VR Editor tools are very similar. You are presented with a spreadsheet style interface to view and modify a range of values in memory.

To modify a value, click on the existing value with the mouse and type in the new value and press return. The change will be immediate and can be made whilst programs are running.

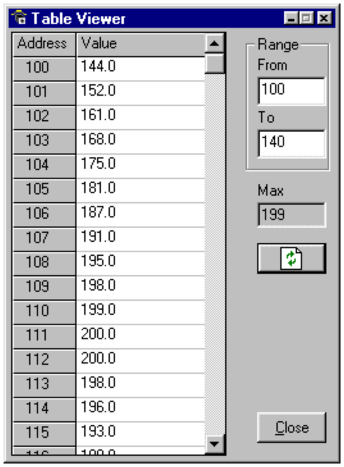


Table Viewer

Address	Value
100	144.0
101	152.0
102	161.0
103	168.0
104	175.0
105	181.0
106	187.0
107	191.0
108	195.0
109	198.0
110	199.0
111	200.0
112	200.0
113	198.0
114	196.0
115	193.0

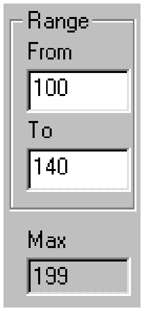
Range
From 100
To 140
Max 199

Options

Range

In both tools you have the option to set the start and end of the range to view. In the Table view tool the max value displays the highest value you can read (this is the system parameter TSIZE).

If the range of values is larger than the dialog box can display, then the list will have a scrollbar to enable all the values to be seen.

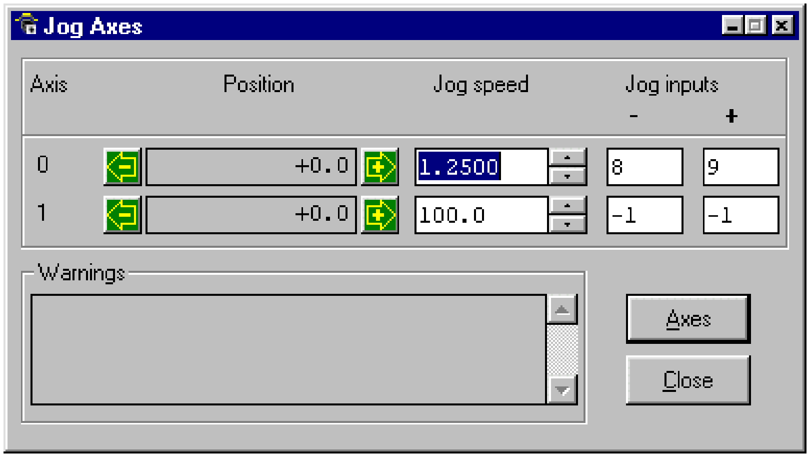

Range
From 100
To 140
Max 199

Refresh Button

This screen does not update automatically, so if a Table or VR is changed by the program you will not see the new value until you refresh the display.

Jog Axes

This window allows the user to move the axes on the *Motion Coordinator*.



Axis	Position	Jog speed	Jog inputs -	Jog inputs +
0	+0.0	1.2500	8	9
1	+0.0	100.0	-1	-1

This window takes advantage of the bi-directional I/O channels on the *Motion Coordinator* to set the jog inputs. The forward, reverse and fast jog inputs are identified by writing to the corresponding axis parameters and are expected to be connected to NC switches. This means that when the input is on (+24V applied) then the corresponding jog function is DISABLED and when the input is off (0V) then the jog function is ENABLED.

The jog functions implemented here disable the fast jog function, which means that the speed at which the jog will be performed is set by the **JOGSPEED** axis parameter. What is more this window limits the jog speed to the range 0..demand speed, where the demand speed is given by the **SPEED** axis parameter.

Before allowing a jog to be initiated, the jog window checks that all the data set in the jog window and on the *Motion Coordinator* is valid for a jog to be performed.

Jog Reverse

This button will initiate a reverse jog. In order to do this, the following check sequence is performed:

- If this is a SERVO or RESOLVER axis and the servo is off then set the warning message
- If this axis has a daughter board and the WatchDog is off then set the warning message
- If the jog speed is 0 the set the warning message
- If the acceleration rate on this axis is 0 then set the warning message
- If the deceleration rate on this axis is 0 then set the warning message
- If the reverse jog input is out of range then set the warning message
- If there is already a move being performed on this axis that is not a jog move then set the warning message

If there were no warnings set, then the message "Reverse jog set on axis?" is set in the warnings window, the **FAST_JOG** input is invalidated for this axis, the **CREEP** is set to the value given in the jog speed control and finally the **JOG_REV** output is turned off, thus enabling the reverse jog function.

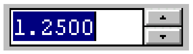
Jog Forward

This button will initiate a forward jog. In order to do this, the following check sequence is performed:

- If this is a SERVO or RESOLVER axis and the servo is off then set the warning message
- If this axis has a daughter board and the WatchDog is off then set the warning message
- If the jog speed is 0 the set the warning message
- If the acceleration rate on this axis is 0 then set the warning message
- If the deceleration rate on this axis is 0 then set the warning message
- If the reverse jog input is out of range then set the warning message
- If there is already a move being performed on this axis that is not a jog move then set the warning message

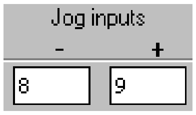
If there were no warnings set, then the message "Forward jog set on axis?" is set in the warnings window, the **FAST_JOG** input is invalidated for this axis, the **CREEP** is set to the value given in the jog speed control, and finally the **JOG_FWD** output is turned off, thus enabling the forward jog function.

Jog Speed



This is the speed at which the jog will be performed. This window limits this value to the range from zero to the demand speed for this axis, where the demand speed is given by the **SPEED** axis parameter. This value can be changed by writing directly to this control or using the jog speed control. The scroll bar changes the jog speed up or down in increments of 1 unit per second

Jog Inputs



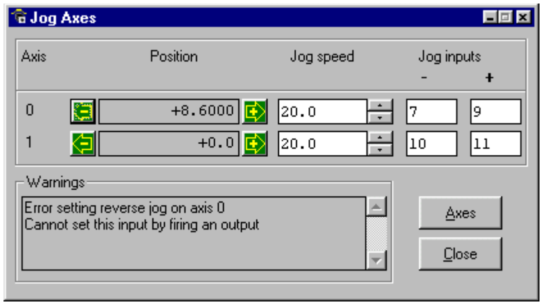
These are the inputs which will be associated with the forward / reverse jog functions.

They must be in the range 8 to the total number of inputs in the system as the input channels 0 to 7 are not bi-directional and so the state of the input cannot be set by the corresponding output.

The input is expected to be ON for the jog function to be disabled and OFF for the reverse jog to be enabled. In order to respect this, when this is set to a valid input number, the corresponding output is set ON and then the corresponding **REV_JOG** axis parameter is set.

Warnings

This shows the status of the last jog request. For example, the screen below shows axis 0 with IO channel 7 selected. This is an Input-only channel and therefore cannot be used in the jog screen.

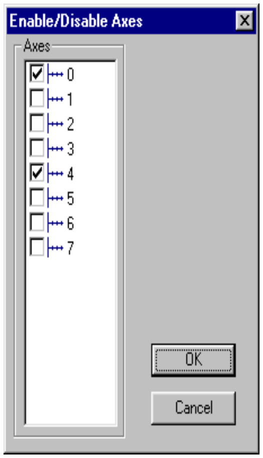


Axis	Position	Jog speed	Jog inputs -	Jog inputs +
0	+8.6000	20.0	7	9
1	+0.0	20.0	10	11

Warnings

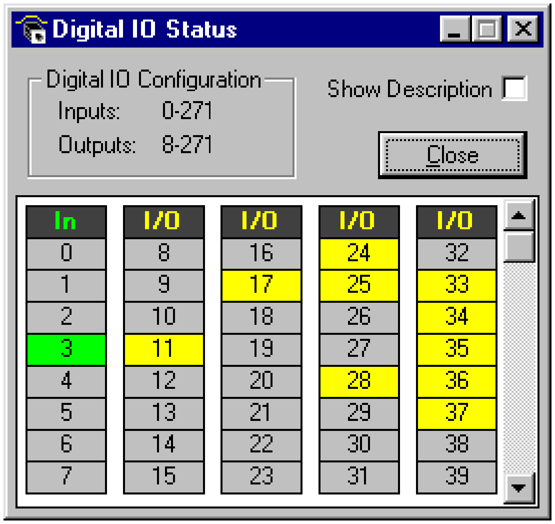
Error setting reverse jog on axis 0
Cannot set this input by firing an output

Axes



This displays an axis selector box which enables the user to select the axis to include in the jog axes display. By default, the physical axes fitted to the controller will be displayed.

Digital IO Status



Digital IO Configuration
Inputs: 0-271
Outputs: 8-271

In	I/O	I/O	I/O	I/O
0	8	16	24	32
1	9	17	25	33
2	10	18	26	34
3	11	19	27	35
4	12	20	28	36
5	13	21	29	37
6	14	22	30	38
7	15	23	31	39

This window allows the user to view the status of all the IO channels and toggle the status of the output channels. It also optionally allows the user to enter a description for each I/O line.

Digital Inputs

This shows the total number of input channels on the *Motion Coordinator*.

Digital Outputs

This shows the total number of output channels on the *Motion Coordinator*.

IO Mimic

Input Bank 0-7

This first bank of 8 LEDs shows the status of the dedicated input channels. If an LED is green then the corresponding input is ON. If an LED is white then the corresponding input is OFF.

Input/Output Banks

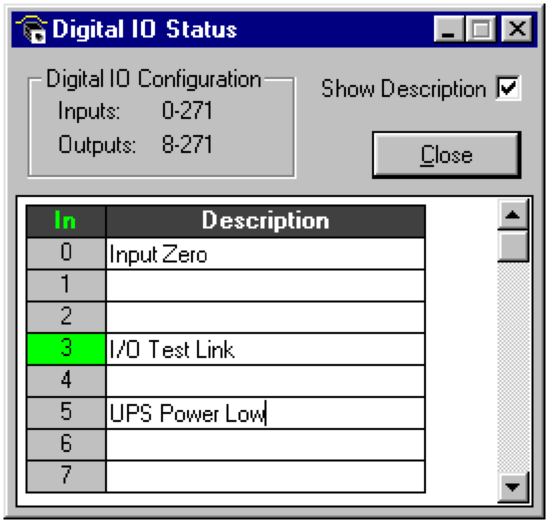
These banks of LEDs show the status the bi-directional IO channels. If an LED is yellow then the corresponding input is ON. If an LED is white then the corresponding input is OFF. Under normal conditions the input status mimics the output status, except:

5) If this input is connected to an external 24V then it may be on without the corresponding output being on.

6) If the output chip detects an overcurrent situation, then the output chip will shut down and so the outputs will not be driven, even though they may be turned on.

If the LED is clicked with the mouse the status corresponding output channel is toggled, i.e. if the LED is white then the output will be turned on, if the LED is yellow then the output channel will be turned off.

Checking the **Show Description** check box will toggle between dsecriptions on, and descriptions off. Descriptions are stored in the project file



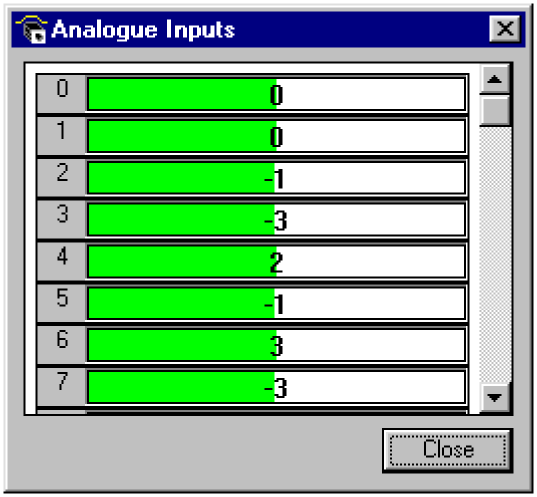
Digital IO Configuration
Inputs: 0-271
Outputs: 8-271

Show Description ✓

In	Description
0	Input Zero
1	
2	
3	I/O Test Link
4	
5	UPS Power Low
6	
7	

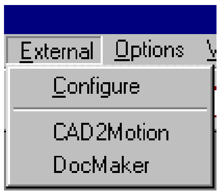
Analogue Input Viewer

The analogue input viewer is only available if the system has analogue inputs. It displays the input values of all analogue inputs in the system using a bar-graph with numeric display. All inputs have the range -2048 to 2047.



0	0
1	0
2	-1
3	-3
4	2
5	-1
6	3
7	-3

Linking to External Tools

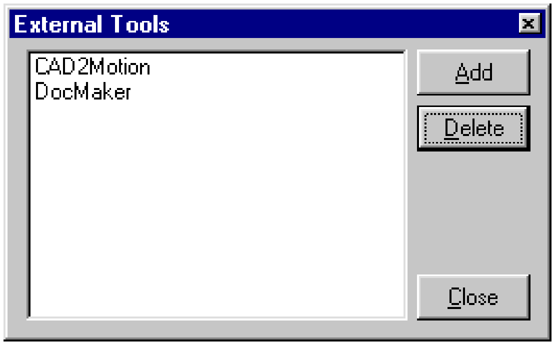


The EXTERNAL menu in *Motion* Perfect allows you to run other programs directly from the main *Motion* Perfect menu. In our example shown here, the menu has been configured to launch two other Trio applications, CAD2Motion and DocMaker. Further information on these applications is given at the end of this chapter.

Note: ***Cad2Motion and DocMaker are available to download from the Trio Website at*** `www.triomotion.com`.

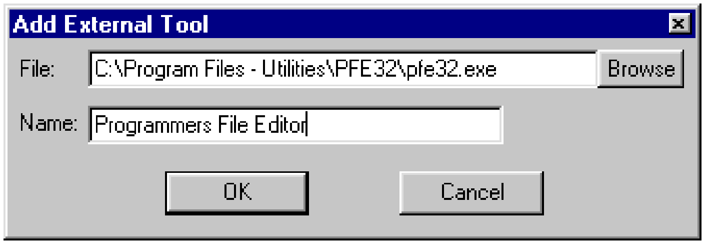
Configuring Items on the External menu

Clicking on the Configure item will bring up a list of all installed applications and from here we can add or delete items from this list.



Adding a new programs to the menu

Clicking on the Add button will open the following dialog:



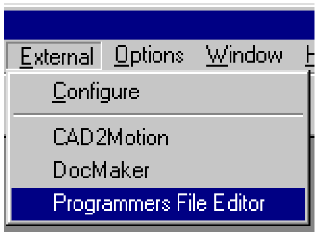
Add External Tool

File: C:\Program Files - Utilities\PFE32\pfe32.exe

Name: Programmers File Editor

You can either directly enter the path and program file name in the "File" box, or use the "Browse" option to open up a standard windows file selector box which you can use to locate the file on your computer.

Once you have selected the file, it will automatically appear in the External menu every time you run *Motion* Perfect 2.

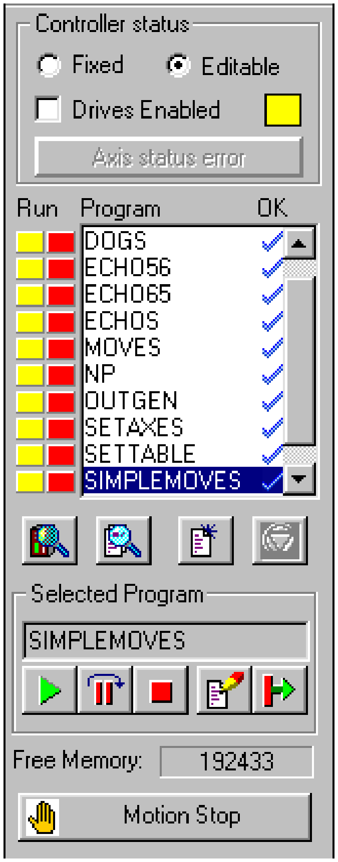


Removing program items from the menu

To delete a program from the External menu, you simply need to click on the program name in the list and press the Delete button.

Note: ***This simply removes the program from the menu. it does NOT affect the original program on disk!***

Control Panel



Programs listed: DOGS, ECHO56, ECHO65, ECHOS, MOVES, NP, OUTGEN, SETAXES, SETTABLE, SIMPLEMOVES

Selected Program: SIMPLEMOVES

Free Memory: 192433

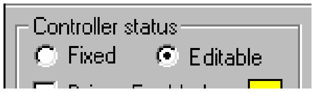
The control panel appears on the left hand side of the main *Motion* Perfect window.

It provides direct links to many of the frequently used operations within *Motion* Perfect, in particular the file and directory functions.

Please Note: Certain Control Panel Features behave differently on controller without a battery backup. The differences are described later in this section.

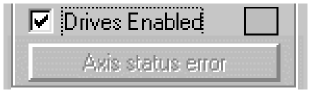
Control Panel Features

Fixed/Editable radio buttons

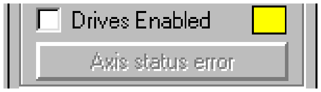


When the project is “fixed”, the programs are copied to the Flash EPROM on the *Motion Coordinator*, the *Motion Coordinator* is set to run from EPROM and the programs cannot be modified by *Motion* Perfect. Usually this is done when the machine programs are completed. The Flash EPROM provides a reliable permanent storage for the programs.

Drives radio button



Drives Enabled

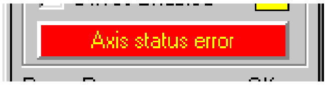


Drives Disabled

This radio button toggles the state of the enable (watchdog) relay on the controller, going between drives disabled (watchdog off) and drives enabled (watchdog on).

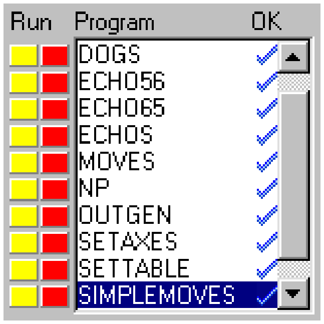
The LED mimic next to this control shows the status of the error LED on the *Motion Coordinator*. If it is yellow then the drives are disabled, if it is grey the drives are enabled and if it is flashing then there has been a motion error on at least one axis of the controller.

Axis Status Error



This will normally be greyed out unless a motion error occurs on the controller.
When an error does occur you can use this button to clear the error condition.

Program directory



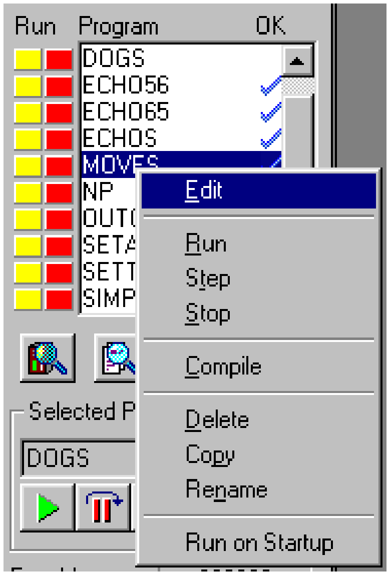
This is a scrollable list of the programs on the controller. The list shows the program name followed by two optional indicators. The first is a number which specifies which process that program is running on. If it is not running then this space is blank. The second indicator shows the status of the program. If it has a tick then the program has been compiled successfully and is ready to be run. If it has a cross then there was an error during the compilation of the program and it cannot be run.

If it is blank then it has not been compiled.

If a program name is clicked then it will become the selected program if there are no programs running. If there are programs running the select will be ignored.

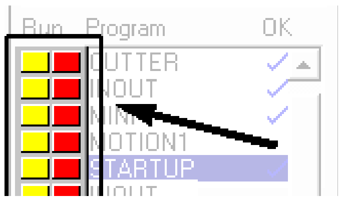
If a program name is double clicked then it will be opened for editing assuming that there are no programs running. If there are programs running then the edit will be ignored.

The names of programs which are currently running are displayed in italics.



Right clicking on an entry in the program list causes a pop-up menu to appear allowing easy access to operations commonly performed on programs. The right click operation highlights the entry in the program list under the cursor whilst the pop-up menu is visible. It reverts to the program which is currently selected on the controller when the menu is closed.

Run buttons

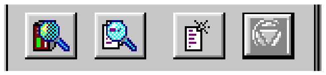


The run buttons provide short cut keys for running, stopping and single stepping programs. They can be in one of three states, red, green or yellow.

RED	Click on the red button to start the corresponding program running. The button will turn Green. *
GREEN	Click on the green button to stop the corresponding program. The button will turn red. *
YELLOW	Click on the yellow button to single step through the program.

* If the program goes into trace mode, through the use of a trace button, the selected program step button, the debug option of the program menu, the debug button on the tool bar or a TRON/STEPLINE command in a program or the terminal window, then the red/green run button will turn yellow. If the button is clicked when it is yellow then the program will be stepped one line.

General Options



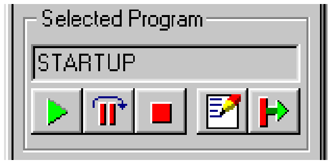
Show Controller Configuration

Show a full directory of all programs in memory

Create a New Program. Same as the Program menu item.

HALT - Stop all programs which are running

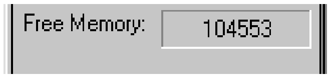
Selected Program



The text box displays the currently selected program and the buttons below, the operations which can be performed on that program.

From left to right they are:
Run, Step, Stop, Edit and Power Up Mode

Free Memory



Shows the total free memory available on the controller

Motion Stop

Cancel moves on all axes and disable the watchdog relay.

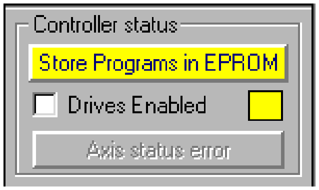
Note *MOTION STOP is a software function. It is not a substitute for a hardware E_Stop circuit and should not be used as an emergency stop.*

Control Panel Variations for Motion Coordinators without battery backup

On those controllers without a battery backup such as the MC202, the fixed / editable radio buttons are replace by a single button labeled "Store Programs into Eprom".

As the controller does not feature a battery, it is essential that you store your programs into the eprom to avoid loss of data.

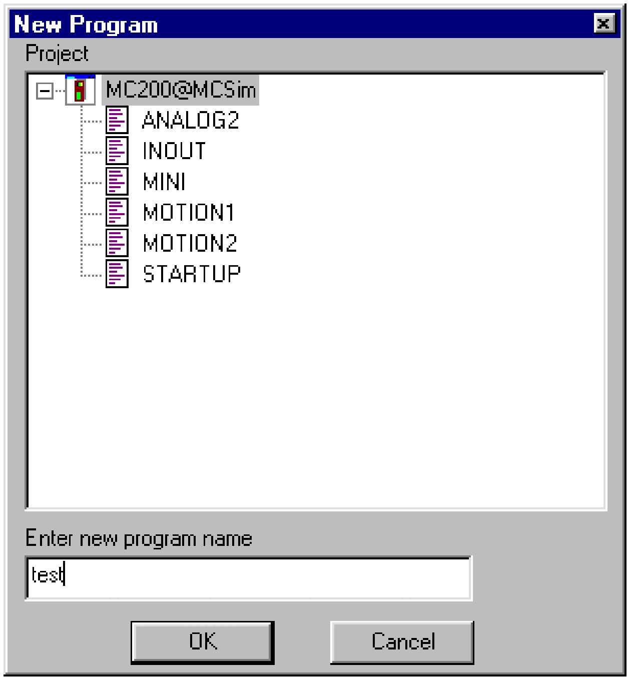
If the programs in memory have been edited, the button will be highlighted to remind you to fix into eprom before exiting from the program



Creating and Running a program

In order to create a new program on the controller, you must first have an active project. If you have already connected to the controller then you can use the default project which was created at this time.

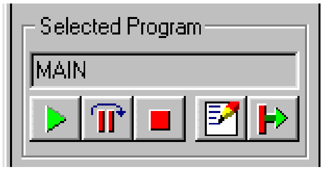
You will be presented with a program selector dialog and prompted to enter a name for your program file. It is a good idea to make this name representative of the task performed by the program, for example “mmi”, “motion”, “logic” or something similar. In the following example, we will add a program called “test” to the current project.



Once you have created a new program it will be added to both the controller and the *Motion* Perfect project file. You can now edit the file in *Motion* Perfect Editor.

The *Motion* Perfect Editor

You can start the Editor from the main Program Menu, the Edit button in the program section of the control panel or by right clicking in an entry in the control panel program list and selecting **Edit** from the pop-up menu.



Program Section of the Control Panel

If you launch the editor from the control panel it will start immediately. From the program menu you will first be prompted with a program selector dialog to confirm the file you wish to edit.

The *Motion* Perfect Editor is designed to operate in a similar manner to any simple text editor found on a PC. Standard operations such a block editing functions, text search and replace and printing are all supported and conform to the standard Windows shortcut keys. In addition it provides TrioBASIC syntax highlighting, program formatting and program debugging facilities.

Editor Options

Options for the editor are controlled by the **Editor Options** dialog. This can be opened by selecting **Options/Editor** from *Motion* Perfect's main menu. The dialog allows to user to change the fonts used for screen display and printing, the colours used for syntax and line highlighting and the spacings used when automatically formatting a program.

Screen Font

This is the font used by *Motion* Perfect to display text in the editor window on the screen. The font is restricted to fixed pitch fonts only.

Printer Font

This is the font used by *Motion* Perfect to print program listings. The font is restricted to fixed pitch fonts only.

Colours

Colours can be specified for the following:

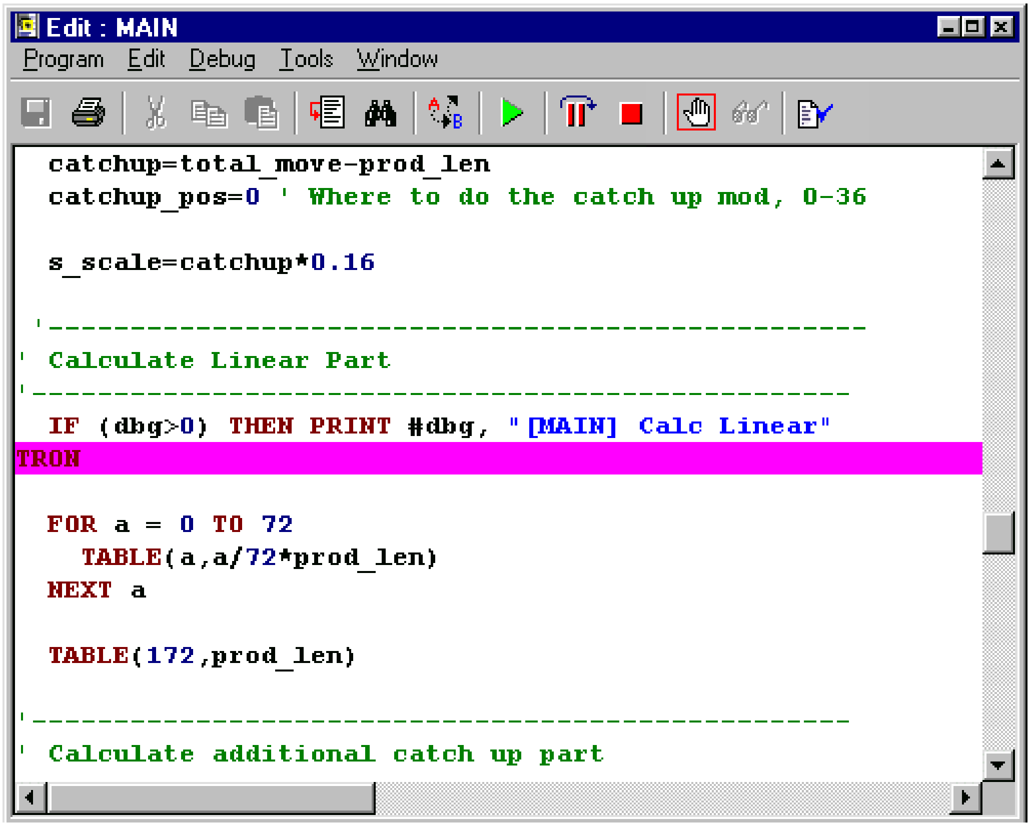
- Normal Text - Text which is not highlighted using systax highlighting.
- Screen Background.
- Current Line (background) - the current line during debugging.
- Break Line (background) - a line containing a break (TRON) command.
- Current Line Break (background) - a line containing a break command which is also the current line.
- Error Line (background) - The first line containing a compilation error.
- BASIC key word - A key word in the TrioBASIC language, usually a command or some type of system variable.

- Comment Text
- Constant Text - text making up a constant value (number).
- Strings
- Label Defenition - where a program label is defined.
- Label Reference - where a jump or branch (GOTO, GOSUB etc.) in program execution is required. The jump or branch changes the execution point to the place where the label is defined.

Format

The format options affect text entry and the automatic reformatting preformed by *Motion* Perfect.
For automatic reformatting the code start column and the tab width are specifiable. As label definitions always start in column 0, the code start column can be used to indent all lines containing code thus making label definitions clearer.

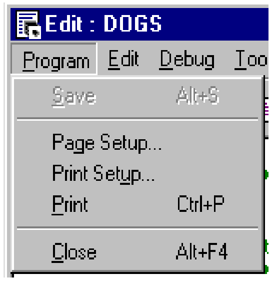
The when **Auto tab on enter** check box is checked pressing the “enter” key will automatically indent the next (new) line to the same position as the current one.



```
  catchup=total_move-prod_len
  catchup_pos=0 ' Where to do the catch up mod, 0-36

  s_scale=catchup*0.16

 '--------------------------------------------------
' Calculate Linear Part
'--------------------------------------------------
  IF (dbg>0) THEN PRINT #dbg, "[MAIN] Calc Linear"
TRON

  FOR a = 0 TO 72
    TABLE(a,a/72*prod_len)
  NEXT a

  TABLE(172,prod_len)

'--------------------------------------------------
' Calculate additional catch up part
```

Editor Menus

Program



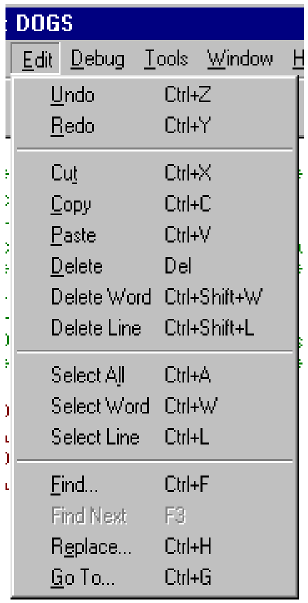
Save

Normally the program is only saved to disk when the editor is closed or a program is run, however if you have modified the program the Save Button will be available and will force *Motion* Perfect to save the file immediately.

Printing

Use **Page Setup** to set the page mrrgins, **Print Setup** to configure you printer settings and the **Print** option to send the program to the printer.

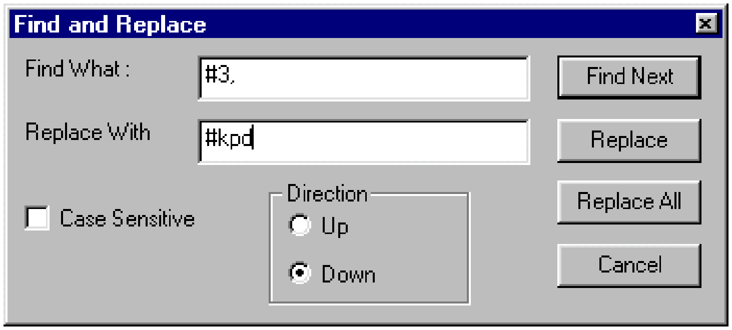
Edit



The edit menu functions are similar to many other text editors and provide the standard block cut/ copy/paste operations as well as a simple text find/ replace, and various select and delete functions.

Find/Replace

The options for the find & replace dialogs are very similar and feature many of the same options



Find and Replace

Find What : #3,

Replace With #kpd

[ ] Case Sensitive

Direction: Up [✓ Down]

You should enter the text to search for in the “Find What” box, and if using find and replace, the text to replace it with in the “Replace With” box.

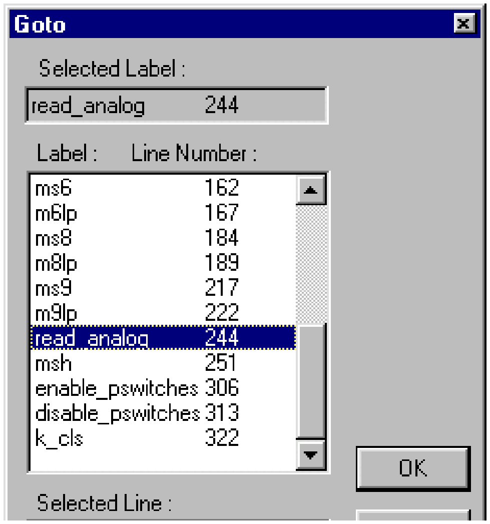
Normally the “Case Sensitive” search option is not selected, You should only use this option if you have an exact pattern to match, generally the default option is best.

GOTO

The Goto option will bring up the following dialog:

A list of labels defined in the program is displayed. You can either select a label from here, or enter a line number directly in the “Selected Line” field.

Press OK and the cursor will jump directly to the beginning of the selected line.



Goto

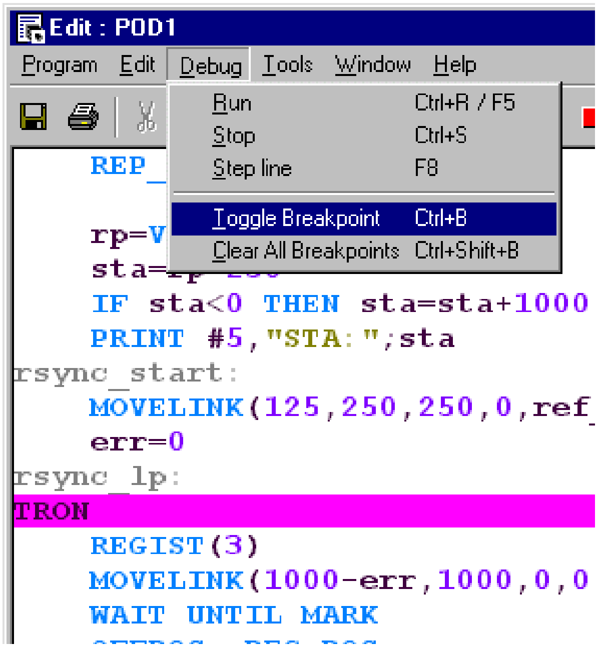
Selected Label : read_analog 244

Label :	Line Number :
ms6	162
m6lp	167
ms8	184
m8lp	189
ms9	217
m9lp	222
read_analog	244
msh	251
enable_pswitches	306
disable_pswitches	313
k_cls	322

Selected Line :

Program Debugger

The *Motion* Perfect debugger allows you to run a program directly from the editor window in a special 'trace mode, executing one line at a time (known as stepping) whilst viewing the line in the window. It is also possible to set breakpoints in the program, and run it at normal speed until it reaches the breakpoint where it will stop, and this line of code will be highlighted in the debug window.



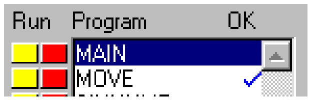
```
    REP_
    rp=V
    sta=
    IF sta<0 THEN sta=sta+1000
    PRINT #5,"STA:";sta
rsync_start:
    MOVELINK(125,250,250,0,ref_
    err=0
rsync_lp:
TRON
    REGIST(3)
    MOVELINK(1000-err,1000,0,0
    WAIT UNTIL MARK
```

When programs are running on the controller, any open editor windows will automatically switch to Debug Mode and will become read-only. Hence, breakpoints are set in the edit window, and the code viewed in the same window in debug mode when the program is running.

Stepping Through a program

To commence stepping a program:

Use the mouse to press the yellow button alongside the required program name in the list box on the control panel



if the required program is currently selected, press the 'Step' button on the control panel() or use the menu item 'Debug-Step line'

The currently executing line of code is indicated in the debug window by highlighting it with a green background, and a breakpoint is highlighted with a red background.

To continue stepping the program, repeatedly press the yellow button alongside the program name in the list box on the control panel, or press the 'Step' button or the 'F8' functional key if the program required is currently selected on the *Motion Coordinator*.

Breakpoints

Breakpoints are special place markers in the code which allow a particular section (or sections) of the program to be identified when debugging the code. If a breakpoint is inserted, the program will pause at that point and return control to *Motion* Perfect where the controller may be interrogated or the program run in step mode as described above.

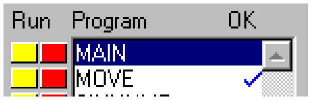
To insert a breakpoint, first position the text cursor on the line at which you want the break to occur, then use either Ctrl-B or the menu item to insert the breakpoint.

The Trio BASIC instruction TRON is used to mark a breakpoint and TROFF to terminate a 'traced' block.

Note: ***It is not possible to add or remove breakpoints whilst any programs are running.***

Running to a breakpoint

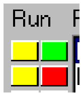
A program can be run to the next break point by:



- using the mouse to press the red button alongside the program name in the list box on the control panel.
- if it is the currently selected program on the *Motion Coordinator*, you can pressing the 'Run' button (▶) on the control panel/editor tool bar, or by using the keyboard <F5> function key.
- by selecting the 'Debug->Run' menu option

Stopping a Program

If it is necessary to stop the program running before it reaches the breakpoint then:



- press the green button alongside the program name (running on the required process) in the list box on the control panel.
- press the stop button (■)on the control panel if the program is currently selected (this will stop all running copies of the program)
- use the 'Debug-Stop' menu option.

Alternatively all programs can be stopped by pressing the 'Halt' button on the control panel, or selecting the 'Program' 'Halt all programs' menu option, or using the <Ctrl><F> key combination.

Switching a running program into trace mode

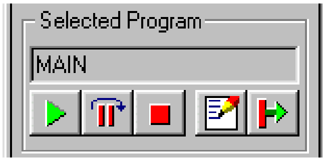
A running program can enter trace (stepping) mode by pressing the yellow button alongside the required program name in the list box on the control panel, or the 'Step' button if the required program is currently selected on the *Motion Coordinator*.

Running Programs

You can start/stop programs running in one of four ways:

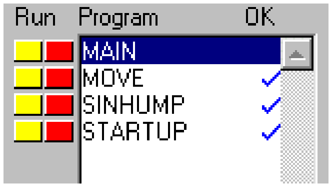
From the control panel

If the program is currently selected (highlighted in the control panel), you can press the green start arrow in the "selected program" box.

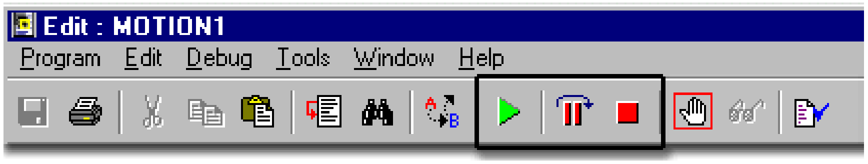


From the program list

Pressing the red button to the left of the program name in the list will start it running, the button will change to green and it will then function as a stop button for the same task.



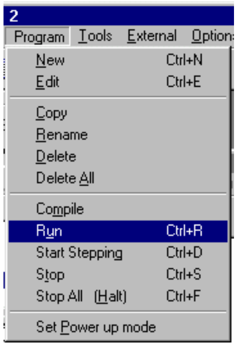
From the editor toolbar



If you have an editor or debug window open for the program you can use either the Debug menu or toolbar buttons to start the program running

From the Program Menu

The program menu provides us with a slightly different option when running the program as we are presented with a program selector box which includes an option to choose which task we want to run the program on

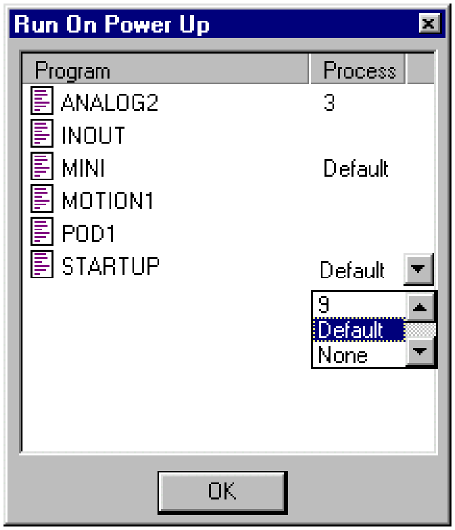


Making programs run automatically

Set Powerup Mode

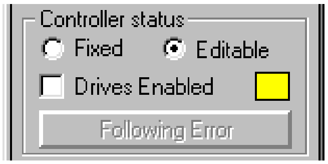
It is possible to make the programs on the controller run automatically when the system first starts up. From the **Program** Menu, select “**Set Powerup Mode**” to open the following dialog.

Click on the program you want to auto run and a small drop-down list will appear to the right of the window. If you are happy to let the controller allocate which task to run on then you should choose “default” as the process number, otherwise you can specify the task explicitly in the box.



Program	Process
ANALOG2	3
INOUT	
MINI	Default
MOTION1	
POD1	
STARTUP	Default

Storing Programs in the Flash EPROM



This is accomplished by selecting the “Fixed” option in the controller status section of the control panel, or the “Fix Program Into EPROM” option from the controller menu.

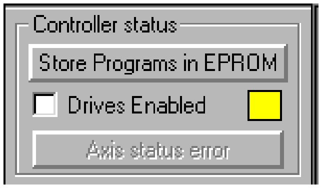
When the controller is fixed into eprom, the programs actually still run from RAM. The information stored is copied into RAM when the controller is first started, therefore if the controller has been switched off for an extended period, or there is any corruption of the RAM, it will be refreshed with a correct copy of the programs.

When the controller is set to fixed you will not be able to edit any programs. In order to make changes you must select “Editable” from the control panel or “Enable Editing” from the controller menu.

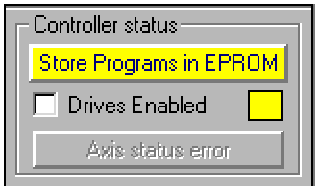
Variations for controllers without battery backup

On those controllers without battery backup, such as the MC202, it is essential that you store your programs into the eprom to avoid loss of data.

The control panel the fixed / editable radio buttons are replaced by a single button labelled "Store Programs into Eprom".

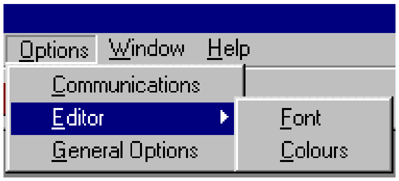


If the programs in memory have been edited, the button will be highlighted to remind you to fix into eprom before exiting the program.



Configuring The *Motion* Perfect 2 Desktop

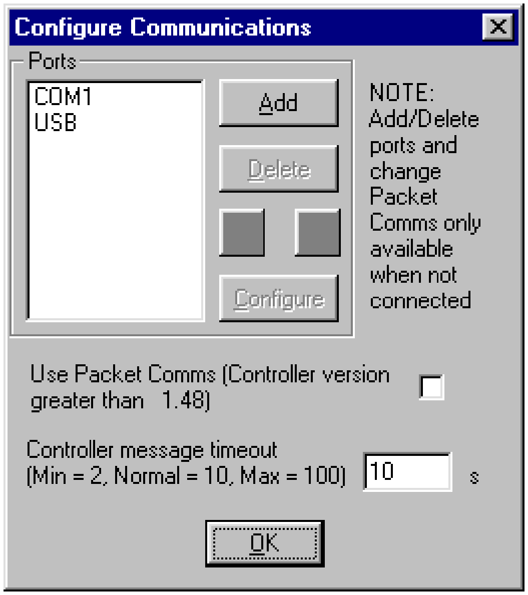
There are a number of ways in which you can configure *Motion* Perfect 2 to suit your requirements.   The Options menu provides a number of choices:-



Communications

Set up the default communications device for *Motion* Perfect 2 to use.

Motion Perfect 2 needs a single serial connection to the controller in order to operate. This can be an RS-232 serial connection, or a USB connection if your controller and PC have USB ports. It will normally recognise the ports installed in your PC and will display these in the Configure Communications window.



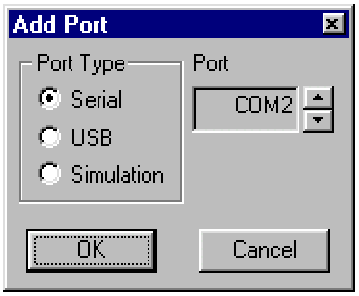
COM1
USB

NOTE: Add/Delete ports and change Packet Comms only available when not connected

Use Packet Comms (Controller version greater than 1.48) [ ]

Controller message timeout (Min = 2, Normal = 10, Max = 100) 10 s

If the port you wish to use is not shown, you need to select the Add Port option which will select the following dialog.



Port Type: [✓ Serial] USB Simulation

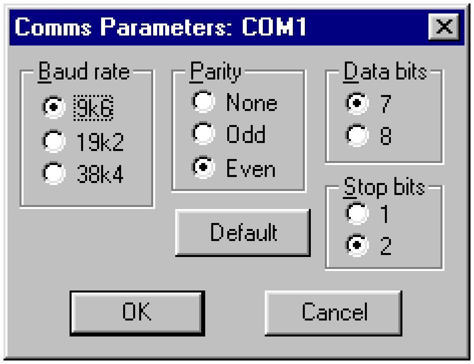
Port: COM2

Most *Motion* Perfect 2 users will connect via a standard RS-232 serial (COM) port. Other options include a connection via a high speed Universal Serial Bus (USB) port or a controller simulation for offline editing.

Choose the port options you require and click on the "OK" button to install the new port.

Changing Comms Port Parameters

The default settings for serial communications is:- 9600 baud, 7 data bits, 2 stop bits, even parity. If you wish to change these values you can do so with the configure button in the Configure Communications dialog.



Baud rate: [✓ 9k6] 19k2 38k4

Parity: None Odd [✓ Even]

Data bits: [✓ 7] 8

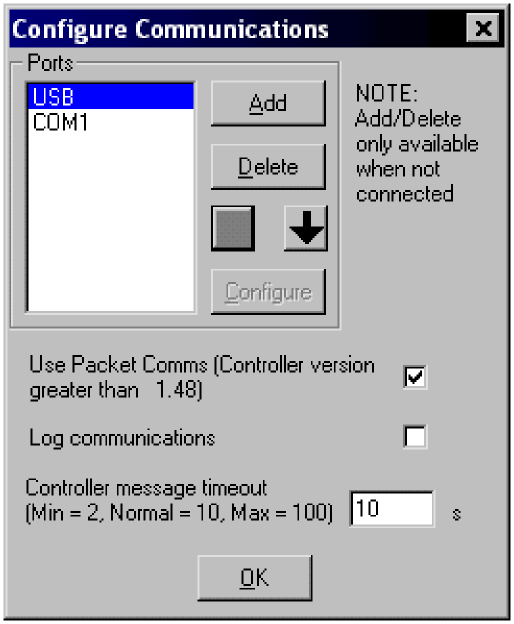
Stop bits: 1 [✓ 2]

Note: ***If you change the port setting to anything other than the default, you may encounter problems when the controller is reset because it will revert to the default values.***

In order to avoid this your controller will need to set the comms parameters within an auto-running program. See the SETCOM instruction in the Trio BASIC reference for further information.

Setting the Default Port

You can change the order in which the ports are scanned by the program by selecting the port in the list and using the up/down arrows to move the items as required.



NOTE: Add/Delete only available when not connected

Use Packet Comms (Controller version greater than 1.48) ✓

Log communications ☐

Controller message timeout (Min = 2, Normal = 10, Max = 100) 10 s

Packet Based Communications

Motion Perfect 2 features an enhanced communications protocol which uses a packet based structure with error checking to significantly improve the reliability of the serial port communications. Packet communications can be used over any type of communications link but as some of these (USB, ethernet) use packetised communications as part of their own protocol, adding an extra level of packetisation may make communications less efficient without giving any reliability improvement.

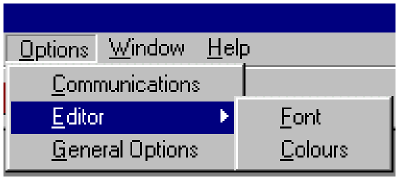
In order to use the Packet Comms mode you must have system software 1.49 or higher.

Controller message timeout

Certain controller operations may cause the controller to stop communicating for a few seconds. If you find that your PC seems to disconnect often, you can change *Motion* Perfect 2's default timeout value to allow the program to wait for a longer time before disconnecting.

Editor Options

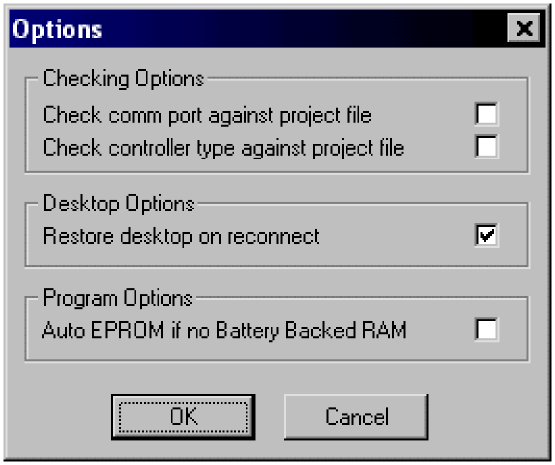
The Editor sub-menu allow you to modify the appearance of the *Motion* Perfect 2 editor to suit your own personal taste. You can change both the default font used and the colours used by the syntax highlighting feature.



General Options

This dialog allows the user to change a number of options relating to how *Motion* Perfect 2 starts up and handles projects.

When you select General Options you will be presented with the following screen.


Options

Checking Options
Check comm port against project file [ ]
Check controller type against project file [ ]

Desktop Options
Restore desktop on reconnect [✓]

Program Options
Auto EPROM if no Battery Backed RAM [ ]

The check-boxes enable the following features:

Check comm port against project file

Checks the comm port being used against the one in the project file when a Check Project operation is performed.

Check controller type against project file

Checks the type of the connected controllerg used against the one in the project file when a Check Project operation is performed.

Restore desktop on reconnect

If this option is selected, the program will attempt to automatically save the desktop layout when disconnecting from the controller.

When you reconnect, *Motion* Perfect 2 will automatically restore the last desktop layout saved.

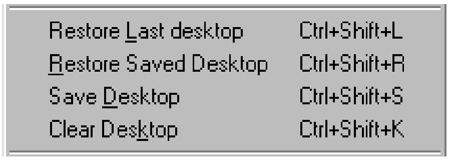
Auto EPROM if no Battery Backed RAM

If the controller does not feature battery-backed RAM memory and this option is selected, the program will attempt to automatically save the controller memory to EPROM before disconnecting from the controller.

Saving the Desktop Layout

When you have a number of windows open, you can save the layout so that it can be quickly restored later. Alternatively the desktop can be set to restore automatically on each re-connection by ticking the checkbox under the menu: Options/General options.

From the Window menu...


Restore Last desktop Ctrl+Shift+L
Restore Saved Desktop Ctrl+Shift+R
Save Desktop Ctrl+Shift+S
Clear Desktop Ctrl+Shift+K

Restore Last Desktop

Restores the last desktop which was automatically saved by *Motion* Perfect 2 when it disconnected from the controller

Restore Saved Desktop

Restore the last desktop saved using the Save Desktop option.

Save Desktop

Saves the current desktop layout to a file on disk

Clear Desktop

Closes all open tool windows.

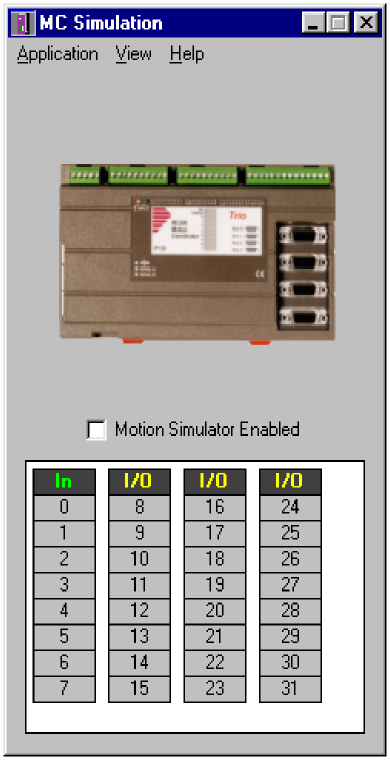
Running *Motion* Perfect 2 Without a Controller

Normally you will run *Motion* Perfect 2 on-line, that is connected to a controller. In fact *Motion* Perfect 2 is designed to operate in this manner and has little functionality without the connection.

In order that you can view or edit your project programs without a controller connected there is a special application to simulate the controller operation and to allow *Motion* Perfect 2 to operate in many ways as if a real controller were connected.

MC Simulation

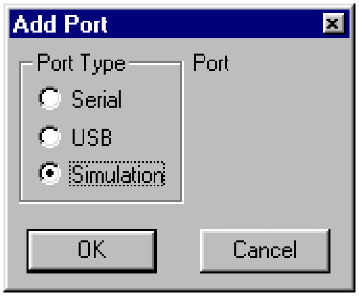
MC Simulation (MCSim) is a very simple program designed to run alongside *Motion* Perfect 2 in the background. There are no options or configurations to worry about, you just have to run the program and connect as usual.



In	I/O	I/O	I/O
0	8	16	24
1	9	17	25
2	10	18	26
3	11	19	27
4	12	20	28
5	13	21	29
6	14	22	30
7	15	23	31

Starting MC Simulatiion from *Motion* Perfect 2

MC Simulatoion is automatically started (if it is not already running) when *Motion* Perfect tries to connect to it. To connect to MCSimuolation either use the **Connect to Simulator** tool button or set up an **MCSim** port in the connection list.

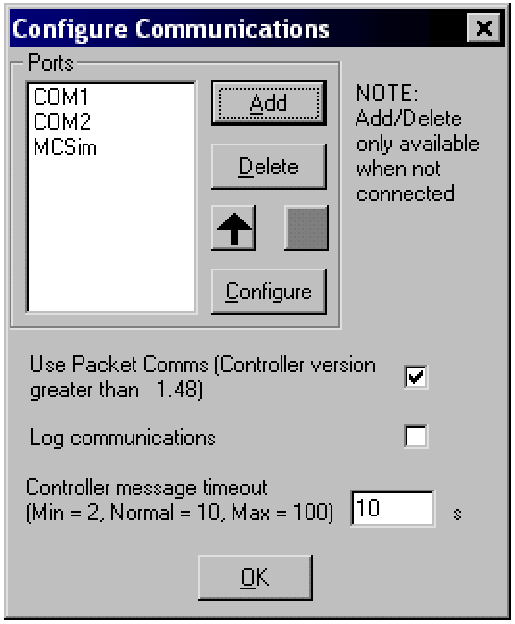

Add Port

Port Type: ( ) Serial, ( ) USB, [✓ Simulation]

Port

OK | Cancel

Use the Add Port option to select a new port and choose "Simulation" as the Port Type.


Configure Communications

Ports:
COM1
COM2
MCSim

Add | Delete | Configure

NOTE: Add/Delete only available when not connected

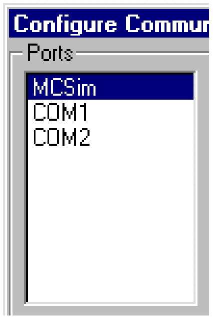
Use Packet Comms (Controller version greater than 1.48) [✓]

Log communications [ ]

Controller message timeout (Min = 2, Normal = 10, Max = 100): 10 s

OK

The new device will normally appear at the end of the list. Use the "move up" button to make it the default option.

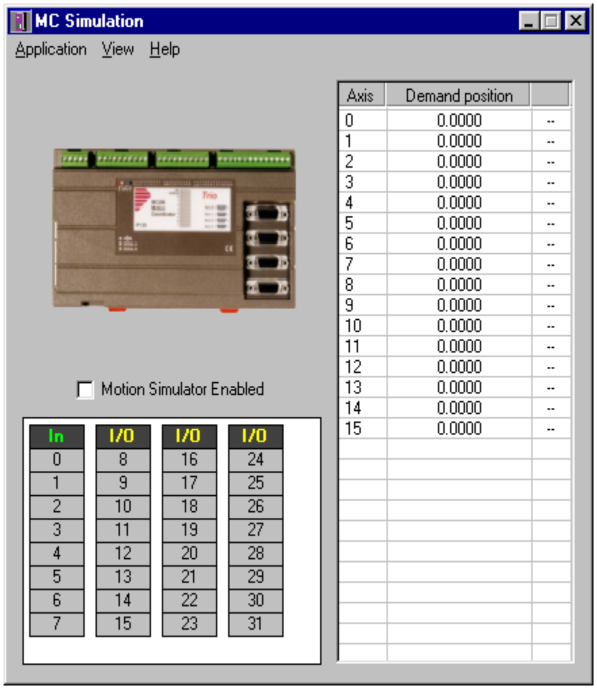

Configure Commun

Ports:
MCSim
COM1
COM2

Limitations of MC Simulation

The MCSimulation program does not yet cover all the functionality present in a real controller. It does allow connection to *Motion* Perfect for program editing and the running of programs in the simulated environment. There are some unsupported TrioBASIC commands (mainly those related to communications busses such as CAN).

The motion engine built into the simulation is still under development although it will handle all move types except linked moves. There is an axis demand position display which can be used to monitor the axes when moves are taking place. This can be toggled on and off by selecting **View/Axes** from the MCSimulation main menu. The motion engine can be enabled/disabled by checking/unchecking the the **Motion Simulator Enabled** check box.

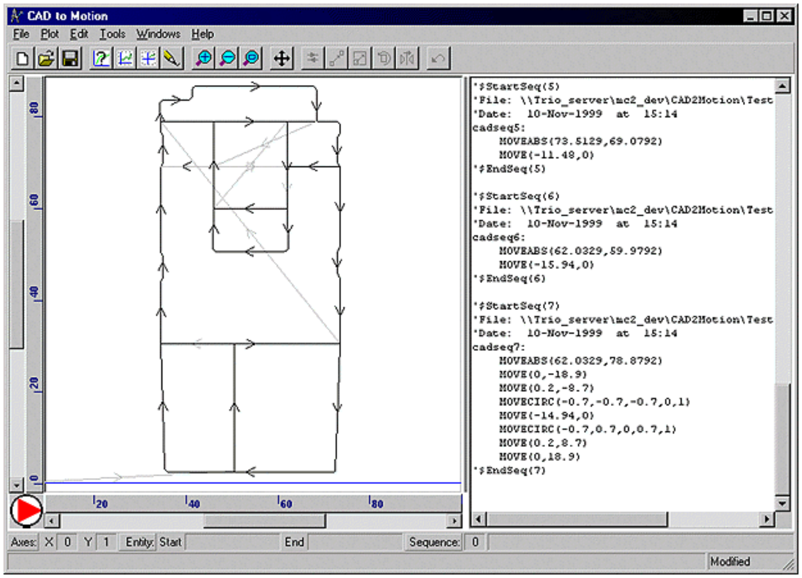


In	I/O	I/O	I/O
0	8	16	24
1	9	17	25
2	10	18	26
3	11	19	27
4	12	20	28
5	13	21	29
6	14	22	30
7	15	23	31

Axis	Demand position	
0	0.0000	--
1	0.0000	--
2	0.0000	--
3	0.0000	--
4	0.0000	--
5	0.0000	--
6	0.0000	--
7	0.0000	--
8	0.0000	--
9	0.0000	--
10	0.0000	--
11	0.0000	--
12	0.0000	--
13	0.0000	--
14	0.0000	--
15	0.0000	--

CAD2Motion

CAD2Motion is a program designed to allow users to translate CAD generated two dimensional motion paths into Trio BASIC programs.


CAD to Motion
File Plot Edit Tools Windows Help
80
60
40
20
0
20
40
60
80
Axes: X 0 Y 1 Entity: Start End Sequence: 0
Modified

```
'$StartSeq(5)
'File: \\Trio_server\mc2_dev\CAD2Motion\Test
'Date:  10-Nov-1999  at  15:14
cadseq5:
    MOVEABS(73.5129,69.0792)
    MOVE(-11.48,0)
'$EndSeq(5)

'$StartSeq(6)
'File: \\Trio_server\mc2_dev\CAD2Motion\Test
'Date:  10-Nov-1999  at  15:14
cadseq6:
    MOVEABS(62.0329,59.9792)
    MOVE(-15.94,0)
'$EndSeq(6)

'$StartSeq(7)
'File: \\Trio_server\mc2_dev\CAD2Motion\Test
'Date:  10-Nov-1999  at  15:14
cadseq7:
    MOVEABS(62.0329,78.8792)
    MOVE(0,-18.9)
    MOVE(0.2,-8.7)
    MOVECIRC(-0.7,-0.7,-0.7,0,1)
    MOVE(-14.94,0)
    MOVECIRC(-0.7,0.7,0,0.7,1)
    MOVE(0.2,8.7)
    MOVE(0,18.9)
'$EndSeq(7)
```

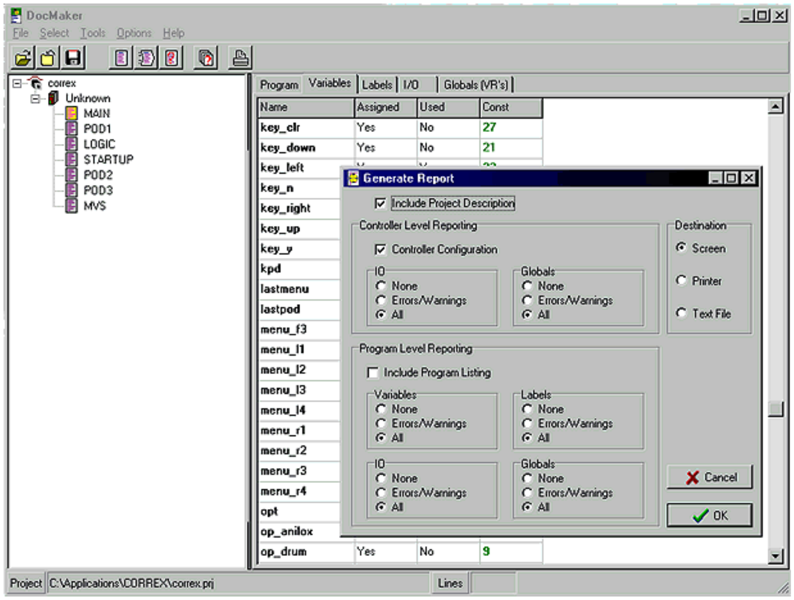
The program allows the user to create motion paths in a CAD package such as AutoCAD and convert them into code executable by a *Motion Coordinator*. Typically the path information will be drawn on a single layer in the CAD package and exported as a DXF file. The DXF file (layer with motion path only) is read into CAD2Motion to create a program to follow the motion path.

The motion path can be manipulated and edited before being saved as a Trio BASIC program file which can be loaded on to a *Motion Coordinator*.

- Can read Industry standard DXF files and TrioBASIC files.
- Outputs files as TrioBASIC programs.
- Graphical display of motion path with full pan and zoom facilities.
- Built in program text editor.
- Program list and graphical display linked together to show correlation between program code and moves within the motion path.
- Built in tools to mirror, scale, shift, reverse and rotate motion paths.
- Full undo on all tool and editor operations.
- Will handle multiple motion paths (sequences) in the same program.

DocMaker

DocMaker is a Windows application designed to assist in documenting a Trio BASIC project created with *Motion* Perfect.


DocMaker
File Select Tools Options Help
Program Variables Labels I/O Globals (VR's)
correx
Unknown
MAIN
POD1
LOGIC
STARTUP
POD2
POD3
MVS
Project C:\Applications\CORREX\correx.prj
Lines

Name	Assigned	Used	Const
key_clr	Yes	No	27
key_down	Yes	No	21
key_left			
key_n			
key_right			
key_up			
key_y			
kpd			
lastmenu			
lastpod			
menu_f3			
menu_l1			
menu_l2			
menu_l3			
menu_l4			
menu_r1			
menu_r2			
menu_r3			
menu_r4			
opt			
op_anilox			
op_drum	Yes	No	9

Generate Report

☑ Include Project Description

Controller Level Reporting
- ☑ Controller Configuration
- IO: None / Errors/Warnings / [✓ All]
- Globals: None / Errors/Warnings / [✓ All]

Destination: [✓ Screen] / Printer / Text File

Program Level Reporting
- ☐ Include Program Listing
- Variables: None / Errors/Warnings / [✓ All]
- Labels: None / Errors/Warnings / [✓ All]
- IO: None / Errors/Warnings / [✓ All]
- Globals: None / Errors/Warnings / [✓ All]

Cancel OK

DocMaker analyses the content of the program files in the project. It can be used to print program listings and to report on the programs (variables, labels, I/O and VR's) and on overall I/O and VR usage. There is also a checking routine which does a quick check on the whole project and flags up possible errors

DocMaker Benefits

- Automatic Analysis of MotionPerfect Project Files
- Highlight potential errors due to labels or variables
- Generates fully cross-referenced reports
- Reformat programs with auto-indenting

Docmaker Hardware Requirements

- IBM PC or Compatible running Microsoft Windows 95 or higher
- Works with all current Motion Coordinators: MC202, MC204, MC216, MC2 and Euro205

Project Autoloader

Trio Project Autoloader is a stand alone program to load projects created using Motion Perfect 2 onto a Trio Motion Coordinator.

The program is small enough to fit onto a 1.44MByte floppy disk and is intended for easy loading of projects onto controllers without the need to run Motion Perfect and so allows OEM manufacturers to update customers equipment easily.

Operation of the program is controlled using a script file which gives a series of commands to be processed, in order, by the program.

Using the Autoloader

General

The autoloader is primerally intended to be used from a floppy disk to update controllers already installed in equipment to allow OEM manufacturers to update customers equipment easily. It can also be used from a hard disk or CD-ROM.

Script File

The commands to be executed are held in a script file AutoLoader.tas which must be in the LoaderFiles directory.

Project

The project to be loaded using LOADPROJECT is in the form of a normal Motion Perfect 2 project. This consists of a directory containing a project definition file and TrioBASIC program files. The directory must have the same name as the project definition file less the extension.

i.e. project definition file TestProj.prj, directory TestProj

The project directory must be in the LoaderFiles directory.

Tables

Any tables to be loaded must be in the form of *.lst files produced by Motion Perfect 2.

Normally these table files will be in the LoaderFiles directory.

Extra Programs

Programs which need to be loaded using LOADPROGRAM because they are not in the project being loaded (or if no project is being loaded)

Normally these program files will be in the LoaderFiles directory.

Files

The autoloader is designed to work with the following file structure (fixed names are shown in bold type).

Base Directory	AutoLoader.exe		
	LoaderFiles	AutoLoader.tas	
		Table1.lst	
		ExtProg1.bas	
		Project	Project.prj
			Prog1.bas
			Prog2.bas

Where:

Base Directory is normally the root directory on a floppy disk (A:\), but can be any directory.

Project is the Motion Perfect 2 project directory for the project to be loaded using the LOADPROJECT command, Project.prj being the project file and Proj?.bas are the program files in the project.

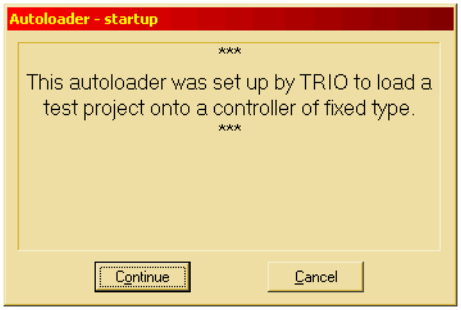
Table?.lst are the table files to be loaded using the LOADTABLE command.

ExtProg?.bas are the extra programs to be loaded using the LOADPROGRAM command.

Running the program

The program can be started in the same way as any other Windows program.

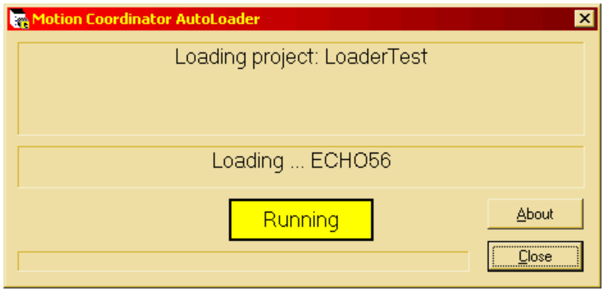
Start Dialog



***
This autoloader was set up by TRIO to load a test project onto a controller of fixed type.
***

The start dialog displays a message specified in the script and has continue and cancel buttons so that the user can exit from the program without running the script.

Main Window



Loading project: LoaderTest

Loading ... ECHO56

The program main window consists of two message windows; one to display the current command and the other to display the name of the program or file currently being loaded. There is a button to show the current status (Starting, running, pass or fail) and a progress bar to show the progress during file and table loading.

The close button closes the dialog. If it is pressed while a script is being processed then script processing will be terminated at the end of the current operation.

Script Commands

The following commands are available for use in script files

`COMMLINK`
`CHECKTYPE`
`CHECKVERSION`
`CHECKUNLOCKED`
`LOADPROGRAM`
`LOADPROJECT`
`EPROM`
`SETRUNFROMEPROM`
`LOADTABLE`
`Comment`

All commands return a result of OK or Fail. An OK result allows script execution to continue, a Fail result will make script execution terminate at that point.

COMMLINK

Purpose: To set the communications port and parameters.

Syntax: COMMLINK <PortSpec>

Where <PortSpec> is a string specifying a communications port and the connection parameters.

For a serial port this string is similar to COM1:9600,7,e,2 to specify the port, speed, number of data bits, parity and number of stop bits. 9600,7,e,2 are the default parameters for a controller.

Examples: COMMLINK COM2:9600,7,e,2

CHECKTYPE

Purpose: To check the controller type.

Syntax:
```
CHECKTYPE <Controller List>
```

Where <Controller List> is a comma separated list of one or more valid controller ID numbers.

i.e. 206,216

Examples:
```
CHECKTYPE 206
CHECKTYPE 202,216,206
```

Controller ID Numbers

Each type of controller returns a different ID number in response to the TrioBASIC command ?CONTROL[0] . The table below gives the ID number for current controllers.

Controller	ID Number
MC202	202
MC204	204
Euro205	205
MC206	206
MC216	216

CHECKVERSION

Purpose: To check the version of the controller system code.

Syntax:
```
CHECKVERSION <Operator><Version>
CHECKVERSION <LowVersion>-<HighVersion>
```

Examples:
```
CHECVERSION >1.49
CHECKVERSION >= 1.51
CHECKVERSION 1.42-1.50
```

CHECKUNLOCKED

Purpose: To check that the controller is not locked.

Syntax:
```
CHECKUNLOCKED
```

LOADPROJECT

Purpose: To load a project from disk onto the controller.

Syntax: `LOADPROJECT <ProjectName>`
Where <ProjectName> is the path of the project directory. If the project directory is in the LoaderFiles directory then it is just the name of the of the project directory.

Examples: `LOADPROJECT TestProj`

LOADPROGRAM

Purpose: To load a program which is not part of a project from disk onto the controller.

Syntax: `LOADPROGRAM <ProgramFile>`

Where <ProgramFile> is the path of the program file. If the program file is in the LoaderFiles directory then it is just the name of the of the program file.

Note: ***This command should always be used after the LOADPROJECT command*.**

Examples: `LOADPROGRAM TestProg.bas`

LOADTABLE

Purpose: `To load a table onto the controller.`

Syntax: `LOADTABLE <TableFile>`
Where <TableFile> is the path of the table file. If the table file is in the Loader-Files directory then this is just the file name of the table file.

This command should always be used after the LOADPROJECT command.

Examples: `LOADTABLE Tbl.lst`

EPROM

Purpose: To store the project currently in controller RAM into EPROM

Syntax: `EPROM`

SETRUNFROMEPROM

Purpose: To set the controller to use the programs stored in its EPROM. (It actually copies the programs from EPROM into RAM at startup).

Syntax: `SETRUNFROMEPROM <State>`

Where <State> is 1 for copy from EPROM and 0 is use programs currently in RAM.

A single @ character can be used to specify state in the project file.

Examples:

```
SETRUNFROMEPROM 1
SETRUNFROMEPROM @
```

Note: This command only applies to controllers which have battery backed RAM (controllers with no battery backed RAM will always copy programs from EPROM).

Script File

The autoloader program uses a script file AutoLoader.tas as a source of commands. These commands are executed in order until all commands have been processed or an error has occurred.

If any command fails the exececution terminates without completing the scripted command sequence.

Sample Script

```
' Test Script
' **************
' Startup Message
# ***
# This autoloader was set up by TRIO to load a test project
# onto a controller of fixed type.
# ***
COMMLINK COM1:9600,7,e,2
CHECKTYPE 206
CHECKVERSION > 1.45
CHECKUNLOCKED
LOADPROJECT LoaderTest
LOADTABLE tbl_1.lst
CHECKPROJECT LoaderTest
LOADPROGRAM flashop.bas
LOADPROGRAM clrtable.bas
LOADPROGRAM settable.bas
EPROM
SETRUNFROMEPROM @
```

For this script to work correctly the LoaderFiles directory must contain a project directory LoaderTest, a table file tbl_1.lst and three program files: flashop.bas, clrtable.bas and settable.bas.

Creating an AutoLoader floppy disk

To create an AutoLoader floppy disk follow the steps below:

- Take a blank floppy disk and copy the AutoLoader.exe program file into the root directory.
- Create a sub-directory called LoaderFiles in the root directory.
- If you are going to load a project then copy the project directory of the project to be loaded into the LoaderFiles directory created above, so that it is a sub-directory of LoaderFiles.
- If you are going to load any tables then copy all the table files into the LoaderFiles directory.
- If you are going to load any extra programs then copy the program files into the LoaderFiles directory.
- Using an ASCII text editor (i.e. Windows Notepad) create and edit the script file (AutoLoader.tas) in the LoaderFiles directory, using the appropriate commands for the operations to be carried out.

CHAPTER

11

TrioPC ActiveX Component Reference

Introduction

The TrioPC ActiveX component provides a direct connection to the Trio MC controllers via a PCI bus, USB or Ethernet link. It can be used in any windows programming language supporting ActiveX (OCX) components, such as Visual Basic, Delphi, Visual C, C++ Builder etc.

Requirements

- PC with one or more of USB interface, Ethernet network interface, or PCI based *Motion Coordinator*.
- Windows 98, ME, 2000 or XP (Windows 2000 or XP only for PCI connection)
- TrioUSB driver - for USB connection
- Trio PCI driver - for PCI connection (Windows 2000 and XP systems only)
- TrioPC OCX
- Knowledge of the Trio *Motion Coordinator* to which the TrioPC ActiveX controls will connect.
- Knowledge of the Trio BASIC programming language.

Installation of the ActiveX Component

Launch the program “Install_TrioPCMotion” and follow the on-screen instructions. The TrioUSB driver and TrioPC ocx will be installed and registered to your Windows environment. The Trio PCI driver will also be installed on systems running Windows 2000 or Windows XP. A Windows Help file is included as an alternative to the printed pages in this manual.

Using the Component

The TrioPC component must be added to the project within your programming environment. Here is an example using Visual Basic, however the exact sequence will depend on the software package used.

From the Menu select Project then Components... (or use shortcut ctrl+T).

When the Components dialogue box has opened, scroll down until you find “TrioPC ActiveX Control Module” then click in the block next to TrioPC. (A tick will appear)

Now click OK and the component should appear in the control panel on the left side of the screen. It is identified as TPC.

Once you have added the TrioPC component to your form, you are ready to build the project and include the TrioPC methods in your programs.

Connection Commands

Open

Description Initialises the connection between the TrioPC ActiveX control and the *Motion Coordinator*.

The connection can be opened over a PCI, USB or Ethernet link, and can operate in either a synchronous or asynchronous mode. In the synchronous mode all the TrioBASIC methods are available. In the asynchronous mode these methods are not available, instead the user must call SendData() to write to the *Motion Coordinator*, and respond to the OnReceiveChannelx event by calling GetData() to read data received from the *Motion Coordinator*. In this way the user application can respond to asynchronous events which occur on the *Motion Coordinator* without having to poll them.

If the user application requires the TrioBASIC methods then the synchronous mode should be selected. However, if the prime role of the user application is to respond to events triggered on the *Motion Coordinator,* then the asynchronous method should be used.

Syntax: `Open(PortType, PortMode)`

Parameters

`short PortType:`	0: USB, 1:N/A, 2:Ethernet, 3:PCI
`short PortMode:`	0: Synchronous Mode, 1:Asynchronous Mode, 3240:Synchronous Mode (for Ethernet connections only).

Return Value: `TRUE` if the connection is successfully established. For a USB connection, this means the TrioUSB driver is active (an MC with a USB card is on, and the USB connections are correct). If a synchronous connection has been opened the ActiveX control must have also successfully recovered the token list from the *Motion Coordinator*. If the connection is not successfully established this method will return `FALSE`.

Example

```
Rem Open a USB connection and refresh the TrioPC indicator
TrioPC_Status = TrioPC1.Open(0, 0)
frmMain.Refresh
```

Close

Description Closes the connection between the TrioPC ActiveX control and the *Motion Coordinator*

Syntax: `Close(PortMode)`

Parameters `short PortMode:` -1: all ports, 0: synchronous port, >1: asynchronous port

Return Value: `None.`

Example

```
Rem Close the connection when form unloads
Private Sub Form_Unload(Cancel As Integer)
    TrioPC1.Close(0)
    frmMain.Refresh
End Sub
```

IsOpen

Description Returns the state of the connection between the TrioPC ActiveX control and the *Motion Coordinator*

Syntax: `IsOpen(PortMode)`

Parameters `short PortMode:` -1: all ports, 0: synchronous port, >1: asynchronous port

Return Value: **`TRUE`** `if port is open,` **`FALSE`** `if it is closed.`

Example

```
Rem Close the connection when form unloads
Private Sub Form_Unload(Cancel As Integer)
    If TrioPC1.IsOpen(0) Then
        TrioPC1.Close(0)
    End If
    frmMain.Refresh
End Sub
```

SetHost

Description Sets the ethernet host IP address, and must be called prior to opening an ethernet connection. The **HostAddress** property can also be used for this function.

Syntax: `SetHost(host)`

Parameters `VARIANT host:` host IP address (eg 192.168.0.250).

Return Value: `None`

Example

```
Rem Set up the Ethernet IP Address of the target Motion
Coordinator
TrioPC1.SetHost("192.168.000.001")
Rem Open a Synchronous connection
TrioPC_Status = TrioPC1.Open(2, 0)
frmMain.Refresh
```

GetConnectionType

Description Gets the connection type of the current connection.

Syntax: `GetConnectionType()`

Parameters `None`

Return Value: -1: No Connection, 0: USB, 1:N/A, 2: Ethernet, 3: PCI

Example

```
Rem Open a Synchronous connection
ConnectError = False
TrioPC_Status = TrioPC1.Open(0, 0)
ConnectionType = TrioPC1.GetConnectionType()
If ConnectionType <> 0 Then
    ConnectError = True
frmMain.Refresh
```

Properties

Board

Description	Specifies the board number for a PCI connection. It must be specified before the **OPEN** command is used.
Type	Long
Access	Read / Write
Default Value	0

Example

```
Rem Open a PCI connection and refresh the TrioPC indicator
If TrioPC.Board <> 0 Then
    TrioPC.Board = 0
End If
TrioPC_Status = TrioPC1.Open(3, 0)
frmMain.Refresh
```

HostAddress

Description	Used for reading or changing the ethernet host IP address, and must be set prior to opening an ethernet connection. The **SetHost** command can also be used for setting the host adddress.
Type	String
Access	Read / Write
Default Value	"192.168.0.250"

Example

```
Rem Open a Ethernet connection and refresh the TrioPC indicator
if TrioPC.HostAddress <> "192.168.0.111" Then
    TrioPC.HostAddress = "192.168.0.111"
End If
TrioPC_Status = TrioPC1.Open(2, 0)
frmMain.Refresh
```

Motion Commands

MoveRel

Description	Performs the corresponding **`MOVE(…)`** command on the *Motion Coordinator*
Syntax:	**`MoveRel(Axes, Distance, [Axis])`**
Parameters:	

`short Axes:`	Number of axes involved in the move command
`VARIANT Distance:`	Distance to be moved, can be a single numeric value or an array of numeric values that contain at least Axes values
`VARIANT Axis:`	Optional parameters that must be a single numeric value that specifies the base axis for this move

Return Value: **`TrioPC STATUS.`**

Base

Description:	Performs the corresponding **`BASE(…)`** command on the *Motion Coordinator*
Syntax:	**`Base(Axes, [Order])`**
Parameters:	

`short Axes:`	Number of axes involved in the move command
`VARIANT Order:`	A single numeric value or an array of numeric values that contain at least Axes values that specify the axis ordering for the subsequent motion commands.

Return Value: **`TrioPC STATUS.`**

MoveAbs

Description: Performs the corresponding `MOVEABS(…) AXIS(…)` command on the *Motion Coordinator*

Syntax: `MoveAbs(Axes, Distance, [Axis])`

Parameters:

`short Axes:`	Number of axes involved in the moveabs command
`VARIANT Distance:`	Absolute positions that specify where the move must terminate, can be a single numeric value or an array of numeric values that contain at least Axes values
`VARIANT Axis:`	Optional parameters that must be a single numeric value that specifies the base axis for this move

Return Value: `TrioPC STATUS.`

MoveCirc

Description: Performs the corresponding MOVECIRC(…) AXIS(…) command on the *Motion Coordinator*

Syntax: `MoveCirc(EndBase, EndNext, CentreBase, CentreNext, Dir, [Axis])`

Parameters:

`double EndBase:`	Distance to the end position on the base axis
`double EndNext:`	Distance to the end position on the axis that follows the base axis
`double CentreBase:`	Distance to the centre position on the base axis
`double CentreNext:`	Distance to the centre position on the axis that follows the base axis
`short Dir:`	A numeric value that sets the direction of rotation. A value of 1 implies a clockwise rotation on a positive axis set, 0 implies an anti-clockwise rotation on a positive axis set.
`VARIANT Axis:`	Optional parameters that must be a single numeric value that specifies the base axis for this move

Return Value: `TrioPC STATUS.`

AddAxis

Description:	Performs the corresponding `ADDAX(…)` command on the *Motion Coordinator*	
Syntax:	`AddAxis(LinkAxis, [Axis])`	
Parameters:	`short LinkAxis:`	A numeric value that specifies the axis to be "added" to the base axis.
	`VARIANT Axis:`	Optional parameters that must be a single numeric value that specifies the base axis for this move
Return Value:	`TrioPC STATUS.`	

CamBox

Description: Performs the corresponding `CAMBOX(…)` command on the *Motion Coordinator*

Syntax:

```
CamBox(TableStart, TableStop, Multiplier, LinkDist, LinkAxis,
         LinkOpt, LinkPos, [Axis])
```

Parameters:

`short TableStart:`	The position in the table data on the *Motion Coordinator* where the cam pattern starts	
`short TableStop:`	The position in the table data on the *Motion Coordinator* where the cam pattern stops	
`double Multiplier:`	The scaling factor to be applied to the cam pattern	
`double LinkDist:`	The distance the input axis must move for the cam to complete	
`short LinkAxis:`	Definition of the Input Axis	
`short LinkOpt:`	**1**	link commences exactly when registration event occurs on link axis
	2	link commences at an absolute position on link axis (see param 7)
	4	CAMBOX repeats automatically and bi-directionally when this bit is set.
`double LinkPos:`	The absolute position on the link axis where the cam will start.	
`VARIANT Axis:`	Optional parameters that must be a single numeric value that specifies the base axis for this move	

Return Value: `TrioPC STATUS.`

Cam

Description Performs the corresponding `CAM(…) AXIS(…)` command on the *Motion Coordinator*

Syntax: `Cam(TableStart, TableStop, Multiplier, LinkDistance, [Axis])`

Parameters:

`short TableStart:`	The position in the table data on the *Motion Coordinator* where the cam pattern starts
`short TableStop:`	The position in the table data on the *Motion Coordinator* where the cam pattern stops
`double Multiplier:`	The scaling factor to be applied to the cam pattern
`double LinkDistance:`	Used to calculate the duration in time of the cam. The LinkDistance/Speed on the base axis specifies the duration. The Speed can be modified during the move, and will affect directly the speed with which the cam is performed
`VARIANT Axis:`	Optional parameters that must be a single numeric value that specifies the base axis for this move

Return Value: `TrioPC STATUS.`

Cancel

Description: Performs the corresponding `CANCEL(…) AXIS(…)` command on the *Motion Coordinator*

Syntax: `Cancel(Mode, [Axis])`

Parameters:

`short Mode:`	Cancel mode. 0 cancels the current move on the base axis, 1 cancels the buffered move on the base axis
`VARIANT Axis:`	Optional parameters that must be a single numeric value that specifies the base axis for this move

Return Value: `TrioPC STATUS.`

Connect

Description:	Performs the corresponding **`CONNECT(…) AXIS(…)`** command on the *Motion Coordinator*
Syntax:	**`Connect(Ratio, LinkAxis, [Axis])`**

Parameters:	**`double Ratio:`**	The gear ratio to be applied
	`short LinkAxis:`	The driving axis
	`VARIANT Axis:`	Optional parameters that must be a single numeric value that specifies the base axis for this move

Return Value: **`TrioPC STATUS`**.

Datum

Description: Performs the corresponding **`DATUM(…) AXIS(…)`** command on the *Motion Coordinator*

Syntax: **`Datum(Sequence, [Axis])`**

Parameters: **`short Sequence:`** The type of datum procedure to be performed:

0. The current measured position is set as demand position (this is especially useful on stepper axes with position verification). **`DATUM(0)`** will also reset a following error condition in the **`AXISSTATUS`** register for all axes.
1. The axis moves at creep speed forward till the Z marker is encountered. The Demand position is then reset to zero and the Measured position corrected so as to maintain the following error.
2. The axis moves at creep speed in reverse till the Z marker is encountered. The Demand position is then reset to zero and the Measured position corrected so as to maintain the following error.
3. The axis moves at the programmed speed forward until the datum switch is reached. The axis then moves backwards at creep speed until the datum switch is reset. The Demand position is then reset to zero and the Measured position corrected so as to maintain the following error.

4. The axis moves at the programmed speed reverse until the datum switch is reached. The axis then moves at creep speed forward until the datum switch is reset. The Demand position is then reset to zero and the Measured position corrected so as to maintain the following error.
5. The axis moves at programmed speed forward until the datum switch is reached. The axis then moves at creep speed until the datum switch is reset. The axis is then reset as in mode 2.
6. The axis moves at programmed speed reverse until the datum switch is reached. The axis then moves at creep speed forward until the datum switch is reset. The axis is then reset as in mode 1.

VARIANT Axis: Optional parameters that must be a single numeric value that specifies the base axis for this move

Return Value: `TrioPC STATUS.`

Forward

Description Performs the corresponding `FORWARD(…) AXIS(…)` command on the *Motion Coordinator*

Syntax: `Forward([Axis])`

Parameters **VARIANT Axis:** Optional parameters that must be a single numeric value that specifies the base axis for this move

Return Value: `TrioPC STATUS.`

Reverse

Description: Performs the corresponding `REVERSE(…) AXIS(…)` command on the *Motion Coordinator*

Syntax: `Reverse([Axis])`

Parameters: **VARIANT Axis:** Optional parameters that must be a single numeric value that specifies the base axis for this move

Return Value: `TrioPC STATUS.`

MoveHelical

Description Performs the corresponding `MOVEHELICAL(…) AXIS(…)` command on the *Motion Coordinator*

Syntax: `MoveHelical(FinishBase, FinishNext, CentreBase, CentreNext, Direction, LinearDistance, [Axis])`

Parameters

`double FinishBase:`	Distance to the finish position on the base axis
`double FinishNext:`	Distance to the finish position on the axis that follows the base axis
`double CentreBase:`	Distance to the centre position on the base axis
`double CentreNext:`	Distance to the centre position on the axis that follows the base axis
`short Direction:`	A numeric value that sets the direction of rotation. A value of 1 implies a clockwise rotation on a positive axis set, 0 implies an anti-clockwise rotation on a positive axis set.
`double LinearDistance:`	The linear distance to be moved on the base axis + 2 whilst the other two axes are performing the circular move
`VARIANT Axis:`	Optional parameters that must be a single numeric value that specifies the base axis for this move

Return Value: `TrioPC STATUS.`

MoveLink

Description: Performs the corresponding `MOVELINK(…) AXIS(…)` command on the *Motion Coordinator*

Syntax: `MoveLink(Distance, LinkDistance, LinkAcc, LinkDec, LinkAxis, LinkOptions, LinkPosn, [Axis])`

Parameters:

`double Distance:`	Total distance to move on the base axis
`double LinkDistance:`	Distance to be moved on the driving axis

`double LinkAcceleration`	Distance to be moved on the driving axis during the acceleration phase of the move
`double LinkDeceleration`	Distance to be moved on the driving axis during the deceleration phase of the move
`short LinkAxis:`	The driving axis for this move.
`short LinkOptions:`	Specifies special processing for this move: **0** no special processing **1** link commences exactly when registration event occurs on link axis **2** link commences at an absolute position on link axis (see param 7) **4** MOVELINK 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)
`double LinkPosition:`	The absolute position on the link axis where the move will start.
`VARIANT Axis:`	Optional parameters that must be a single numeric value that specifies the base axis for this move

Return Value: `TrioPC STATUS.`

MoveModify

Description Performs the corresponding `MOVEMODIFY(…) AXIS(…)` command on the *Motion Coordinator*

Syntax: `MoveModify(Position, [Axis]`

Parameters:

`double Position:`	Absolute position of the end of move for the base axis.
`VARIANT Axis:`	Optional parameters that must be a single numeric value that specifies the base axis for this move

Return Value: `TrioPC STATUS.`

RapidStop

Description:	Performs the corresponding `RAPIDSTOP(…)` command on the *Motion Coordinator*
Parameters:	None
Return Value:	`TrioPC STATUS`.

Process Control Commands

Run

Description: Performs the corresponding `RUN(…)` command on the *Motion Coordinator*

Syntax: `Run(Program, Process)`

Parameters:

`BSTR Program:`	String that specifies the name of the program to be run.
`VARIANT Process:`	Optional parameter that must be a single numeric value that specifies the process on which to run this program.

Return Value: `TrioPC STATUS`.

Stop

Description: Performs the corresponding `STOP(…)` command on the *Motion Coordinator*

Syntax: `Stop(Program, Process)`

Parameters:

`BSTR Program:`	String that specifies the name of the program to be stopped.
`VARIANT Process:`	Optional parameter that must be a single numeric value that specifies the process on which the program is running.

Return Value: `TrioPC STATUS`.

Variable Commands

GetTable

Description:	Retrieves and writes the specified table values into the given array.
Syntax:	`GetTable(StartPosition, NumberOfValues, Values)`
Parameters	`Long StartPosition:` Table location for first value in array
	`Long NumberOfValues:` Size of array to be transferred from Table Memory.
	`VARIANT Values:` A single numeric value or an array of numeric values, of at least size NumberOfValues, into which the values retrieved from the Table Memory will be stored.
Return Value:	`TrioPC STATUS.`

GetVariable

Description:	Returns the current value of the specified system variable. To specify different base axes, the `BASE` command must be used.
Syntax:	`GetVariable(Variable, Value)`
Parameters	`BSTR Variable:` Name of the system variable to read
	`double *Value:` Variable in which to store the value read
Return Value:	`TrioPC STATUS.`

GetVr

Description: Returns the current value of the specified Global variable.

Syntax: `GetVr(Variable, Value)`

Parameters:

`short Variable:`	Number of the VR variable to read.
`double *Value:`	Variable in which to store the value read.

Return Value: `TrioPC STATUS.`

SetTable

Description: Sets the specified table variables to the values given in an array.

Syntax: `SetTable(StartPosition, NumberOfValues, Values)`

Parameters

`Long StartPosition:`	Table location for first value in array
`Long NumberOfValues:`	Size of array to be transferred to Table Memory.
`VARIANT Values:`	A single numeric value or an array of numeric values that contain at least NumberOfValues values to be placed in the Table Memory.

Return Value: `TrioPC STATUS.`

SetVariable

Description: Sets the current value of the specified system variable. To specify different base axes, the `BASE` command must be used.

Syntax: `SetVariable(Variable, Value`

Parameters

`BSTR Variable:`	Name of the system variable to write
`double Value:`	Variable in which the value to write is stored.

Return Value: `TrioPC STATUS.`

SetVr

Description:	Sets the value of the specified Global variable.	
Syntax:	`SetVr(Variable, Value)`	
Parameters:	`BSTR Variable:`	Number of the VR variable to write
	`double Value:`	Variable in which the value to write is stored.
Return Value:	`TrioPC STATUS`.	

Input / Output Commands

Ain

Description:	Performs the corresponding `AIN(…)` command on the *Motion Coordinator*.	
Syntax:	`Ain(Channel, Value)`	
Parameters	`short Channel:`	AIN channel to be read.
	`double *Value:`	Variable in which to store the value read.
Return Value:	`TrioPC STATUS`.	

Get

Description:	Performs the corresponding `GET #`… command on the *Motion Coordinator*.	
Syntax:	`Get(Channel, Value)`	
Parameters	`short Channel:`	Comms channel to be read
	`double *Value:`	Variable in which to store the value read.
Return Value:	`TrioPC STATUS`.	

In

Description:	Performs the corresponding `IN(…)` command on the *Motion Coordinator*	
Syntax:	`In(StartChannel, StopChannel, Value)`	
Parameters:	`short StartChannel:`	First digital I/O channel to be checked.
	`short StopChannel:`	Last digital I/O channel to be checked.
	`long *Value:`	Variable to store the value read.

Return Value:	`TrioPC STATUS`.

Input

Description:	Performs the corresponding `INPUT #`... command on the *Motion Coordinator*.	
Syntax:	`Input(Channel, Value)`	
Parameters:	`short Channel:`	Comms channel to be read
	`double *Value:`	Variable in which to store the value read.
Return Value:	`TrioPC STATUS`.	

Key

Description	Performs the corresponding `KEY #`... command on the *Motion Coordinator*.	
Syntax:	`Key(Channel, Value)`	
Parameters	`short Channel:`	Comms channel to be read
	`double *Value:`	Variable in which to store the value read.
Return Value:	`TrioPC STATUS`.	

Linput

Description:	Performs the corresponding `LINPUT #` command on the *Motion Coordinator*.	
Syntax:	`Linput(Channel, Startvr)`	
Parameters:	`short Channel:`	Comms channel to be read
	`short StartVr:`	Number of the VR variable into which to store the first key press read.
Return Value:	`TrioPC STATUS`.	

Op

Description:	Performs the corresponding `OP(…)` command on the *Motion Coordinator*	
Syntax:	`Op(Output, State)`	
Parameters:	`VARIANT Output:`	Numeric value. If this is the only value specified then it is the bit map of the outputs to be specified, otherwise it is the number of the output to be written.
	`VARIANT State:`	Optional numeric value that specifies the desired status of the output, 0 implies off, not-0 implies on.
Return Value:	`TrioPC STATUS`.	

Pswitch

Description Performs the corresponding `PSWITCH(…)` command on the *Motion Coordinator*

Syntax: `Pswitch(Switch, Enable, Axis, OutputNumber, OutputStatus, SetPosition, ResetPosition)`

Parameters:

`short Switch:`	Switch to be set
`short Enable:`	1 to enable, 0 to disable
`VARIANT Axis:`	Optional numeric value that specifies the base axis for this command
`VARIANT OutputNumber:`	Optional numeric value that specifies the number of the output to set
`VARIANT OutputStatus:`	Optional numeric value that specifies the signalled status of the output, 0 implies off, not-0 implies on.
`VARIANT SetPosition:`	Optional numeric value that specifies the position at which to signal the output
`VARIANT ResetPosition:`	Optional numeric value that specifies the position at which to reset the output.

Return Value: `TrioPC STATUS`.

ReadPacket

Description: Performs the corresponding `READPACKET(…)` command on the *Motion Coordinator*

Syntax: `ReadPacket(PortNumber, StartVr, NumberVr, Format)`

Parameters:

`short PortNumber:`	Number of the comms port to read (0 or 1).
`short StartVr:`	Number of the first variable to receive values read from the comms port.
`short NumberVr:`	Number of variables to receive.
`short Format:`	Numeric format in which the numbers will arrive

Return Value: `TrioPC STATUS`.

Record

Description: Performs the corresponding `RECORD(…)` command on the *Motion Coordinator*

Syntax: `Record(Transitions, TablePosition)`

Parameters:

`short Transitions:`	Number of transitions to record.
`long TablePosition:`	Start position in the table to store the transitions.

Return Value: `TrioPC STATUS`.

Regist

Description Performs the corresponding `REGIST(…)` command on the *Motion Coordinator*

Syntax: `Regist(Mode, Dist)`

Parameters: `short Mode:` Registration mode

1. Axis absolute position when Z Mark Rising
2. Axis absolute position when Z Mark Falling
3. Axis absolute position when Registration Input Rising
4. Axis absolute position when Registration Input Falling
5. Sets pattern recognition mode

`double Dist:` Only used in pattern recognition mode and specifies the distance over which to record the transitions.

Return Value: `TrioPC STATUS`.

Send

Description:	Performs the corresponding `SEND(…)` command on the *Motion Coordinator*	
Syntax:	`Send(Destination, Type, Data1, Data2)`	
Parameters:	`short Destination:`	Address to which the data will be sent
	`short Type:`	Type of message to be sent: 1. Direct variable transfer 2. Keypad offset
	`short Data1:`	Data to be sent. If this is a keypad offset message then it is the offset, otherwise it is the number of the variable on the remote node to be set.
	`short Data2:`	Optional numeric value that specifies the value to be set for the variable on the remote node.
Return Value:	`TrioPC STATUS.`	

Setcom

Description:	Performs the corresponding `SETCOM(…)` command on the *Motion Coordinator*	
Syntax	`Setcom(Baudrate, DataBits, StopBits, Parity, [Port], [Control])`	
Parameters:	`long BaudRate:`	Baud rate to be set
	`short DataBits:`	Number of bits per character transferred
	`short StopBits:`	Number of stop bits at the end of each character
	`short Parity:`	Parity mode of the port (0=>none, 1=>odd, 2=> even)
	`VARIANT Port:`	Optional numeric value that specifies the port to set (0..3)
	`VARIANT Control:`	Optional numeric value that specifies whether to enable or disable handshaking on this port
Return Value:	`TrioPC STATUS.`	

General commands

Execute

Description:	Performs the corresponding **`EXECUTE`** … command on the *Motion Coordinator*.
Syntax:	**`Execute(Command)`**
Parameters	**`BSTR Command:`** String that contains a valid Trio BASIC command
Return Value:	`TrioPC STATUS:` **`TRUE`** if the command was sent successfully to the *Motion Coordinator* and the **`EXECUTE`** command on the *Motion Coordinator* was completed successfully and the command specified by the **`EXECUTE`** command was tokenised, parsed and completed successfully.

GetData

Description	This method is used when an asynchronous connection has been opened, to read data received from the *Motion Coordinator* over a particular channel. The call will empty the appropriate channel receive data buffer held by the ActiveX control.	
Syntax:	**`GetData(channel, data)`**	
Parameters	**`short channel:`**	Channel over which the required data was received (5,6,7, or 9).
	`VARIANT data:`	data received by the control from the *Motion Coordinator*
Return Value:	`TrioPC STATUS:` TRUE - if the given channel is valid, the connection open and the data read correctly from the buffer.	

SendData

Description	This method is used when the connection has been opened in the asynchronous mode, to write data to the *Motion Coordinator* over a particular channel.	
Syntax:	`SendData(channel, data)`	
Parameters	`short channel:`	channel over which to send the data (5,6,7, or 9).
	`VARIANT data:`	data to be written to the *Motion Coordinator*
Return Value:	`TrioPC STATUS:` TRUE - if the given channel is valid, the connection open, and the data written out correctly.	

Events

OnBufferOverrunChannel5/6/7/9

Description: One of these events will fire if a particular channel data buffer overflows. The ActiveX control stores all data received from the *Motion Coordinator* in the appropriate channel buffer when the connection has been opened in asynchronous mode. As data is received it is the responsibility of the user application to call the GetData() method whenever the OnReceiveChannelx event fires(or otherwise to call the method periodically) to prevent a buffer overrun. Which event is fired will depend upon which channel buffer overran.

Syntax: `OnBufferOverrunChannelx()`

Parameters: `None.`

Return Value: `None.`

OnReceiveChannel5/6/7/9

Description: One of these events will fire when data is received from the *Motion Coordinator* over a connection which has been opened in the asynchronous mode. Which event is fired will depend upon over which channel the *Motion Coordinator* sent the data. It is the responsibility of the user application to call the GetData() method to retrieve the data received.

Syntax: `OnReceiveChannelx()`

Parameters: `None.`

Return Value: `None.`

TrioPC status

Many of the methods implemented by the TrioPC interface return a boolean status value. The value will be **`TRUE`** if the command was sent successfully to the *Motion Coordinator* and the command on the *Motion Coordinator* was completed successfully. It will be **`FALSE`** if it was not processed correctly, or there was a communications error.

Appendix 1

Reference

ATYPE

#	Description
0	No axis daughter board fitted *
1	Stepper Axis *
2	Servo Axis *
3	Encoder Reference Axis *
4	Stepper with internal feedback into encoder registers *
5	Resolver Axis
6	Voltage output
7	Absolute SSI Servo
8	CAN daughter board
9	Remote CAN axis
10	PSWITCH Axis
11	Remote DriveLink axis
12	Reserved
13	Embedded axis
14	Encoder Output
15	Reserved
16	Remote SERCOS speed axis
17	Remote SERCOS position axis
18	Remote CanOpen position axis *
19	Remote CanOpen speed axis *
20	Reserved
21	Remote User Specific CAN axis

* Only these ATYPE's are currently relevant on the PCI 208 *Motion Coordinator*

COMMSTYPE

#	Description
26	P184 4 Axis DAC Module in expansion slot
27	P185 8 Axis DAC Module in expansion slot
24	SERCOS Module in expansion slot

AXISSTATUS / ERRORMASK

Bit	Description:	Value:
0	Unused	1
1	Following error warning range	2
2	Communications error to remote drive	4
3	Remote drive error	8
4	In forward limit	16
5	In reverse limit	32
6	Datuming	64
7	Feedhold	128
8	Following error exceeds limit	256
9	In forward software limit	512
10	In reverse software limit	1024
11	Cancelling move	2048
12	Encoder power supply overload	4096
13	Set on SSI axis after initialisation	8192

CONTROL

Controller	CONTROL system parameter
Motion Coordinator MC202	202
Motion Coordinator Euro205	205
Motion Coordinator MC206	206
Motion Coordinator PCI 208	208
Motion Coordinator MC216	216
Motion Coordinator MC224	224

Communications Ports

Chan	**Device:-**
0	Port 0 - Motion Perfect / Command Line
1	Unused on PCI 208
2	Unused on PCI 208
3	Unused on PCI 208
4	Unused on PCI 208
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	Unused on PCI 208

MTYPE

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

Data Formats and Floating-Point Operations

The TMS320C3x processor used by the *Motion Coordinator* features several different data types. In the *Motion Coordinator* we use two main formats. The following descriptions are taken directly from the TI documentation.

Single-Precision Floating Point Format

In the single precision format, the floating-point number is represented by an 8-bit exponent field (e) and a twos complement 24-bit mantissa field (man) with and implied significant non-sign bit.

31 24	23	22 0
e	s	f

Operations are performed with an implied binary point between bits 23 and 22. When the implied most significant non-sign bit is made explicit, it is located to the immediate left of the binary point.

The floating point number *'x'* is given by:

$$x = \begin{cases} 01.f \times 2^{e} & \text{if } s=0 \\ 10.f \times 2^{e} & \text{if } s=1 \\ 0 & \text{if } e=-128 \end{cases}$$

The following examples illustrate the range and precision if the single-precision floating-point format:

Most Positive:	$x = (2 - 2^{-23}) \times 2^{127}$	$= 3.4028234 \times 10^{38}$
Least Positive:	$x = 1 \times 2^{-127}$	$= 5.8774717 \times 10^{-39}$
Least Negative:	$x = (-1 - 2^{-23}) \times 2^{-127}$	$= -5.8774724 \times 10^{-39}$
Most Negative:	$x = -2 \times 2^{127}$	$= -3.4028236 \times 10^{38}$

Single-Precision Integer Format

In the single precision integer format, the integer is represented in twos complement notation.

31	0
s	

The range of an integer *x*, represented in the single-precision integer format, is:

$$-2^{31} <= x <= 2^{31} - 1$$

Product Codes

Processors	
P180	PCI 208

Options -	
P181	Breakout board
P187	2.5 m Breakout cable
P182	Additional stepper axis
P183	Additional servo axis
P315	Can 16 I/O expansion module
P325	Can analog 8 input module

Plug-in Option Modules	
P184	4 Axis DAC module
P185	8 Axis DAC module

INDEX