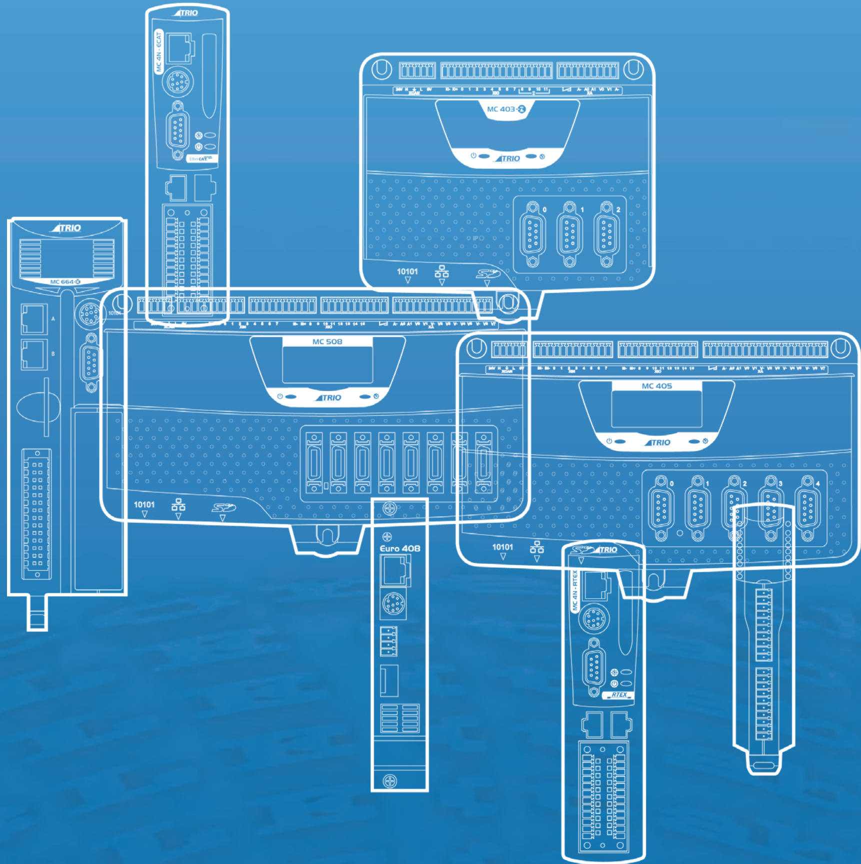


Motion Coordinator - 4xx Range

HARDWARE REFERENCE MANUAL
Version 7.6

SOFTWARE REFERENCE MANUAL
(page152)Version 7.5

Trio Motion Technology

Motion Coordinator 4xx Range

Hardware Reference Manual

Seventh Edition • 2016
Revision 6

UK
Trio Motion Technology Ltd.
Phone: +44 (0)1684 292333
Fax: +44 (0)1684 297929

USA
Trio Motion Technology LLC.
Phone: + 1 724 472 4100
Fax: +1 724 472 4101

CHINA
Trio Shanghai
Tel: +86 21 587 976 59
Fax: +86 21 587 942 89

INDIA
Trio India
Phone: +91 827 506 5446

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.

Users must ensure that in all cases of normal operation, controller malfunction, or unexpected behaviour, the safety of operators, programmers or any other person is totally ensured.

This manual uses the following icons for your reference:

Information that relates to safety issues and critical software information	Information to highlight key features or methods.	Useful tips and techinques.

Contents

Contents

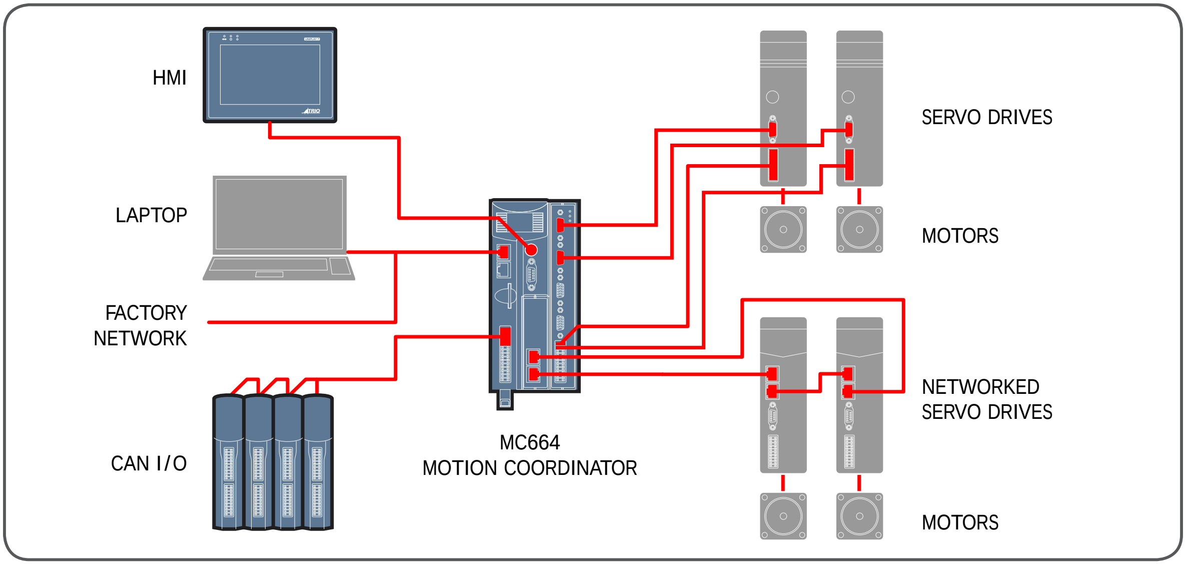
1 INTRODUCTION

Introduction to the MC4xx Range

The MC4 range *Motion Coordiantors* are the latest in the Trio pedigree representing over 25 years of motion control experience. Run your machine faster and with greater precision with these new generation *Motion Coordinators* based on a 64 bit technology.

Choose the motor and drives to best suit your application without compromise, the MC4xx range provides interface options for traditional servo, stepper and piezo control together with many digital interfaces for current digital servo drives. Increase the flexibility of your equipment with support for up to 64 axes of motion control. Trio's tradition of modular configuration has evolved into convenient MC464 clip-on modules allowing the system designer to precisely build the configuration needed for the job.

The MC405 and MC403 share the same advanced software and hardware techniques with the MC464, but come in 2 compact and cost-effective packages for machine applications requiring lower axis counts.



Typical System Configuration

TYPICAL SYSTEM CONFIGURATION

The MC4xx range supports programs written in TrioBASIC, allowing a smooth upgrade path from earlier types of *Motion Coordinator*. In addition, the standard IEC 61131-3 languages are supported, allowing both logical I/O and motion programming in Ladder, Function Block, Structured Text and Sequential Function Chart. A rich set of motion function blocks allows the programmer to have full access to the familiar Trio Motion command set.

I/O expansion is provided via a built-in CANbus interface. The built-in Ethernet port supports both the programming interface and many Ethernet based fieldbuses. These can be used simultaneously. Further fieldbus networks supporting common factory protocols are supported in the MC464 via the HMS AnyBus® adapter module.

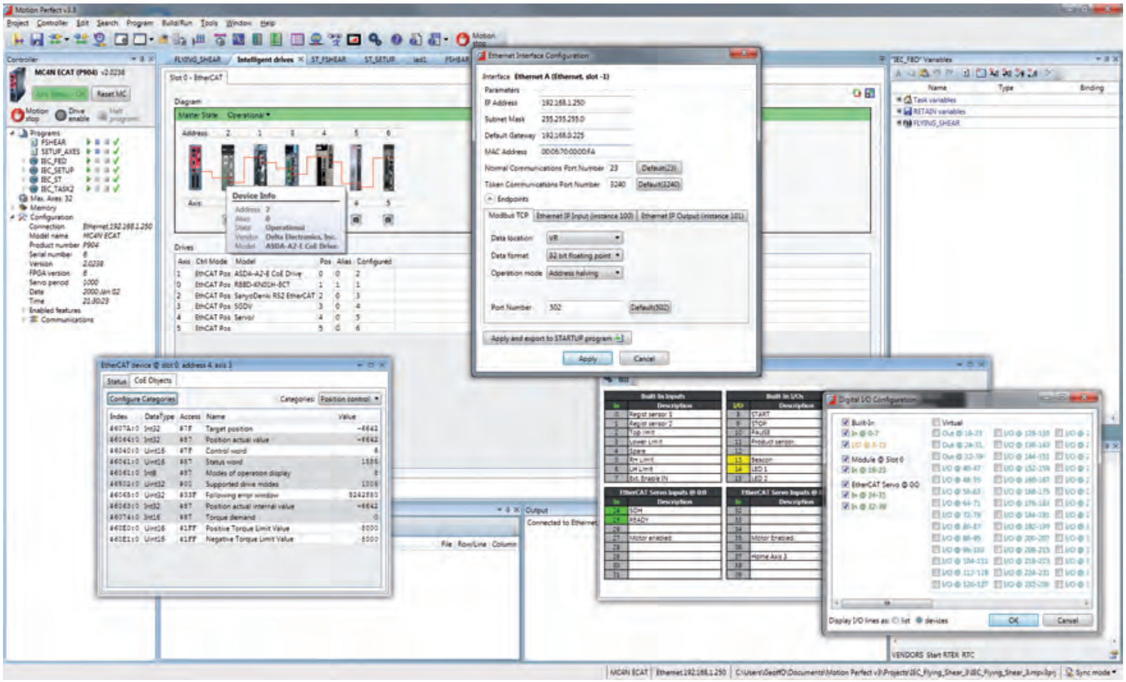
The MC664 and MC464 axis expansion modules feature many options for Drive Network interfaces, analogue servo, pulse/direction, absolute or incremental feedback and accurate hardware registration. Up to 7 half-height expansion modules or 3 full-height expansion modules can be attached. This modular approach along with Trio's feature enable code system for axis activation allows the whole system to be scaled exactly to need.

The MC4N-ECAT is dedicated to running remote servo and stepper drives via the EtherCAT real time automation bus. The MC4N-RTEX runs Panasonic Real Time EXpress drives. Versions of the MC4N-ECAT and MC4N-RTEX are available for 2, 4, 8, 16 and 32 motor axes

The MC403 and MC405 each come in 2 main variants; either 3 or 5 axis pulse+direction output, or as 2 or 4 axis servo with a single 5th axis encoder port.

SETUP AND PROGRAMMING

To program the *Motion Coordinator*, a PC is connected via an Ethernet link. The dedicated *Motion* Perfect program is normally used to provide a wide range of programming facilities on a PC running Microsoft Windows XP, Vista or Windows 7 versions.



Motion Perfect 3

Once connected to the *Motion Coordinator*, the user has direct access to TrioBASIC which provides an easy and 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, synchronised multi axis moves and unsynchronised multi axis moves as well as the control of the digital I/O.

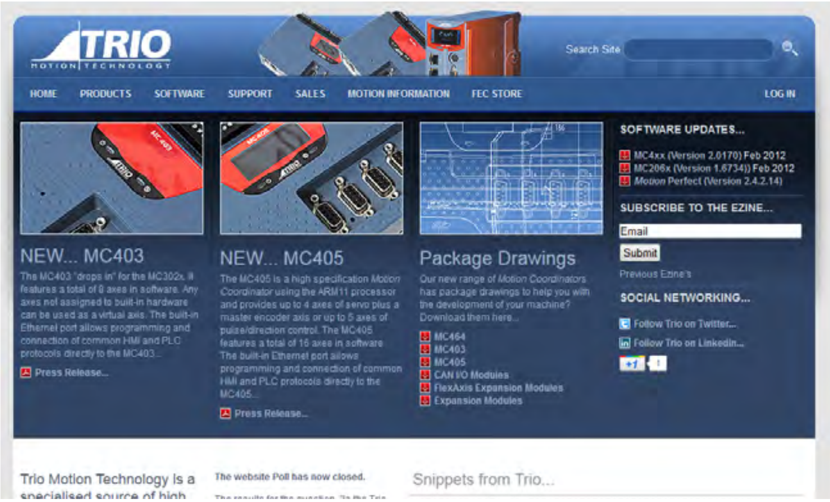
The MC4 xx range features multi-tasking TrioBASIC and the standard IEC 61131-3 language. Multiple TrioBASIC programs plus Ladder Diagram (LD), Function Block (FB), Structured Text (ST) and Sequential Function Chart (SFC) can be constructed and run simultaneously to make programming complex applications much easier. *Motion* Perfect version 3 is needed to access the full IEC 61131-3 functionality. MPv3 provides a seamless programming, compilation and debug environment that can work in real-time with any of the MC4 range *Motion Coordinator*s. A motion library is provided which enables the familiar Trio Motion Technology commands to be included in IEC 61131-3 programs.

FEATURES

- Supports digital drive systems up to 128 axis
- Based on 64bit MIPS and ARM processor technology
- 64bit position integers
- High accuracy double floating point resolution
- Multi-tasking BASIC programming
- IEC61131-3 programming support
- Motion buffers up to 64 moves
- Robotics, gears, interpolation and synchronisation built-in
- I/O expansion up to 528 I/O points
- Ethernet programming interface
- Backlit LCD display (MC664, MC464, MC4N MC508 and MC405)
- Expansion flexibility with clip on modules allowing quick interchangibility (MC664 and MC464)
- Anybus Module support allowing flexible factory communication options (MC664 and MC464)

THE TRIO MOTION TECHNOLOGY WEBSITE

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



- Website Features
- Latest News
- Product Information
- Manuals
- Programming Tools
- System Software Updates
- Technical Support
- User’s Forum
- Application Examples
- Employment Opportunities

WWW.TRIOMOTION.COM

2

HARDWARE OVERVIEW

Hardware

Motion Coordinator MC664 (-X)

OVERVIEW

The *Motion Coordinator* MC664 is Trio's highest specification modular servo control positioner with the ability to control servo or stepper motors by means of Digital Drive links (e.g. EtherCAT, RTEX, etc) or via traditional analogue and encoder or pulse and direction. A maximum of 7 expansion modules can be fitted to control up to 128 axes which gives the flexibility required in modern system design. The MC664 is housed in a rugged plastic case with integrated earth chassis and incorporates all the isolation circuitry necessary for direct connection to external equipment in an industrial environment. Filtered power supplies are included so that it can be powered from the 24V d.c. logic supply present in most industrial cabinets.

It is designed to be configured and programmed for the application using a PC running the *Motion* Perfect application software, and then may be set to run "standalone" if an external computer is not required for the final system.

There are two versions of the MC664. A single core processor allowing the MC664 to replace the MC464 in most applications. The MC664-X includes a quad-core A9 processor and is recommended for high performance applications such as robotics and for systems with large numbers of axes.

The Multi-tasking version of TrioBASIC for the MC664 allows up to 22 TrioBASIC programs to be run simultaneously on the controller using pre-emptive multi-tasking. In addition, the operating system software includes the IEC 61131-3 standard run-time environment (licence key required).

PROGRAMMING

The Multi-tasking ability of the MC664 allows parts of a complex application to be developed, tested and run independently, although the tasks can share data and motion control hardware. IEC 61131-3 programs can be run at the same time as TrioBASIC allowing the programmer to select the best features of each. The MC664-X runs applications and motion in seperate cores for increased performance.

I/O CAPABILITY

The MC664 has 8 built-in 24V inputs and 8 bi-directional I/O channels. These may be used for system interaction or may be defined to be used by the controller for end of travel limits, registration, datuming and feedhold functions if required. Each of the Input/Output channels has a status indicator to make it easy to check them at a glance. The MC664 can have up 512 external Input/Output channels connected using DIN rail mounted CAN I/O modules. These units connect to the built-in CAN channel. In addition, the built-in EtherCAT port can support up to 1024 I/O points.

COMMUNICATIONS

A 10/100 base-T Ethernet port is fitted as standard and this is the primary communications connection to the MC664. Many protocols are supported including Telnet, Modbus TCP, Ethernet IP and TrioPCMotion. Check the Trio website (www.triomotion.com) for a complete list.

The MC664 has one built in RS232 port and one built in duplex RS485 channel for simple factory communication systems. Either the RS232 port or the RS485 port may be configured to run the Modbus or Hostlink protocol for PLC or HMI interfacing.

If the built-in CAN channel is not used for connecting I/O modules, it may optionally be used for CAN communications. E.g. DeviceNet slave or CanOpen master.

A second RJ45 socket is enabled for precisely timed EtherCAT communication with drives and I/O devices.

The Anybus CompactCom Carrier Module (P875) can be used to add other fieldbus communications options

REMOVABLE STORAGE

The MC664 has a SD Card slot which allows a simple means of transferring programs, firmware and data without a PC connection. Offering the OEM easy machine replication and servicing.

The memory slot is compatable with a wide range of SD cards up to 16Gbytes using the FAT32 compatible file system.

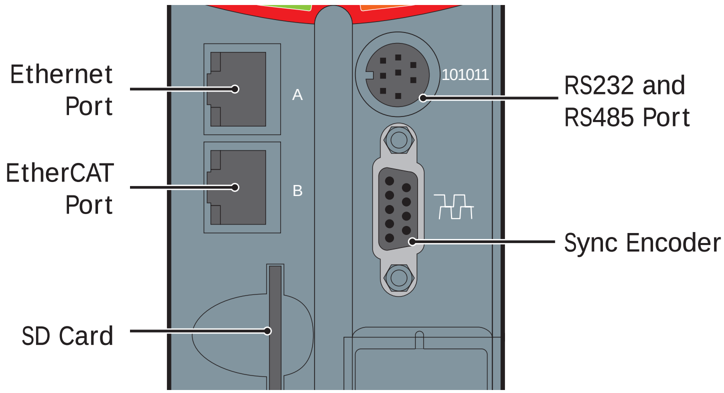
AXIS POSITIONING FUNCTIONS

The motion control generation software receives instructions to move an axis or axes from the TrioBASIC or IEC 61131-3 language which is running concurrently on the same processor. The motion generation software provides control during operation to ensure smooth, coordinated movements with the velocity profiled as specified by the controlling program. Linear interpolation may be performed on groups of axes, and circular, helical or spherical interpolation in any two/three orthogonal axes. Each axis may run independently or they may be linked in any combination using interpolation, CAM profile or the electronic gearbox facilities.

Consecutive movements may be merged to produce continuous path motion and the user may program the motion using programmable units of measurement (e.g. mm, inches, revs etc.). The module may also be programmed to control only the axis speed. The positioner checks the status of end of travel limit switches which can be used to cancel moves in progress and alter program execution.

The processing power of the MC664 allows real-time robotic transforms to be run which convert world coordinates into the required motor angles. Many typical mechanical arrangements are handled including Scara, Delta, complex “wrist” and 6 degrees of freedon (D.O.F).

CONNECTIONS TO THE MC664

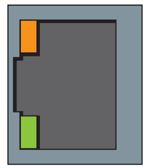


ETHERNET PORT CONNECTION

Physical layer: 10/100 base-T

Connector: RJ45

The Ethernet port is the default connection between the *Motion Coordinator* and the host PC running *Motion* Perfect programming.



ETHERCAT PORT

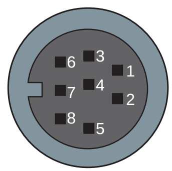
EtherCAT master port for connection to servo/stepper drives and I/O devices using industry standard EtherCAT protocols.

MC664 SERIAL CONNECTIONS

The MC664 features two serial ports. Both ports are accessed through a single 8 pin connector.

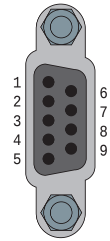
SERIAL CONNECTOR

Pin	Function	Note
1	RS485 Data In A Rx+	Serial Port #2
2	RS485 Data In B Rx-	
3	RS232 Transmit	Serial Port #1
4	0V Serial	
5	RS232 Receive	Serial Port #1
6	Internal 5V	5V supply is limited to 150mA, shared with sync port
7	RS485 Data Out Z Tx-	Serial Port #2
8	RS485 Data Out Y Tx+	Serial Port #2



SYNC ENCODER

The sync encoder port is bidirectional. It can be used as a reference encoder input or as an encoder simulation output to act as a master reference for other parts of the system.

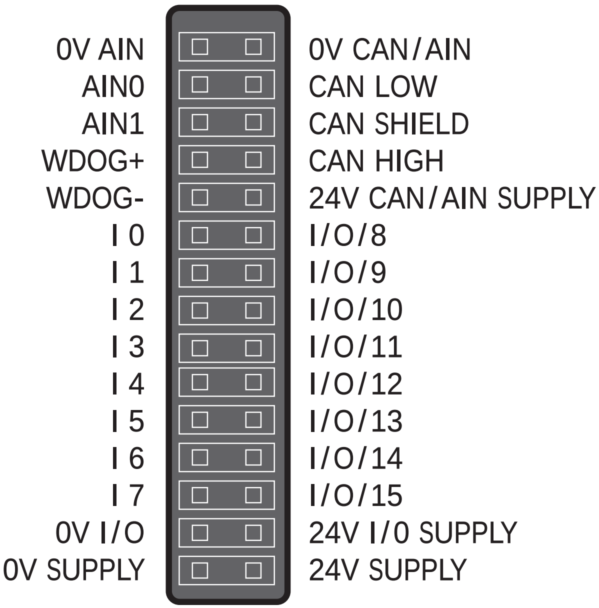


Pin	Encoder	Absolute	Pulse & Direction
1	Enc. A	Clk +	Step +
2	Enc. /A	Clk -	Step -
3	Enc. B	N/C	Direction +
4	Enc. /B	N/C	Direction -
5	0V Encoder	0V Enc.	0V Stepper
6	Enc. Z	Data +	Enable +
7	Enc. /Z	Data -	Enable -
8	5V*	5V	5V*
9	Registration Input (5V)		
*5V supply is limited to 150mA (shared with serial port)			

REGISTRATION

The MC664 built in port has 2 available registration events. These can be used with the Z mark, the registration input on the sync port, or up to 2 inputs of the MC664 digital inputs 0 - 7, mapped by **REG _ INPUTS**.

24V POWER SUPPLY INPUT



The MC664 is powered entirely via the 24V d.c.supply connections. The unit uses internal DC-DC converters to generate independent 5V logic supply, the encoder/serial 5V supply and other internal power supplies. I/O, analogue and CANbus circuits are isolated from the main 24V power input and must be powered separately. For example; it is often necessary to power the CANbus network remotely via the CANbus cable.

24V d.c., Class 2 transformer or power source required for **UL** compliance. The MC664 is grounded via the metal chassis. It MUST be installed on an unpainted metal plate or DIN rail which is connected to earth.

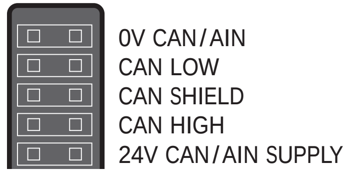
AMPLIFIER ENABLE (WATCHDOG) RELAY OUTPUTS

One internal relay contact is available to enable external amplifiers when the controller has powered up correctly and the system and application software is ready. The amplifier enable is a solid-state relay with an ON resistance of 25 ohms at 100mA. The enable relay will be open circuit if there is no power on the controller OR a motion error exists on a servo axis OR the user program sets it open with the **WDOG**=OFF command.

The amplifier 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.

All stepper and servo amplifiers must be inhibited when the amplifier enable output is open circuit

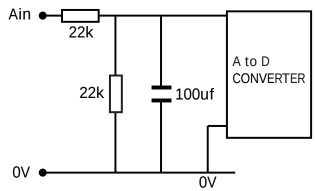
CANBUS



The MC664 features a built-in CAN channel. This is primarily intended for Input/Output expansion via Trio's range of CAN digital and analogue I/O modules. It may be used for other purposes when I/O expansion is not required.

The CANbus port is electrically equivalent to a DeviceNet node.

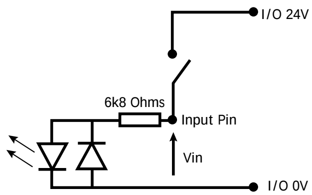
ANALOGUE INPUTS



Two built-in 12 bit analogue inputs are provided which are set up with a scale of 0 to 10V. External connection to these inputs is via the 2-part terminal strip on the lower front panel.

A 24V d.c. supply must be applied to the CANbus port to provide power for the analogue input circuit.

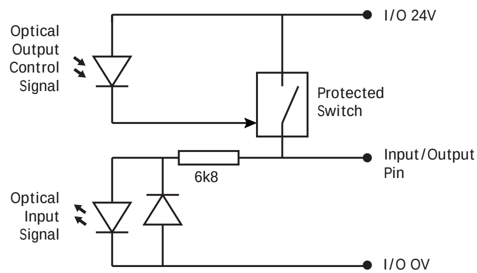
24V INPUT CHANNELS



The *Motion Coordinator* has 16 24V Input channels built into the master unit. These may be expanded to 1024 Inputs by the addition of CAN-16 I/O modules and EtherCAT I/O.

The first 8 channels (0 ... 7) are input only, using high speed opto-isolators suitable for position capture (**REGISTRATION**). Channels 8 to 15 are bi-directional and may be used for Input or Output to suit the application.

24V I/O CHANNELS

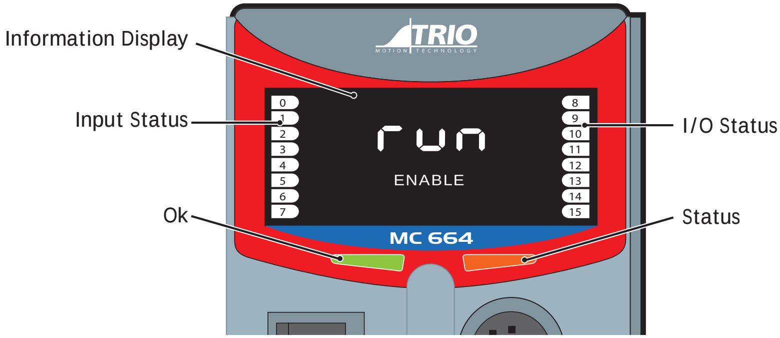


Input/output channels 8..15 are bi-directional and may be used for Input or Output to suit the application. The inputs have a protected 24V sourcing output connected to the same pin. If the channel is to be used as an Input then the Output should not be switched on in the program. The output circuit has electronic over-current protection and thermal protection which shuts the output down when the current exceeds 250mA.

Care should be taken to ensure that the 250mA limit for the output circuit is not exceeded, and that the total load for the group of 8 outputs does not exceed 1A

BACKLIT DISPLAY

The information display area shows the IP address and subnet mask during power-up and whenever an Ethernet cable is first connected to the MC664. During operation, this display shows run, Off or Err to indicate the MC664 status. Below the main status display are the ERROR and ENABLE indicators.



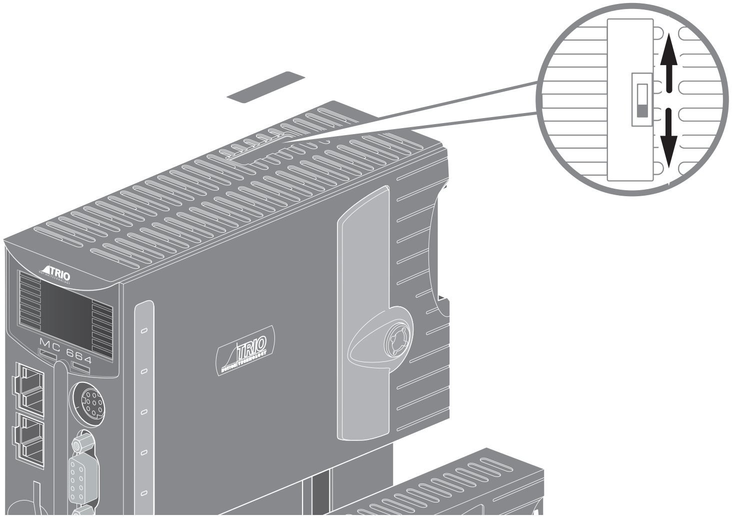
ERROR	An error has occurred (see Error Display Codes table below for details).
ENABLE	When illuminated, WDOG is ON.

A bank of 8 indicators at the left side shows the Digital Input States and a similar bank on the right shows the state of I/O8 to I/O15. The I/O displayed can be altered using the **`DISPLAY`** command.

Two LED's are provided to show the processor (OK) and system status.

Error Display Codes		
Unn	Unit error on slot nn	
Ann	Axis error on axis aa	
Caa	Configuration error on unit aan	ie: too many axes
Exx	System error	E00 - RAM error 8bit BB - RAM (VR) E01 - RAM error 16 bit BB - RAM (TABLE) E04 - VR/TABLE corrupt entry E05 - Invalid MC_CONFIG file E06 - Started in SAFE mode (system timeout) E07 - FPGA Error E08 - Flash memory error E09 - ProcessoR Exception

RECOVERY SWITCH



MC664 FEATURE SUMMARY

Size	201 mm x 56 mm x 155 mm (HxWxD).
Weight	750g
Operating Temp.	0 - 45 degrees C.
Control Inputs	Forward Limit, Reverse Limit, Datum Input, Feedhold Input.
Communication Ports	RS232 channel: up to 38400 baud. RS485 channel: up to 38400 baud. CANbus port (DeviceNet and CANopen compatible) Ethernet: 10/100 BaseT multiple port connection.
Position Resolution	64 bit position count.
Speed Resolution	32 bits. Speed may be changed at any time. Moves may be merged.
Servo Cycle	4ms max. 125μs minimum (50μs MC664-X)
Programming	Multi-tasking TrioBASIC system, maximum 22 user processess. IEC 61131-3 programming system.
Interpolation modes	Linear 1-64 axes, circular, helical, spherical, CAM Profiles, speed control, electronic gearboxes.
Memory	8 Mbyte user memory. 2 Mbyte TABLE memory. Automatic flash EPROM program storage.
Table	512,000 table positions. 196,608 positions in Flash memory. Option to store table.
VR	65,536 VR positions in Flash memory.
SD Card	Standard SD Card compatible to 16Gbytes. Used for storing programs and/or data.
Power Input	24V d.c., Class 2 transformer or power source. 18..29V d.c. at 625mA typical.
Amplifier Enable Output	Normally open solid-state relay rated 24V ac/dc nominal. Maximum load 100mA. Maximum voltage 29V.
Analogue Inputs	2 isolated x 12 bit 0 to 10V.
Serial / Encoder Power Output	5V at 150mA.
Digital Inputs	8 Opto-isolated high speed 24V inputs.
Digital I/O	8 Opto-isolated 24V outputs. Current sourcing (PNP) 250 mA. (max. 1A per bank of 8).

Motion Coordinator MC508

OVERVIEW

The *Motion Coordinator* MC508 is based on Trio's high-performance ARM Coretex-A9 ® double-precision technology and provides 8 axes of servo, or 8 - 16 axes of pulse+direction control for stepper drives or pulse-input servo drives. Trio uses advanced **FPGA** techniques to reduce the size and fit the pulse output and servo circuitry in a compact DIN-rail mounted package. The MC508 is housed in a rugged plastic case with integrated earth chassis and incorporates all the isolation circuitry necessary for direct connection to external equipment in an industrial environment. Filtered power supplies are included so that it can be powered from the 24V d.c. logic supply present in most industrial cabinets.

It is designed to be configured and programmed for the application using a PC running Trio's *Motion* Perfect application software, and then may be set to run "standalone" if an external computer is not required for the final system. Programs and data are stored directly to **FLASH** memory, thus eliminating the need for battery backed storage.

The Multi-tasking version of TrioBASIC for the MC508 allows up to 22 TrioBASIC programs to be run simultaneously on the controller using pre-emptive multi-tasking. In addition, the operating system software includes a the IEC 61131-3 standard run-time environment (licence key required).

PROGRAMMING

The Multi-tasking ability of the MC508 allows parts of a complex application to be developed, tested and run independently, although the tasks can share data and motion control hardware. The 22 available tasks can be used for TrioBASIC or IEC 61131-3 programs, or a combination of both can be run at the same time, thus allowing the programmer to select the best features of each.

I/OCAPABILITY

The MC508 has 16 built in 24V inputs, selectable in banks of 8 between NPN and PNP operation and 16 output channels. These may be used for system interaction or the inputs may be defined to be used by the controller for end-of-travel limits, registration, homing and feedhold functions if required. 16 programmable status indicators are provided for I/O monitoring. The MC508 can have up 512 additional external Input and Output channels connected using DIN rail mounted CAN I/O modules. These units connect to the built-in CANbus port.

COMMUNICATIONS

A 10/100 Base-T Ethernet port is fitted as standard and this is the primary communications connection to the MC508. Many protocols are supported including Telnet, Modbus TCP, UDP, Ethernet IP and TrioPCMotion. Check the Trio website (www.triomotion.com) for a complete list.

The MC508 has one built in RS232 port and one built in duplex RS485 channel for simple factory communication systems. Either the RS232 port or the RS485 port may be configured to run the Modbus or Hostlink protocol for PLC or HMI interfacing.

If the built-in CAN channel is not used for connecting I/O modules, it may optionally be used for CAN communications. E.g. DeviceNet, CanOpen etc.

REMOVABLE STORAGE

The MC508 has a micro-SD Card slot which allows a simple means of transferring programs, firmware and data without a PC connection. Offering the OEM easy machine replication and servicing.

The memory slot is compatible with a wide range of micro-SD cards up to 16 GB using the FAT32 compatible file system.

AXIS POSITIONING FUNCTIONS

The motion control generation software receives instructions to move an axis or axes from the TrioBASIC or IEC 61131-3 language which is running concurrently on the same processor. The motion generation software provides control during operation to ensure smooth, coordinated movements with the velocity profiled as specified by the controlling program. Linear interpolation may be performed on groups of axes, and circular, helical or spherical interpolation in any two/three orthogonal axes. Each axis may run independently or they may be linked in any combination using interpolation, CAM profile or the electronic gearbox facilities.

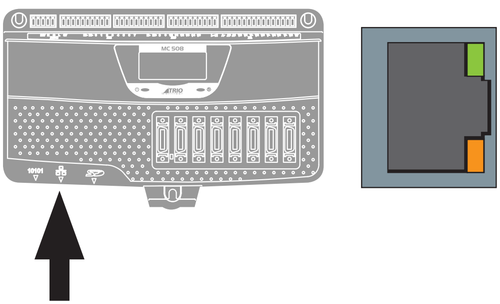
Consecutive movements may be merged to produce continuous path motion and the user may program the motion using programmable units of measurement (e.g. mm, inches, revs etc.). The module may also be programmed to control only the axis speed. The positioner checks the status of end of travel limit switches which can be used to cancel moves in progress and alter program execution.

CONNECTIONS TO THE MC508

ETHERNET PORT CONNECTION

Physical layer: 10/100 Base-T

CONNECTOR: RJ45



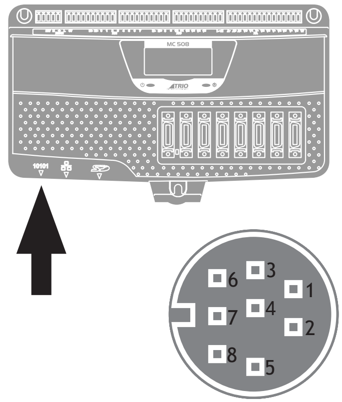
The Ethernet port is the default connection between the *Motion Coordinator* and the host PC running the *Motion* Perfect development application.

SERIAL CONNECTIONS

The MC508 features two serial ports. Both ports are accessed through a single 8 pin connector.

SERIAL CONNECTOR

Pin	Function	Note
1	RS485 Data In A Rx+	Serial Port #2
2	RS485 Data In B Rx-	
3	RS232 Transmit	Serial Port #1
4	0V Serial	
5	RS232 Receive	Serial Port #1
6	Internal 5V	5V supply is limited to 150mA, shared with encoder ports
7	RS485 Data Out Z Tx-	Serial Port #2
8	RS485 Data Out Y Tx+	Serial Port #2



PULSE+DIRECTION OUTPUTS / ENCODER INPUTS

The MC508 is designed to support any combination of servo and pulse driven motor drives on the standard controller hardware. There are 2 versions of the MC508; the servo version and the pulse output only version. In the P848 pulse output version, only axes 0 to 7 can be configured. The P849 servo version makes axes 8 to 15 available as pulse and direction output.

Each of the first eight axes (0-7) can be enabled as servo (P849 version), pulse output or encoder according to the user's requirements by setting the axis **ATYPE** parameter. Axes 8 to 15 can be set as either pulse output or encoder on the P849 version.

The function of the 20-pin MDR connectors will be dependent on the specific axis configuration which has been defined. If the axis is setup as a servo, (P849 only) the connector will provide the analogue speed signal and encoder input. If the axis is configured as a pulse output, the connector provides differential outputs for step/direction or simulated encoder, and enable signals.

The flexible axis connector also provides 2 digital inputs (24V) and a current-limited 5V output capable of powering most encoders. This simplifies wiring and eliminates external power supplies.

Pin	Incremental Encoder Function	Pulse & Direction Function	Pulse & Direction Function (P849 ONLY)	Absolute Encoder Function
1	Enc A(n)	Pulse(n)	Pulse(n)	Clock(n)
2	Enc /A(n)	/Pulse(n)	/Pulse(n)	/Clock(n)
3	Enc B(n)	Dir(n)	Dir(n)	NC
4	Enco /B(n)	/Dir(n)	/Dir(n)	NC
5	+5V Enc (100mA max.)			
6	Do not connect			
7	WDOG(n)+			

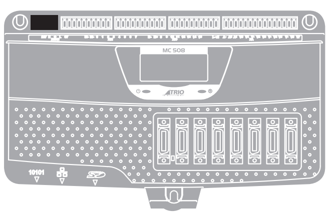
Pin	Incremental Encoder Function	Pulse & Direction Function	Pulse & Direction Function (P849 ONLY)	Absolute Encoder Function
8	WDOG(n)-			
9	Input A+ (16 + n*2)			
10	Input A/B Common			
11	Enc Z(n)	Enable(n)	Pulse(n+8)	Data(n)
12	Enc /Z(n)	/Enable(n)	/Pulse(n+8)	/Data(n)
13	NC	NC	Dir(n+8)	NC
14	NC	NC	/Dir(n+8)	NC
15	0V Enc			
16	Do not connect			
17	VOUT + (n)			
18	VOUT - (n)			
19	Do not connect			
20	Input B + (17 + n*2)			
Shell	Screen			

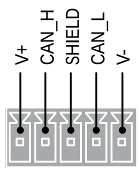
1. n=axis number
2. WDOG(n)+/- = normally open solid state relay, rated 24V@100mA (one per axis)
3. Input A/B Common, 0V_Enc & VOUT- are all isolated so must be connected with the correct signals.
4. +5V Output 400mA maximum current output is shared between all 8 axis connectors and the serial connector. 100mA maximum per axis connector.

REGISTRATION

Axes 0 to 7 each have 2 available registration events. These are assigned in a flexible way to any of the first 8 digital inputs or can be used with the Z mark input on the encoder port.

5-WAY CONNECTOR



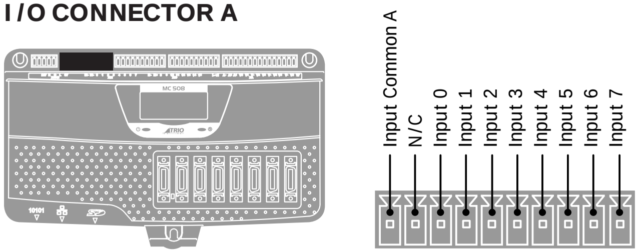


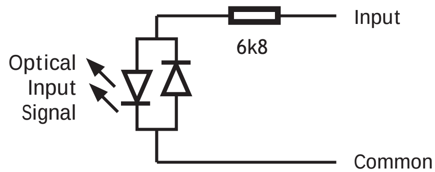
This is a 5 way 3.5 mm pitch connector. The connector is used both to provide the 24 Volt power to the MC508 and provide connections for I/O expansion via Trio's digital and analogue CAN I/O expanders. 24 Volts must be provided as this powers the unit.

This 24 Volt input is internally isolated from the I/O 24 Volts and the +/-10V voltage outputs.

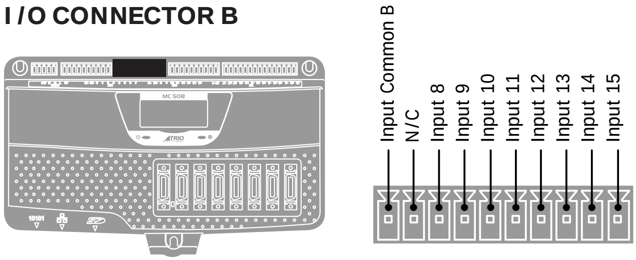
24V d.c., Class 2 transformer or power source required for UL compliance. The MC508 is grounded via the metal chassis. It MUST be installed on an unpainted metal plate or DIN rail which is connected to earth. An earth screw is also provided on the rear of the chassis for bonding the MC508 to ground.

I/O CONNECTOR A





I/O CONNECTOR B

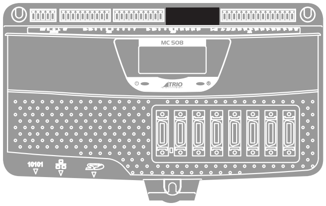


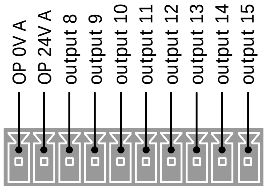
24V INPUT CHANNELS

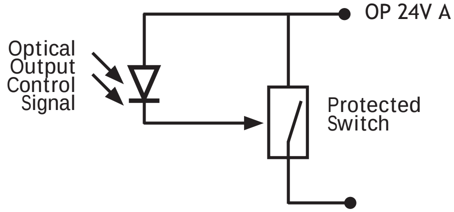
The MC508 has 32 dedicated 24V Input channels built into the master unit. A further 512 inputs can be provided by the addition of CAN I/O modules. The dedicated input channels are labelled channels 0..7, 8..15 and 2 per flexible axis connector (16..31). Two terminals marked XAC and XBC are provided for the input common connections. Connect XAC/XBC to 0V for PNP (source) input operation or connect to +24V for NPN (sink) operation. Input connectors A and B are independent so one can be PNP while the other is NPN. Flexible axis connector inputs are fixed function PNP inputs.

Inputs 0 to 7 can be used as registration inputs for axes 0 to 7, using the **REGIST** command.

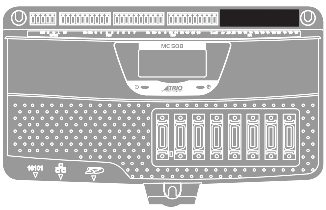
I/O CONNECTOR C

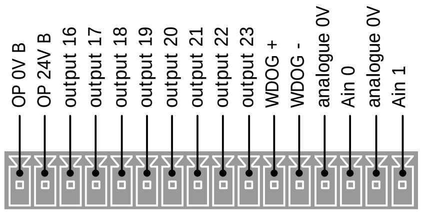






I/OO CONNECTOR D





24V OUTPUT POWER

The XC-/XD- 0 Volts and XC+/XD+ 24 Volts are used to power the 24 Volt digital outputs. XD-/XD+ also powers the analogue I/O, including the servo DAC outputs.

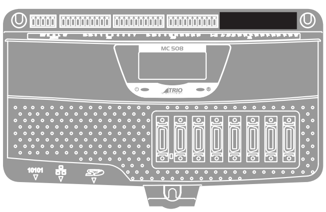
Each digital I/O connector is isolated from the module power inputs and from the other I/O connectors.

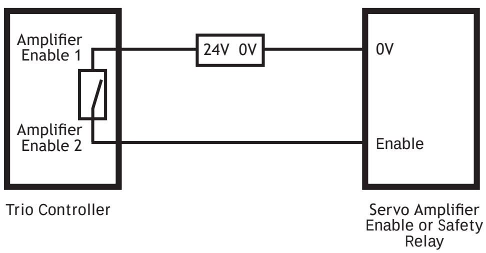
24V OUTPUT CHANNELS

Output channels 8..23 are output only of PNP type 24V source. The output circuit has electronic over-current protection and thermal protection which shuts the output down when the current exceeds 500mA.

Care should be taken to ensure that the 500mA limit for each output circuit is not exceeded, and that the total load for the group of 8 outputs does not exceed 4 Amps.

AMPLIFIER ENABLE (WATCHDOG) RELAY OUTPUTS





An internal relay contact is available to enable external amplifiers when the controller has powered up correctly and the system and application software is ready. The amplifier enable is a solid-state relay with an ON resistance of 25Ω at 100mA. The enable relay will be open circuit if there is no power on the controller OR a motion error exists on a servo axis OR the user program sets it open with the **WDOG**=OFF command.

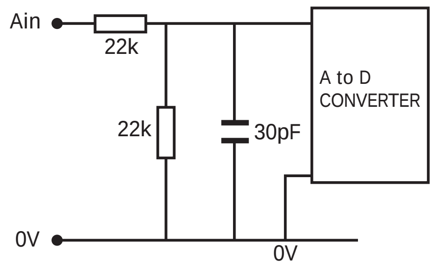
The amplifier enable relay may, for example, be incorporated within a hold-up circuit or chain that must be intact before a 3-phase power input is made live.

All stepper and servo amplifiers must be inhibited when the amplifier enable output is open circuit

ANALOGUE INPUTS

Two built-in 12 bit analogue inputs are provided which are set up with a scale of 0 to 10V. External connection to these inputs is via the 2-part terminal strip I/O connector D.

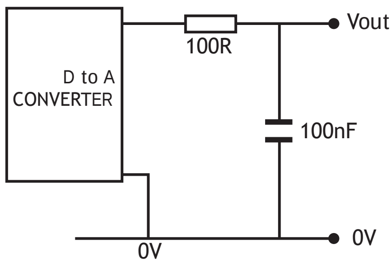
A 24V d.c. supply must be applied to I/O connector D (XD+/XD-) to provide power for the analogue input circuit.



ANALOGUE OUTPUTS

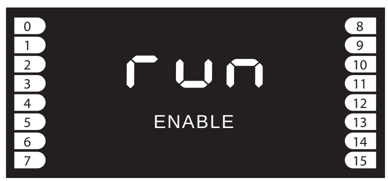
The MC508 has 8 12-bit analogue outputs, one per flexible axis connector, scaled at +/-10V. Each output is assigned to one servo axis, or in the case where the axis is not used, or is set as a pulse+direction/ simulated encoder output, the analogue output may be set to a voltage directly in software.

A 24V d.c. supply must be applied to I/O connector D to provide power for the analogue output circuit.



BACKLIT DISPLAY

On power-up, the information display area shows bt during the boot process, then the MC508 version is displayed, showing P848 for the 8 axis pulse output version and P849 for the 8 axis servo + 8 axis pulse output version. The IP address and subnet mask is shown on power-up and whenever an Ethernet cable is first connected to the MC508.



During operation, this display shows run, OFF or Err to indicate the MC508 status. Below the main status display are the **ERROR** and **ENABLE** indicators.

ERROR:	An error has occurred (see Error Display Codes table below for details).
ENABLE:	When illuminated, WDOG is ON.

A bank of 8 indicators at the left side shows the State of digital Inputs 0..7 and a similar bank on the right shows the state of inputs 8..15. The I/O displayed can be altered using the **DISPLAY** command.

Two LED's are provided to show the processor (OK) and system status.

ERROR DISPLAY CODES

Ann	Axis error on axis nn	
Caa	Configuration error on unit aa	Ie: too many axes
Exx	System error	E00 - RAM error 8bit BB - RAM (VR)
		E01 - RAM error 16 bit BB - RAM (TABLE)
		E03 – N/A
		E04 - VR/TABLE corrupt entry
		E05 – Invalid MC_CONFIG file
		E06 – Started in SAFE mode
		E07 – FPGA error
		E08 – Flash memory error
		E09 – Processor exception

MC508 FEATURE SUMMARY

Size	132 mm x 226 mm x 35 mm (HxWxD).
Weight	640g
Operating Temp.	0 - 45 degrees C.
Control Inputs	Forward Limit, Reverse Limit, Datum Input, Feedhold Input.
Communication Ports	RS232 channel: up to 128k baud.
	CANbus port (DeviceNet and CANopen compatible)
	Ethernet: 10/100 BaseT multiple port connection.
Position Resolution	64 bit position count.
Speed Resolution	32 bits. Speed may be changed at any time. Moves may be merged.
Servo Cycle	125μs minimum, 1ms default, 2ms max.
Programming	Multi-tasking TrioBASIC system and IEC 61131-3 programming system. Maximum 22 user processes.
Interpolation modes	Linear 1-8 axes, circular, helical, spherical, CAM Profiles, speed control, electronic gearboxes.
Memory	8 Mbyte user memory. Automatic flash EPROM program and data storage.

VR	16384 global VR data in FLASH memory. (automatic-store)
TABLE	512,000 x 64 bit TABLE memory. Option to auto-save 64,000 TABLE points
SD Card	Standard micro-SD Card compatible to 16 GB. Used for storing programs and/or data.
Real Time Clock	Capacitor backed for 10 days of power off.
Power Input	24V d.c., Class 2 transformer or power source.
	Processor/CANbus 18..29V d.c. at 225mA.
	Analogue I/O 18..29V d.c. at 50 mA.
	Digital Outputs, 18..29V d.c at up to 4 Amps per bank of 8.
Amplifier Enable Output	Normally open solid-state relay rated 24V ac/dc nominal. Maximum load 100mA. Maximum Voltage 29V.
Analogue Inputs	2 isolated, 12 bit, 0 to 10V.
Serial / Encoder Power Output	5V at 150mA.
Digital Inputs	32 Opto-isolated 24V inputs. 16 are selectable PNP/NPN.
Digital Outputs	16 Opto-isolated 24V outputs. Current sourcing (PNP) 500 mA. (max. 4A per bank of 8).
Product Code	P848 : MC508, 8 axis stepper
	P849 : MC508, 8 axis servo or stepper + 8 axis stepper or encoder

Motion Coordinator MC464

OVERVIEW

The *Motion Coordinator* MC464 is Trio's new generation modular servo control positioner with the ability to control servo or stepper motors by means of Digital Drive links (e.g. EtherCAT, Sercos, etc) or via traditional analogue and encoder or pulse and direction. A maximum of 7 expansion modules can be fitted to control up to 64 axes which gives the flexibility required in modern system design. The MC464 is housed in a rugged plastic case with integrated earth chassis and incorporates all the isolation circuitry necessary for direct connection to external equipment in an industrial environment. Filtered power supplies are included so that it can be powered from the 24V d.c. logic supply present in most industrial cabinets.

It is designed to be configured and programmed for the application using a PC running the *Motion* Perfect application software, and then may be set to run "standalone" if an external computer is not required for the final system.

The Multi-tasking version of TrioBASIC for the MC464 allows up to 22 TrioBASIC programs to be run simultaneously on the controller using pre-emptive multi-tasking. In addition, the operating system software includes the IEC 61131-3 standard run-time environment (licence key required).

PROGRAMMING

The Multi-tasking ability of the MC464 allows parts of a complex application to be developed, tested and run independently, although the tasks can share data and motion control hardware. IEC 61131-3 programs can be run at the same time as TrioBASIC allowing the programmer to select the best features of each.

I/O CAPABILITY

The MC464 has 8 built in 24V inputs and 8 bi-directional I/O channels. These may be used for system interaction or may be defined to be used by the controller for end of travel limits, registration, datuming and feedhold functions if required. Each of the Input/Output channels has a status indicator to make it easy to check them at a glance. The MC464 can have up 512 external Input/Output channels connected using DIN rail mounted CAN I/O modules. These units connect to the built-in CAN channel.

COMMUNICATIONS

A 10/100 base-T Ethernet port is fitted as standard and this is the primary communications connection to the MC464. Many protocols are supported including Telnet, Modbus TCP, Ethernet IP and TrioPCMotion. Check the Trio website (www.triomotion.com) for a complete list.

The MC464 has one built in RS232 port and one built in duplex RS485 channel for simple factory

communication systems. Either the RS232 port or the RS485 port may be configured to run the Modbus or Hostlink protocol for PLC or HMI interfacing.

If the built-in CAN channel is not used for connecting I/O modules, it may optionally be used for CAN communications. E.g. DeviceNet slave or CANopen master.

The Anybus CompactCom Carrier Module (P875) can be used to add other fieldbus communications options

REMOVABLE STORAGE

The MC464 has a SD Card slot which allows a simple means of transferring programs, firmware and data without a PC connection. Offering the OEM easy machine replication and servicing.

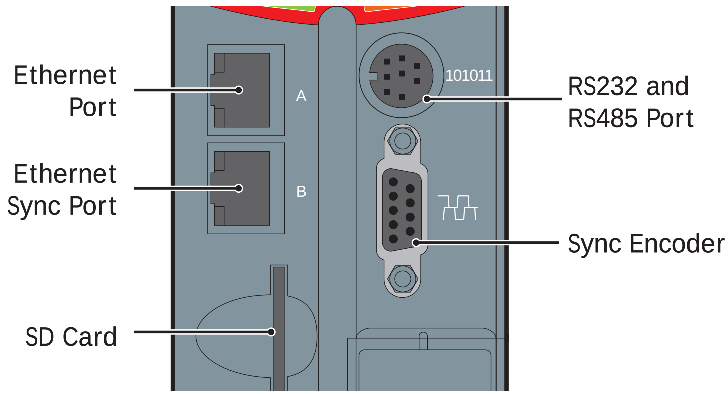
The memory slot is compatable with a wide range of SD cards up to 2Gbytes using the FAT32 compatible file system.

AXIS POSITIONING FUNCTIONS

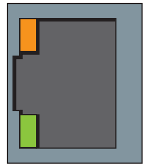
The motion control generation software receives instructions to move an axis or axes from the TrioBASIC or IEC 61131-3 language which is running concurrently on the same processor. The motion generation software provides control during operation to ensure smooth, coordinated movements with the velocity profiled as specified by the controlling program. Linear interpolation may be performed on groups of axes, and circular, helical or spherical interpolation in any two/three orthogonal axes. Each axis may run independently or they may be linked in any combination using interpolation, CAM profile or the electronic gearbox facilities.

Consecutive movements may be merged to produce continuous path motion and the user may program the motion using programmable units of measurement (e.g. mm, inches, revs etc.). The module may also be programmed to control only the axis speed. The positioner checks the status of end of travel limit switches which can be used to cancel moves in progress and alter program execution.

CONNECTIONS TO THE MC464



ETHERNET PORT CONNECTION



Physical layer: 10/100 base_T

Connector: RJ45

The Ethernet port is the default connection between the *Motion Coordinator* and the host PC running *Motion* Perfect programming.

ETHERNET SYNC PORT

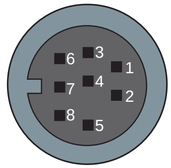
Not used.

MC464 SERIAL CONNECTIONS

The MC464 features two serial ports. Both ports are accessed through a single 8 pin connector.

SERIAL CONNECTOR

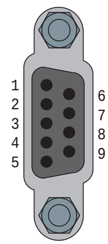
Pin	Function	Note
1	RS485 Data In A Rx+	Serial Port #2
2	RS485 Data In B Rx-	
3	RS232 Transmit	Serial Port #1
4	0V Serial	
5	RS232 Receive	Serial Port #1
6	Internal 5V	5V supply is limited to 150mA, shared with sync port
7	RS485 Data Out Z Tx-	Serial Port #2
8	RS485 Data Out Y Tx+	Serial Port #2



SYNC ENCODER

The sync encoder port is bidirectional. It can be used as a reference encoder input or as an encoder simulation output to act as a master reference for other parts of the system.

Pin	Function	Pulse & Direction
1	Enc. A	Step+
2	Enc. /A	Step-
3	Enc. B	Direction+
4	Enc. /B	Direction-
5	0V Encoder	0V Stepper
6	Enc. Z	Enable+
7	Enc. /Z	Enable-
8	5V *	5V*
9	5V Registration input	5V Registration input

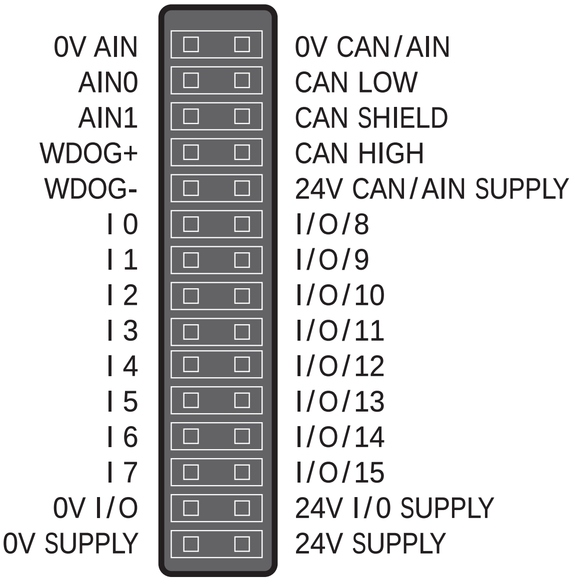


Pin	Function	Pulse & Direction
*5V supply is limited to 150mA (shared with serial port)		

REGISTRATION

The MC464 built in port has 2 available registration events. These can be used with the Z mark, the registration input on the sync port, input 0 or input 1.

24V POWER SUPPLY INPUT



The MC464 is powered entirely via the 24V d.c.supply connections. The unit uses internal DC-DC converters to generate independent 5V logic supply, the encoder/serial 5V supply and other internal power supplies. I/O, analogue and CANbus circuits are isolated from the main 24V power input and must be powered separately. For example; it is often necessary to power the CANbus network remotely via the CANbus cable.

24V d.c., Class 2 transformer or power source required for **UL** compliance. The MC464 is grounded via the metal chassis. It MUST be installed on an unpainted metal plate or DIN rail which is connected to earth.

AMPLIFIER ENABLE (WATCHDOG) RELAY OUTPUTS

One internal relay contact is available to enable external amplifiers when the controller has powered up correctly and the system and application software is ready. The amplifier enable is a solid-state relay with an ON resistance of 25 ohms at 100mA. The enable relay will be open circuit if there is no power on the controller OR a motion error exists on a servo axis OR the user program sets it open with the **WDOG**=OFF command.

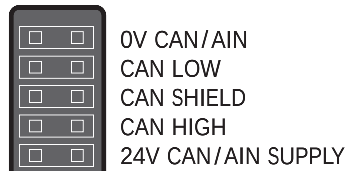
The amplifier enable relay may, for example, be incorporated within a hold-up circuit or chain that must be intact before a 3-phase power input is made live.

All stepper and servo amplifiers must be inhibited when the amplifier enable output is open circuit

CANBUS

The MC464 features a built-in CAN channel. This is primarily intended for Input/Output expansion via Trio's range of CAN digital and analogue I/O modules. It may be used for other purposes when I/O expansion is not required.

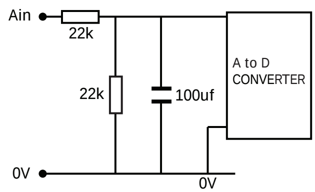
The CANbus port is electrically equivalent to a DeviceNet node.



ANALOGUE INPUTS

Two built-in 12 bit analogue inputs are provided which are set up with a scale of 0 to 10V. External connection to these inputs is via the 2-part terminal strip on the lower front panel.

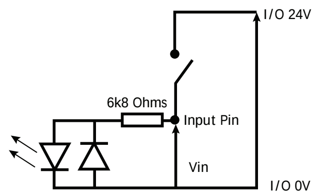
A 24V d.c. supply must be applied to the CANbus port to provide power for the analogue input circuit.



24V INPUT CHANNELS

The *Motion Coordinator* has 16 24V Input channels built into the master unit. These may be expanded to 256 Inputs by the addition of CAN-16 I/O modules.

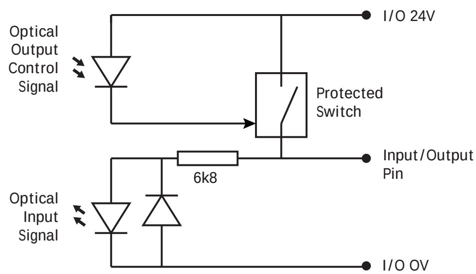
All of the 24V Input channels have the same circuit although 8 on the master unit have 24V Output channels connected to the same pin. These bi-directional channels may be used for Input or Output to suit the application. If the channel is to be used as an Input then the Output should not be switched on in the program.



24V I/O CHANNELS

Input/output channels 8..15 are bi-directional and may be used for Input or Output to suit the application. The inputs have a protected 24V sourcing output connected to the same pin. If the channel is to be used as an Input then the Output should not be switched on in the program. The input circuitry is the same as on the dedicated inputs. The output circuit has electronic over-current protection and thermal protection which shuts the output down when the current exceeds 250mA.

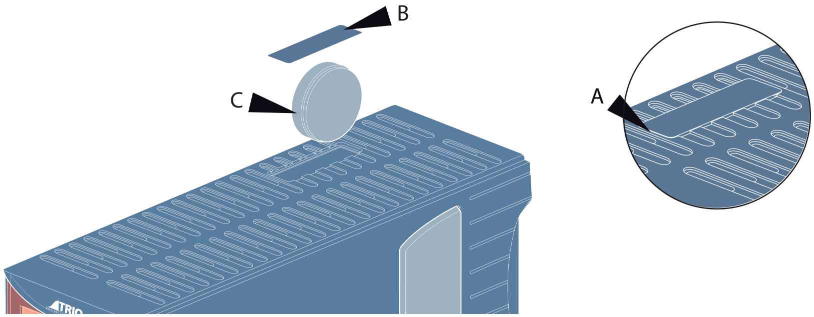
Care should be taken to ensure that the 250mA limit for the output circuit is not exceeded, and that the total load for the group of 8 outputs does not exceed 1A



BATTERY

The MC464 incorporates a user replaceable battery for the battery back-up RAM. For replacement, use battery model CR2450 or equivalent.

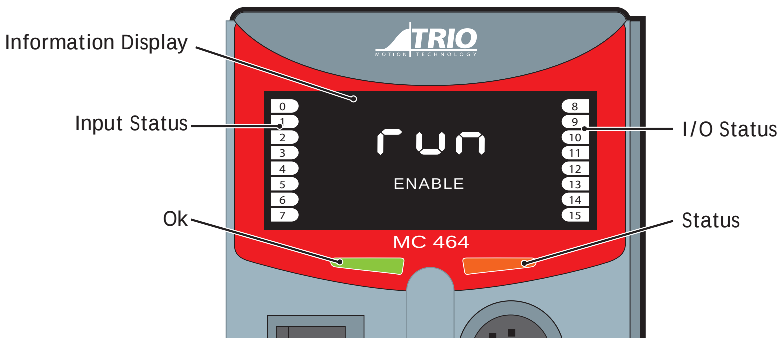
To replace the battery, insert screwdriver under the frontmost ventilation slot (A) and prize off the battery cover (B) and pull the battery ribbon to lift the battery (C) from the MC464. Replacing is the reverse of the procedure.



To Avoid losing the memory contents, the new battery should be inserted within 30 seconds of the old one being removed.

BACKLIT DISPLAY

The information display area shows the IP address and subnet mask during power-up and whenever an Ethernet cable is first connected to the MC464. During operation, this display shows run, Off or Err to indicate the MC464 status. Below the main status display are the ERROR, ENABLE and BATTERY LOW indicators.



ERROR An error has occurred (see Error Display Codes table below for details).
ENABLE When illuminated, WDOG is ON.
BATTERY LOW When illuminated the battery needs replacing.

A bank of 8 indicators at the left side shows the Digital Input States and a similar bank on the right shows the state of I/O8 to I/O15. The I/O displayed can be altered using the **DISPLAY** command.

Two LED's are provided to show the processor (OK) and system status.

Error Display Codes		
Unn	Unit error on slot nn	
Ann	Axis error on axis aa	
Caa	Configuration error on unit aan	ie: too many axes
Exx	System error	E00 - RAM error 8bit BB - RAM (VR) E01 - RAM error 16 bit BB - RAM (TABLE) E03 - Battery Error E04 - VR/TABLE corrupt entry E05 – Invalid MC_CONFIG file E06 – Started in SAFE mode

MC464 FEATURE SUMMARY

Size	201 mm x 56 mm x 155 mm (HxWxD).
Weight	750g
Operating Temp.	0 - 45 degrees C.
Control Inputs	Forward Limit, Reverse Limit, Datum Input, Feedhold Input.
Communication Ports	RS232 channel: up to 38400 baud. RS485 channel: up to 38400 baud. CANbus port (DeviceNet and CANopen compatible) Ethernet: 10/100 BaseT multiple port connection.
Position Resolution	64 bit position count.
Speed Resolution	32 bits. Speed may be changed at any time. Moves may be merged.
Servo Cycle	125μs minimum, 1ms default, 2ms max.
Programming	Multi-tasking TrioBASIC system, maximum 20 user processess. IEC 61131-3 programming system.
Interpolation modes	Linear 1-64 axes, circular, helical, spherical, CAM Profiles, speed control, electronic gearboxes.
Memory	8 Mbyte user memory. 2 Mbyte TABLE battery-backed memory. Automatic flash EPROM program storage.
Table	512,000 table positions. 196,608 positions in battery backed memory.
VR	65,536 VR positions in battery backed memory.
SD Card	Standard SD Card compatible to 2Gbytes. Used for storing programs and/or data.
Power Input	24V d.c., Class 2 transformer or power source. 18..29V d.c. at 625mA typical.
Amplifier Enable Output	Normally open solid-state relay rated 24V ac/dc nominal. Maximum load 100mA. Maximum voltage 29V.
Analogue Inputs	2 isolated x 12 bit 0 to 10V.
Serial / Encoder Power Output	5V at 150mA.
Digital Inputs	8 Opto-isolated 24V inputs.
Digital I/O	8 Opto-isolated 24V outputs. Current sourcing (PNP) 250 mA. (max. 1A per bank of 8).

Motion Coordinator MC4N-Mini EtherCAT Master

OVERVIEW

The MC4N-ECAT is a new concept in high performance *Motion Coordinator*s which is dedicated to running remote servo and stepper drives via the EtherCAT real time automation bus. It is based on an up-rated version of the 532MHz ARM 11 processor which makes it ideal for high axis count machines or robotic applications.

It is designed to be configured and programmed for the application using a PC running the *Motion* Perfect application software, and then may be set to run “standalone” if an external computer is not required for the final system.

The Multi-tasking version of TrioBASIC for the MC4N-ECAT allows up to 22 TrioBASIC programs to be run simultaneously on the controller using pre-emptive multi-tasking. In addition, the operating system software includes the IEC 61131-3 standard run-time environment (licence key required).

Versions of the MC4N-ECAT are available for 2, 4, 8, 16 and 32 motor axes. All versions feature 32 software axes any of which may be used as virtual axes if not assigned to EtherCAT hardware.

PROGRAMMING

The Multi-tasking ability of the MC4N-ECAT allows parts of a complex application to be developed, tested and run independently, although the tasks can share data and motion control hardware. IEC 61131-3 programs can be run at the same time as TrioBASIC allowing the programmer to select the best features of each.

I/O CAPABILITY

The MC4N has 8 built in 24V inputs and 8 bi-directional I/O channels. These may be used for system interaction or may be defined to be used by the controller for end of travel limits, registration, datuming and feedhold functions if required. Each of the Input/Output channels has a status indicator to make it easy to check them at a glance. The MC4N-ECAT can have up 512 external Input/Output channels connected using DIN rail mounted CAN I/O modules. These units connect to the built-in CAN channel.

COMMUNICATIONS

A 10/100 base-T Ethernet port is fitted as standard and this is the primary communications connection to the MC4N-ECAT. Many protocols are supported including Telnet, Modbus TCP, Ethernet IP and TrioPCMotion. Check the Trio website (www.triomotion.com) for a complete list.

The MC4N-ECAT has one built in RS232 port and one built in duplex RS485 channel for simple factory communication systems. Either the RS232 port or the RS485 port may be configured to run the Modbus or Hostlink protocol for PLC or HMI interfacing.

If the built-in CAN channel is not used for connecting I/O modules, it may optionally be used for CAN communications. E.g. DeviceNet slave or CanOpen master.

REMOVABLE STORAGE

The SD Card maybe used for storing or transfering programs, reciepes and data to and from the MC4N-ECAT. The card must be FAT32 format and a maximum 16Gb size.

SD Cards may be FAT16 formatted when purchased. Re-format in a PC to FAT32 prior to use.

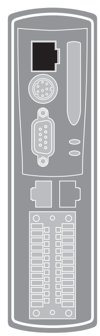
AXIS POSITIONING FUNCTIONS

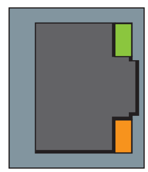
The motion control generation software receives instructions to move an axis or axes from the TrioBASIC or IEC 61131-3 language which is running concurrently on the same processor. The motion generation software provides control during operation to ensure smooth, coordinated movements with the velocity profiled as specified by the controlling program. Linear interpolation may be performed on groups of axes, and circular, helical or spherical interpolation in any two/three orthogonal axes. Each axis may run independently or they may be linked in any combination using interpolation, CAM profiles or the electronic gearbox facilities.

Consecutive movements may be merged to produce continuous path motion and the user may program the motion using programmable units of measurement (e.g. mm, inches, revs etc.). The module may also be programmed to control only the axis speed. The positioner checks the status of end of travel limit switches which can be used to cancel moves in progress and alter program execution.

CONNECTIONS TO THE MC4N

ETHERNET PORT CONNECTION





Physical layer: 10/100 base_T

Connector: RJ45

A standard Ethernet connector is provided for use as the primary programming interface.

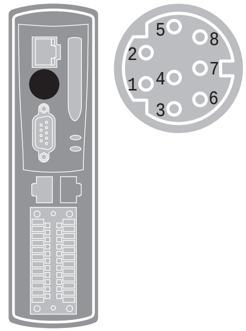
The Trio programming software, *Motion* Perfect, must be installed on a Windows based PC that is fitted with an Ethernet connection. The IP address is displayed on the MC4N display for a few seconds after power-up or when an Ethernet cable is plugged in.

Ethernet cable must be CAT 5 or better.

The Standard Ethernet connection may also be used for Ethernet-IP, Modbus and other factory communications.

SERIAL CONNECTIONS

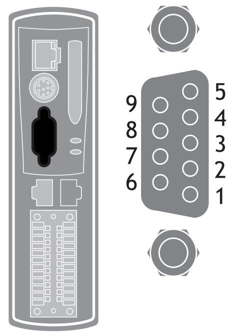
Pin	Function	Note
1	RS485 Data In A Rx+	Serial Port #2
2	RS485 Data In B Rx-	
3	RS232 Transmit	Serial Port #1
4	0V Serial/Encoder	
5	RS232 Receive	Serial Port #1
6	5V Output	150mA max (Current shared with encoder port)
7	RS485 Data Out Z Tx-	Serial Port #2
8	RS485 Data Out Y Tx+	



FLEXIBLE AXIS PORT

Pin	Encoder	Stepper Axis	Absolute Encoder
1	Enc. A	Step +	Clock
2	Enc. /A	Step -	/Clock
3	Enc. B	Direction +	--------
4	Enc. /B	Direction -	--------
5	0V Serial/Encoder	0V Serial/Encoder	0V 0V Serial/Encoder
6	Enc. Z	Enable +	Data
7	Enc. /Z	Enable -	/Data
8	5V*	5V*	5V*
9	Not Connected	Not Connected	Not Connected

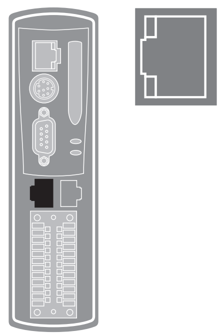
*Current limit is 150mA max. Shared with serial port.



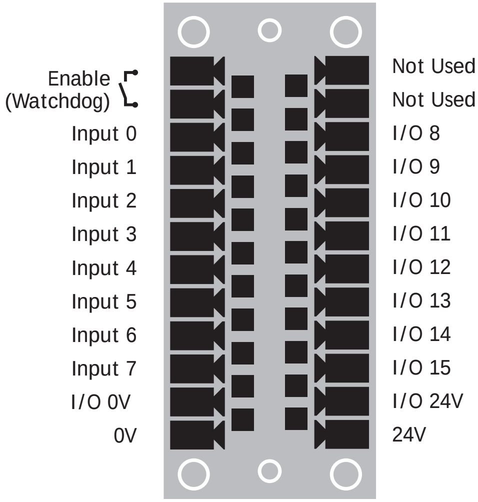
ETHERCAT PORT

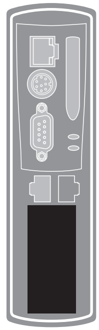
The MC4N-ECAT acts as an EtherCAT master. EtherCAT drives and I/O devices are normally connected in a chain. Other topologies are possible when specialised EtherCAT routers are used in the network.

Up to 32 EtherCAT axes and 1024 digital I/O points may be connected via the EtherCAT bus.

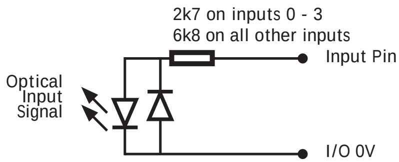


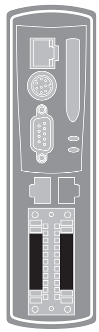
I/O CONNECTOR



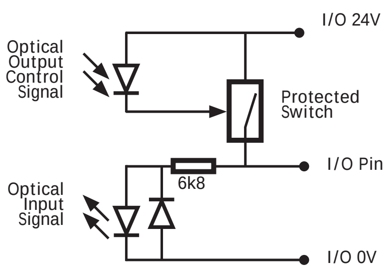


Inputs 0 - 3 have fast opto-couplers for use as axis registration inputs. Inputs 4-7 may also be used as registration inputs.





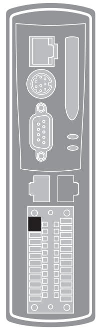
Inputs / Outputs 8 - 15

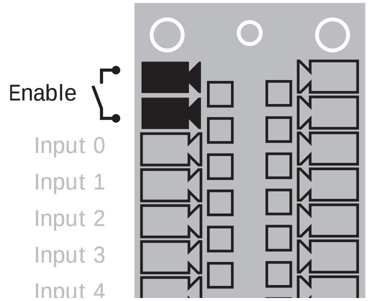


The MC4N is grounded via the metal chassis. Fit a short shield connection between the chassis earth screw and the earthed metal mounting panel / plate.

AMPLIFIER ENABLE (WATCHDOG) RELAY OUTPUT

An internal relay may be used to enable external amplifiers when the controller has powered up correctly and the system and application software are ready. The amplifier enable is a single pole solid state relay with a normally open "contact". The enable relay contact will be open circuit if there is no power on the controller OR an axis error exists OR the user program sets it open with the **WDOG=OFF** command.





EtherCAT drives will be enabled via the EtherCAT network so the "Amplifier Enable" connection is not normally required.

All non EtherCAT stepper and servo amplifiers MUST **be inhibited when the amplifier enable output is open circuit**

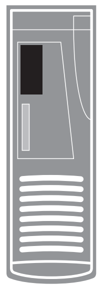
An additional safety relay may be required so as to meet machine safety approvals.

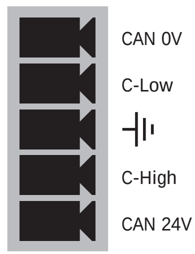
5 WAY CAN CONNECTOR

This is a 5 way 3.5mm pitch connector. The connector is used both to provide the 24 Volt power to the MC4N CAN circuit and provide connections for I/O expansion via Trio's CAN I/O expanders. A 24V dc, Class 2 transformer or power source should be used.

This 24 Volt input is internally isolated from the I/O 24V and main 24V power.

The **CAN** connector may be left unused.

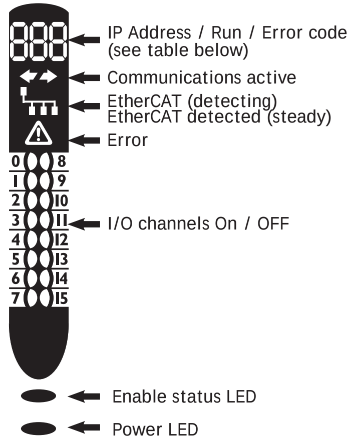




DISPLAY

The IP address and subnet mask of the MC4N-ECAT is shown on the LCD display for a few seconds after power-up. The factory default IP address is 192.168.0.250. This can be changed using the **IP _ ADDRESS** commands via the *Motion* Perfect software tool.

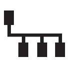
Display Example	Description	Details
SYS	Displayed on controller start	
901	Model code : Displayed on power up	P900 : 2 axes P901 : 4 axes P902 : 8 axes P903 : 16 axes P904 : 32 axes
192.168.0.250	IP Address :	Displayed on power up OR after ethernet connection for 15 seconds
Unn	Unit error on slot nn	
Ann	Axis error on axis nn	
Caa	Configuration error on unit aa	ie: too many axes
Run / Off	Enable status	
Err xx	Error codes	Ann : Error on Axis nn Unn : Unit error on slot nn Caa : Configuration error on unit nn, ie: too many axes E04 : VR/TABLE corrupt entry



COMMUNICATIONS ACTIVE


This symbol appears when the firmware has detected one or more valid EtherCAT nodes on the network.

ETHERCAT DETECTION


This symbol shows the EtherCAT connection status.

Indicator	EtherCAT State
Flashing	INIT, PRE-OP or SAFE-OP
Steady	OPERATIONAL

ERROR

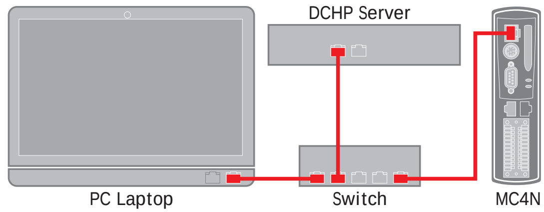
This symbol shows when an error condition has occurred. See the numerical display for more information.

NETWORK SET-UP

NETWORK CONNECTION

Set **IP _ ADDRESS** in MC4N-ECAT to an available unused address. It MUST match the subnet in use. Set the PC to use **DHCP** server.

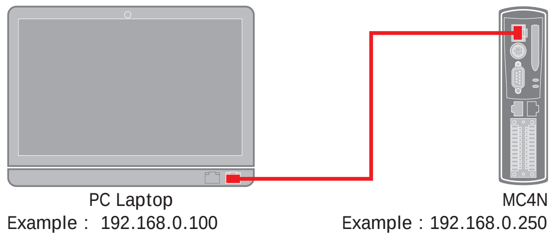
The MC4N always has a fixed **IP _ ADDRESS**.



POINT-TO-POINT OR CLOSED NETWORK

(No DHCP server)

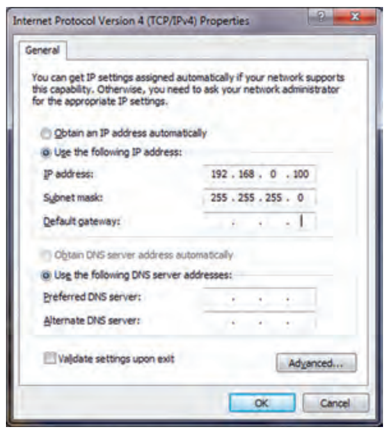
The PC MUST be set to a fixed IP_ADDRESS.



The first 3 “octets” MUST be the same as the MC4N-ECAT and the last MUST be different, but not 000, 254 or 255.

SETTING A FIXED IP ADDRESS

In Windows 7. Open “Network and Sharing Centre” then change “Adapter Settings”. Select the properties of the Local Area Network and the IPv4 properties. The IP Address is set to 192.168.0.100 with subnet mask set to 255.255.255.0. Assuming that the MC4N has **IP _ ADDRESS**=192.168.0.250 or similar.



MC4N FEATURE SUMMARY

Size	157 mm x 40 mm x 120 mm (HxWxD).
Weight	432g
Operating Temp.	0 - 45 degrees C.
Control Inputs	Forward Limit, Reverse Limit, Datum Input, Feedhold Input.
Communication Ports	RS232 channel: up to 38400 baud. RS485 channel: up to 38400 baud. CANbus port (DeviceNet and CANopen compatible) Ethernet: 10/100 BaseT multiple port connection. EtherCAT Port Flexible Axis Port
Position Resolution	64 bit position count.
Speed Resolution	32 bits. Speed may be changed at any time. Moves may be merged.
Servo Cycle	125μs minimum, 1ms default, 2ms max.
Programming	Multi-tasking TrioBASIC system, maximum 22 user processess. IEC 61131-3 programming system.
Interpolation modes	Linear 1-32 axes, circular, helical, spherical, CAM Profiles, speed control, electronic gearboxes.
Memory	8 Mbyte user memory. Automatic flash EPROM program and data storage.
Table	512,000 table positions stored in flash memory.
VR	4096 stored in flash memory.
SD Card	Standard SD Card (FAT 32) compatible to 16Gbytes. Used for storing programs and/or data.
Power Input	24V d.c., Class 2 transformer or power source. 18..29V d.c. at 625mA typical.
Amplifier Enable Output	Normally open solid-state relay rated 24V ac/dc nominal. Maximum load 100mA. Maximum voltage 29V.
Serial / Encoder Power Output	5V at 150mA.
Digital Inputs	8 Opto-isolated 24V inputs.
Digital I/O	8 Opto-isolated 24V outputs. Current sourcing (PNP) 250 mA. (max. 1A per bank of 8).
Product Codes	P900 : MC4N-ECAT 2 Axis P901 : MC4N-ECAT 4 Axis P902 : MC4N-ECAT 8 Axis P903 : MC4N-ECAT 16 Axis P904 : MC4N-ECAT 32 Axis

Motion Coordinator MC4N-Mini RTEX Master

OVERVIEW

The MC4N-RTEX is a new concept in high performance *Motion Coordinator*s which is dedicated to running remote servo and stepper drives via the RTEX Real Time EXpress automation bus. It is based on an up-rated version of the 532MHz ARM 11 processor which makes it ideal for high axis count machines or robotic applications.

It is designed to be configured and programmed for the application using a PC running the *Motion* Perfect application software, and then may be set to run “standalone” if an external computer is not required for the final system.

The Multi-tasking version of TrioBASIC for the MC4N-RTEX allows up to 22 TrioBASIC programs to be run simultaneously on the controller using pre-emptive multi-tasking. In addition, the operating system software includes the IEC 61131-3 standard run-time environment (licence key required).

Versions of the MC4N are available for 2, 4, 8, 16 and 32 motor axes. All versions feature 32 software axes any of which may be used as virtual axes if not assigned to RTEX hardware.

PROGRAMMING

The Multi-tasking ability of the MC4N-RTEX allows parts of a complex application to be developed, tested and run independently, although the tasks can share data and motion control hardware. IEC 61131-3 programs can be run at the same time as TrioBASIC allowing the programmer to select the best features of each.

I/O CAPABILITY

The MC4N-RTEX has 8 built in 24V inputs and 8 bi-directional I/O channels. These may be used for system interaction or may be defined to be used by the controller for end of travel limits, registration, datuming and feedhold functions if required. Each of the Input/Output channels has a status indicator to make it easy to check them at a glance. The MC4N-RTEX can have up 512 external Input/Output channels connected using DIN rail mounted CAN I/O modules. These units connect to the built-in CAN channel.

COMMUNICATIONS

A 10/100 base-T Ethernet port is fitted as standard and this is the primary communications connection to the MC4N-RTEX. Many protocols are supported including Telnet, Modbus TCP, Ethernet IP and TrioPCMotion. Check the Trio website (www.triomotion.com) for a complete list.

The MC4N-RTEX has one built in RS232 port and one built in duplex RS485 channel for simple factory

communication systems. Either the RS232 port or the RS485 port may be configured to run the Modbus or Hostlink protocol for PLC or HMI interfacing.

If the built-in CAN channel is not used for connecting I/O modules, it may optionally be used for CAN communications. E.g. DeviceNet slave or CanOpen master.

REMOVABLE STORAGE

The SD Card maybe used for storing or transfering programs, reciepes and data to and from the MC4N-RTEX. The card must be FAT32 format and a maximum 16Gb size.

SD Cards may be FAT16 formatted when purchased. Re-format in a PC to FAT32 prior to use.

AXIS POSITIONING FUNCTIONS

The motion control generation software receives instructions to move an axis or axes from the TrioBASIC or IEC 61131-3 language which is running concurrently on the same processor. The motion generation software provides control during operation to ensure smooth, coordinated movements with the velocity profiled as specified by the controlling program. Linear interpolation may be performed on groups of axes, and circular, helical or spherical interpolation in any two/three orthogonal axes. Each axis may run independently or they may be linked in any combination using interpolation, CAM profiles or the electronic gearbox facilities.

Consecutive movements may be merged to produce continuous path motion and the user may program the motion using programmable units of measurement (e.g. mm, inches, revs etc.). The module may also be programmed to control only the axis speed. The positioner checks the status of end of travel limit switches which can be used to cancel moves in progress and alter program execution.

CONNECTIONS TO THE MC4N-RTEX

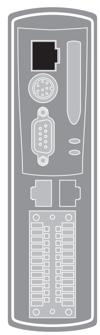
ETHERNET PORT CONNECTION

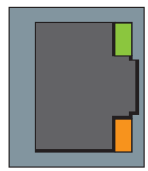
Physical layer: 10/100 base_T

Connector: RJ45

A standard Ethernet connector is provided for use as the primary programming interface.

The Trio programming software, *Motion* Perfect, must be installed on a Windows based PC that is fitted with an Ethernet connection. The IP address is displayed on the MC4N-RTEX display for a few seconds after power-up or when an Ethernet cable is plugged in.



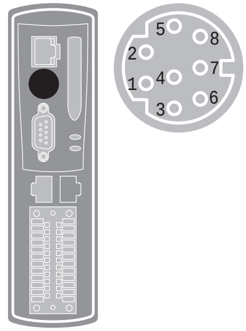


Ethernet cable must be CAT 5 or better.

The Standard Ethernet connection may also be used for Ethernet-IP, Modbus and other factory communications.

SERIAL CONNECTIONS

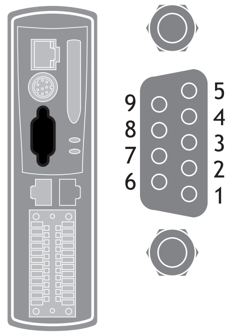
Pin	Function	Note
1	RS485 Data In A Rx+	Serial Port #2
2	RS485 Data In B Rx-	
3	RS232 Transmit	Serial Port #1
4	0V Serial	
5	RS232 Receive	Serial Port #1
6	5V Output	150mA max (Current shared with encoder port)
7	RS485 Data Out Z Tx-	Serial Port #2
8	RS485 Data Out Y Tx+	



FLEXIBLE AXIS PORT

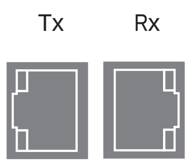
Pin	Encoder	Stepper Axis	Absolute Encoder
1	Enc. A	Step +	Clock
2	Enc. /A	Step -	/Clock
3	Enc. B	Direction +	--------
4	Enc. /B	Direction -	--------
5	0V Serial/Encoder	0V Serial/Encoder	0V Serial/Encoder
6	Enc. Z	Enable +	Data
7	Enc. /Z	Enable -	/Data
8	5V*	5V*	5V*
9	Not Connected	Not Connected	Not Connected

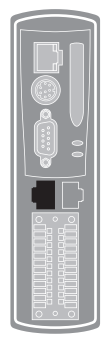
*Current limit is 150mA max. Shared with serial port.



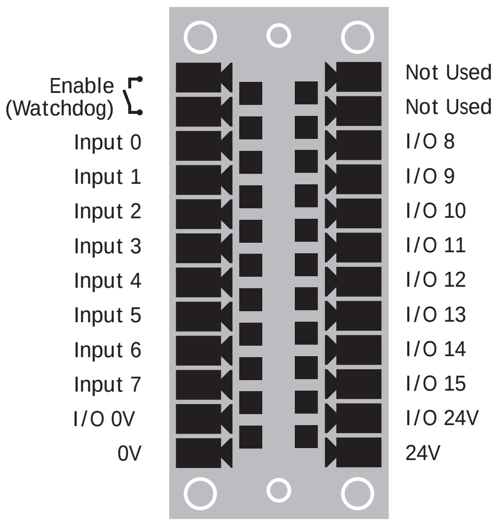
REAL TIME EXPRESS PORT

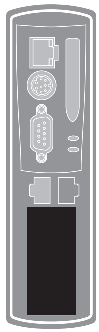
The MC4N-RTEX acts as an Panasonic RTEX master. RTEX drives are normally connected in a ring. Up to 32 RTEX axes may be connected via the RTEX bus.



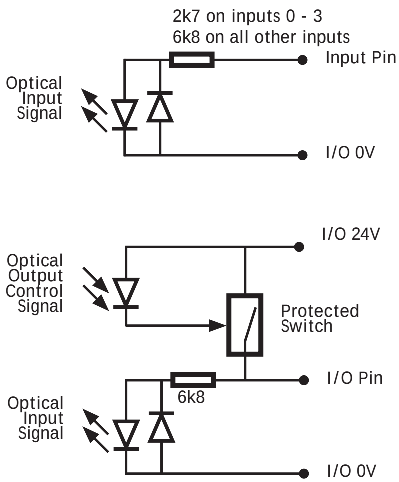


I/O CONNECTOR

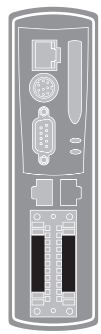




Inputs 0 - 3 have fast opto-couplers for use as axis registration inputs. Inputs 4-7 may also be used as registration inputs.



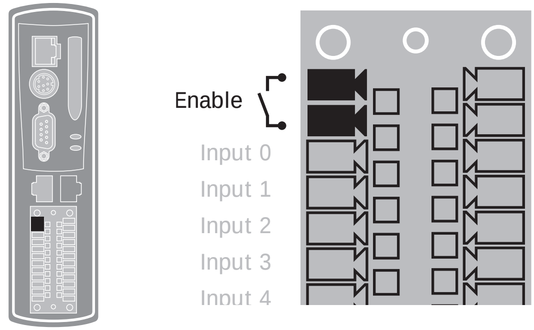
Inputs / Outputs 8 - 15



The MC4N is grounded via the metal chassis. Fit a short shield connection between the chassis earth screw and the earthed metal mounting panel / plate.

AMPLIFIER ENABLE (WATCHDOG) RELAY OUTPUT

An internal relay may be used to enable external amplifiers when the controller has powered up correctly and the system and application software are ready. The amplifier enable is a single pole solid state relay with a normally open “contact”. The enable relay contact will be open circuit if there is no power on the controller OR an axis error exists OR the user program sets it open with the **WDOG=OFF** command.



RTEX drives will be enabled via the RTEX network so the “Amplifier Enable” connection is not normally required.

All non RTEX stepper and servo amplifiers MUST be inhibited when the amplifier enable output is open circuit

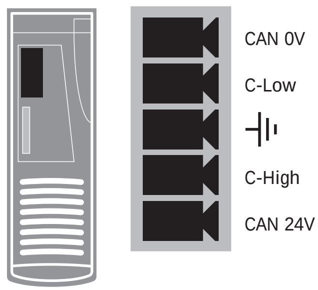
An additional safety relay may be required so as to meet machine safety approvals.

5 WAY CAN CONNECTOR

This is a 5 way 3.5mm pitch connector. The connector is used both to provide the 24 Volt power to the MC4N CAN circuit and provide connections for I/O expansion via Trio’s CAN I/O expanders. A 24V dc, Class 2 transformer or power source should be used.

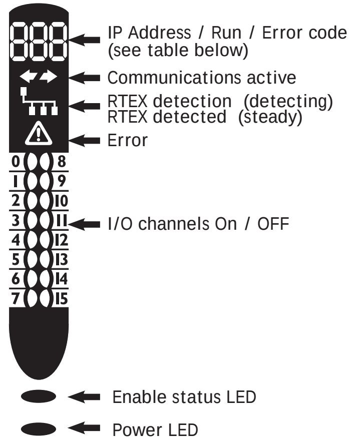
This 24 Volt input is internally isolated from the I/O 24 Volts and main 24V power.

The **CAN** connector may be left unused.



DISPLAY

The IP address and subnet mask of the MC4N is shown on the LCD display for a few seconds after power-up. The factory default IP address is 192.168.0.250. This can be changed using the **IP _ ADDRESS** command via the *Motion* Perfect v3 software tool.



Display Example	Description	Details
SYS	Displayed on controller start	
901	Model code : Displayed on power up	P906 : 2 axes P907 : 4 axes P908 : 8 axes P909 : 16 axes P910: 32 axes
192.168.0.250	IP Address :	Displayed on power up OR after ethernet connection for 15 seconds
Unn	Unit error on slot nn	
Ann	Axis error on axis nn	
Caa	Configuration error on unit aa	ie: too many axes
Run / Off	Enable status	
Err xx	Error codes	Ann : Error on Axis nn Unn : Unit error on slot nn Caa : Configuration error on unit nn, ie: too many axes E04 : VR/TABLE corrupt entry

COMMUNICATIONS ACTIVE

This symbol appears when the firmware has detected one or more valid RTEX nodes on the network.

RTEX DETECTION

This symbol shows the RTEX connection status.

Indicator	RTEX State
Flashing	Detecting Drives
Steady	OPERATIONAL

ERROR

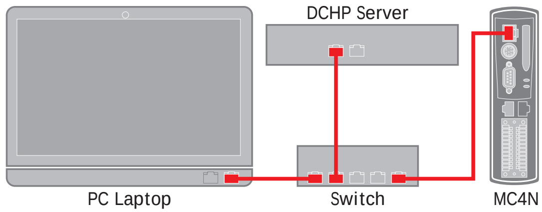
This symbol shows when an error condition has occurred. See the numerical display for more information.

NETWORK SET-UP

NETWORK CONNECTION

Set **IP _ ADDRESS** in MC4N-RTEX to an available unused address. It MUST match the subnet in use. Set the PC to use **DHCP** server.

The MC4N always has a fixed **IP _ ADDRESS**.

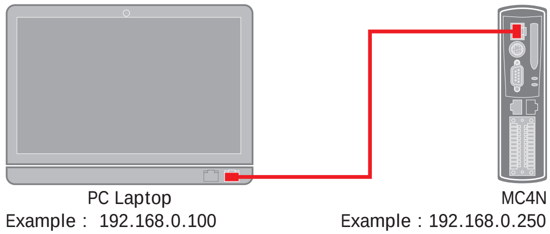


POINT-TO-POINT OR CLOSED NETWORK

(No DHCP server)

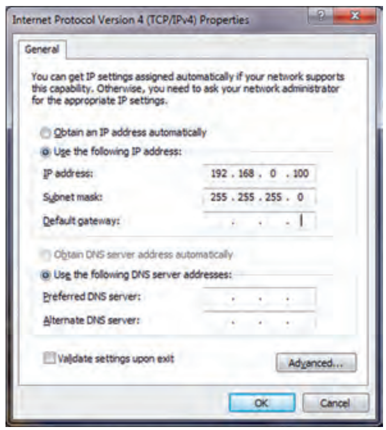
The PC MUST be set to a fixed IP_ADDRESS.

The first 3 “octets” MUST be the same as the MC4N-RTEX and the last MUST be different, but not 000, 254 or 255.



SETTING A FIXED IP ADDRESS

In Windows 7. Open “Network and Sharing Centre” then change “Adapter Settings”. Select the properties of the Local Area Network and the IPv4 properties. The IP Address is set to 192.168.0.100 with subnet mask set to 255.255.255.0. Assuming that the MC4N has **IP _ ADDRESS**=192.168.0.250 or similar.



MC4N-RTEX FEATURE SUMMARY

Size	157 mm x 40 mm x 120 mm (HxWxD).
Weight	432g
Operating Temp.	0 - 45 degrees C.
Control Inputs	Forward Limit, Reverse Limit, Datum Input, Feedhold Input.
Communication Ports	RS232 channel: up to 38400 baud. RS485 channel: up to 38400 baud. CANbus port (DeviceNet and CANopen compatible) Ethernet: 10/100 BaseT multiple port connection. RTEX Port (x2: Tx and Rx) Flexible Axis Port
Position Resolution	64 bit position count.
Speed Resolution	32 bits. Speed may be changed at any time. Moves may be merged.
Servo Cycle	125µs minimum, 1ms default, 2ms max.
Programming	Multi-tasking TrioBASIC system, maximum 22 user processess. IEC 61131-3 programming system.
Interpolation modes	Linear 1-32 axes, circular, helical, spherical, CAM Profiles, speed control, electronic gearboxes.
Memory	8 Mbyte user memory. Automatic flash EPROM program and data storage.
Table	512,000 table positions stored in flash memory.
VR	4096 stored in flash memory.
SD Card	Standard SD Card (FAT 32) compatible to 16Gbytes. Used for storing programs and/or data.
Power Input	24V d.c., Class 2 transformer or power source. 18..29V d.c. at 625mA typical.
Amplifier Enable Output	Normally open solid-state relay rated 24V ac/dc nominal. Maximum load 100mA. Maximum voltage 29V.
Serial / Encoder Power Output	5V at 150mA.
Digital Inputs	8 Opto-isolated 24V inputs.
Digital I/O	8 Opto-isolated 24V outputs. Current sourcing (PNP) 250 mA. (max. 1A per bank of 8).
Product Codes	P906 : MC4N-RTEX 2 Axis P907 : MC4N-RTEX 4 Axis P908 : MC4N-RTEX 8 Axis P909 : MC4N-RTEX 16 Axis P910 : MC4N-RTEX 32 Axis

Motion Coordinator MC403

OVERVIEW

The *Motion Coordinator* MC403 is based on Trio's high-performance ARM11 double-precision technology and provides 2 axes of servo plus a master encoder axis, or 3 axes of pulse+direction control for stepper drives or pulse-input servo drives. Trio uses advanced FPGA techniques to reduce the size and fit the pulse output and servo circuitry in a compact DIN-rail mounted package. The MC403 is housed in a rugged plastic case with integrated earth chassis and incorporates all the isolation circuitry necessary for direct connection to external equipment in an industrial environment. Filtered power supplies are included so that it can be powered from the 24V d.c. logic supply present in most industrial cabinets.

It is designed to be configured and programmed for the application using a PC running Trio's *Motion* Perfect application software, and then may be set to run "standalone" if an external computer is not required for the final system. Programs and data are stored directly to Flash memory, thus eliminating the need for battery backed storage.

The Multi-tasking version of TrioBASIC for the MC403 allows up to 6 TrioBASIC programs to be run simultaneously on the controller using pre-emptive multi-tasking. In addition, the operating system software includes a the IEC 61131-3 standard run-time environment (licence key required).

A reduced functionality version, the MC403-Z has all the fesatures of the full MC403 except that there are no analogue outputs and the encoder function of axes 0 and 1 is incremental encoder only.

PROGRAMMING

The Multi-tasking ability of the MC403 allows parts of a complex application to be developed, tested and run independently, although the tasks can share data and motion control hardware. The 6 available tasks can be used for TrioBASIC or IEC 61131-3 programs, or a combination of both can be run at the same time, thus allowing the programmer to select the best features of each.

I/O CAPABILITY

The MC403 has 8 built in 24V inputs and 4 bi-directional I/O channels. These may be used for system interaction or may be defined to be used by the controller for end of travel limits, registration, datuming and feedhold functions if required. The MC403 can have up 512 external Input and Output channels connected using DIN rail mounted CAN I/O modules. These units connect to the built-in CANbus port.

COMMUNICATIONS

A 10/100 base-T Ethernet port is fitted as standard and this is the primary communications connection to the MC403. Many protocols are supported including Telnet, Modbus TCP, Ethernet IP and TrioPCMotion. Check the Trio website (www.triomotion.com) for a complete list.

The MC403 has one built in RS232 port and one built in duplex RS485 channel for simple factory communication systems. Either the RS232 port or the RS485 port may be configured to run the Modbus or Hostlink protocol for PLC or HMI interfacing.

If the built-in CAN channel is not used for connecting I/O modules, it may optionally be used for CAN communications. E.g. DeviceNet, CANopen etc.

REMOVABLE STORAGE

The MC403 has a micro-SD Card slot which allows a simple means of transferring programs, firmware and data without a PC connection. Offering the OEM easy machine replication and servicing.

The memory slot is compatible with a wide range of micro-SD cards up to 16Gbytes using the FAT32 compatible file system.

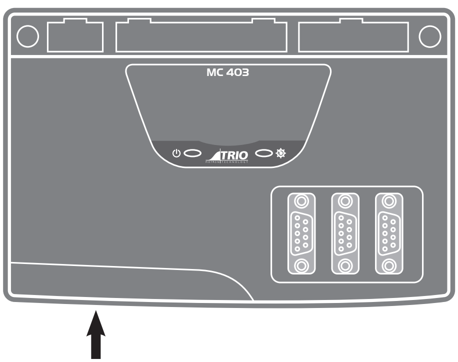
AXIS POSITIONING FUNCTIONS

The motion control software receives instructions to move an axis or axes from the TrioBASIC or IEC 61131-3 language which is running concurrently on the same processor. The motion generation software provides control during operation to ensure smooth, coordinated movements with the velocity profiled as specified by the controlling program. Linear interpolation may be performed on groups of axes, and circular, helical or spherical interpolation in any two/three orthogonal axes. Each axis may run independently or they may be linked in any combination using interpolation, CAM profile or the electronic gearbox facilities.

Consecutive movements may be merged to produce continuous path motion and the user may program the motion using programmable units of measurement (e.g. mm, inches, revs etc.). The module may also be programmed to control only the axis speed. The positioner checks the status of end of travel limit switches which can be used to cancel moves in progress and alter program execution.

CONNECTIONS TO THE MC403

ETHERNET PORT CONNECTION

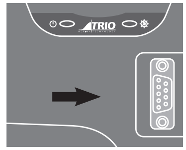


Physical layer: 10/100 base_T

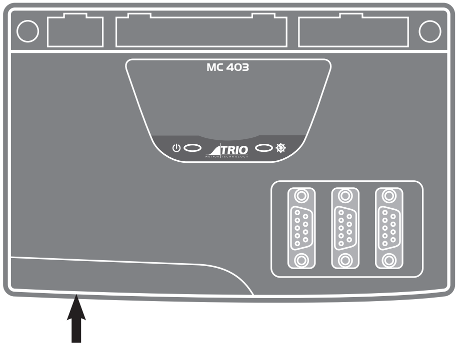
Connector: RJ45

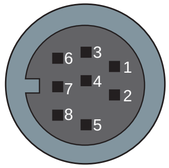
The Ethernet port is the default connection between the *Motion Coordinator* and the host PC running the *Motion* Perfect development application.

To reset the **IP _ ADDRESS**, **IP _ GATEWAY** and **IP _ NETMASK** to their default values press the IP reset button and power cycle the controller while keeping the button pressed.



MC403 SERIAL CONNECTIONS



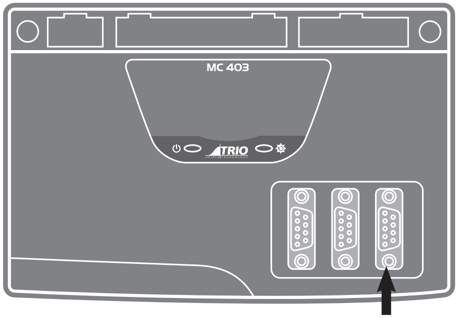


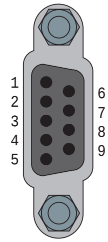
The MC403 features two serial ports. Both ports are accessed through a single 8 pin connector.

SERIAL CONNECTOR

Pin	Function	Note
1	RS485 Data In A Rx+	Serial Port #2
2	RS485 Data In B Rx-	
3	RS232 Transmit	Serial Port #1
4	0V Serial	
5	RS232 Receive	Serial Port #1
6	Internal 5V	5V supply is limited to 150mA, shared with sync port
7	RS485 Data Out Z Tx-	Serial Port #2
8	RS485 Data Out Y Tx+	Serial Port #2

MC403 PULSE OUTPUTS / ENCODER INPUTS





The MC403 is designed to support any combination of servo and pulse input motor drives on the standard controller hardware. The MC403 has 3 versions: 1 axis servo, 2 axis servo and pulse output only. There are also 2 versions of the MC403-Z: 2 axis pulse output and 3 axis pulse output.

Each of the first two axes (0-1) can be enabled as servo(1), pulse and direction or encoder according to the user's requirements by setting the axis **ATYPE** parameter. Axis 2 can be set as either pulse+direction or encoder in all versions.

The function of the 9-pin 'D' connectors will be dependent on the specific axis configuration which has been defined. If the axis is setup as a servo or encoder, the connector will provide the encoder input. If the axis is configured as a pulse+direction, the connector provides differential outputs for step/direction and enable signals.

The encoder port also provides a current-limited 5V output capable of powering most encoders. This simplifies wiring and eliminates external power supplies.

(1) Servo versions of the MC403 only.

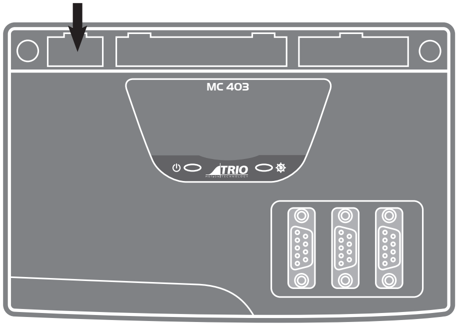
Pin	Function	Pulse & Direction	Absolute Encoder **
1	Enc. A	Step+	Clock+
2	Enc. /A	Step-	Clock-
3	Enc. B	Direction+	N/C
4	Enc. /B	Direction-	N/C
5	0V Encoder	0V Pulse+direction	0V Encoder
6	Enc. Z	Enable+	Data+
7	Enc. /Z	Enable-	Data-
8	5V *	5V*	5V*
9	N/C	N/C	N/C

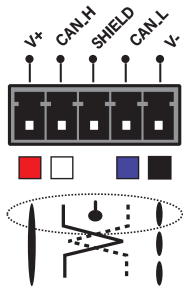
*5V supply is limited to 150mA (shared with serial port)
**Not available on axes 0 and 1 of the MC403-Z

REGISTRATION

Each MC403 encoder port has 2 available registration events. These are assigned in a flexible way to any of the 8 digital inputs or can be used with the Z mark input on the encoder port.

5-WAY CONNECTOR



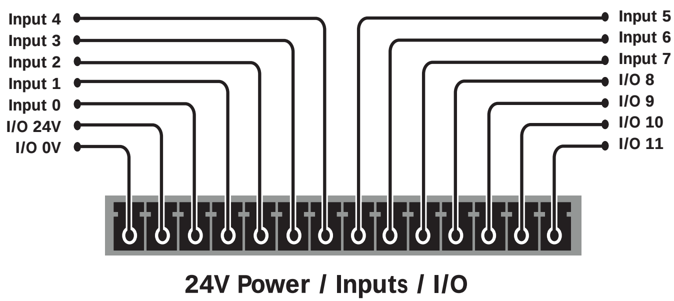


This is a 5 way 3.5 mm pitch connector. The connector is used both to provide the 24 Volt power to the MC403 and provide connections for I/O expansion via Trio's digital and analogue CAN I/O expanders. 24 Volts must be provided as this powers the unit.

This 24 Volt input is internally isolated from the I/O 24 Volts and the +/-10V Voltage outputs.

24V d.c., Class 2 transformer or power source required for **UL** compliance. The MC403 is grounded via the metal chassis. It MUST be installed on an unpainted metal plate or DIN rail which is connected to earth. An earth screw is also provided on the rear of the chassis for bonding the MC403 to ground.

I/O CONNECTOR 1

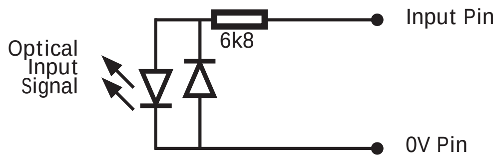


24V INPUT CHANNELS

The MC403 has 8 dedicated 24V Input channels built into the master unit. A further 256 inputs can be provided by the addition of CAN I/O modules. The dedicated input channels are labelled channels 0..7.

Inputs 0 to 7 can be used as registration inputs for axes 0 to 2, using the **REGIST** command.

I/O POWER INPUTS



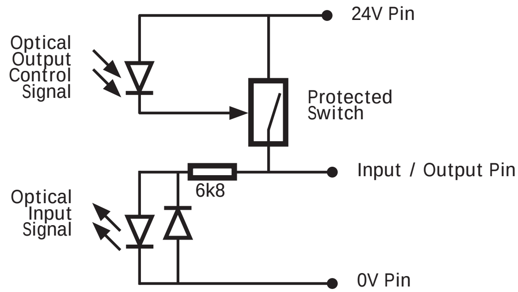
The I/O 0 Volts (I/O-) and I/O 24 Volts (I/O+) are used to power the 24 Volt digital IO and the analogue I/O, including the servo DAC outputs.

The digital I/O connections are isolated from the module power inputs. The analogue inputs and outputs are isolated from the digital I/O and the module power inputs.

24V I/O CHANNELS

Input/output channels 8..11 are bi-directional. The inputs have a protected 24V sourcing output connected to the same pin. If the output is unused it may be used as an input in the program. The input circuitry is the same as on the dedicated inputs. The output circuit has electronic over-current protection and thermal protection which shuts the output down when the current exceeds 250mA.

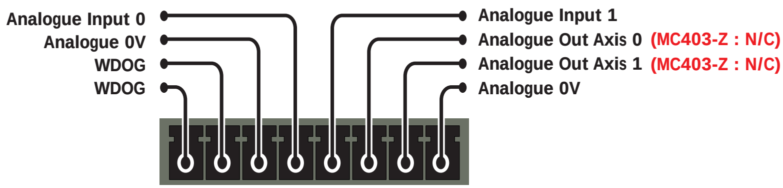
Care should be taken to ensure that the 250mA limit for each output circuit is not exceeded, and that the total load for the group of 4 outputs does not exceed 1 amp.



I/O CONNECTOR 2

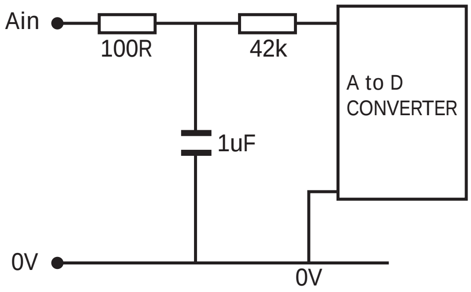
AMPLIFIER ENABLE (WATCHDOG) RELAY OUTPUTS

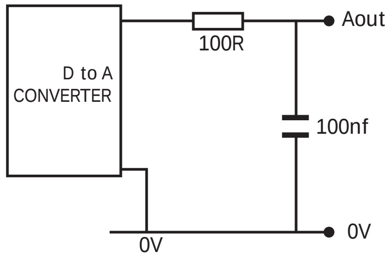
An internal relay contact is available to enable external amplifiers when the controller has powered up correctly and the system and application software is ready. The amplifier enable is a solid-state relay with an ON resistance of 25Ω at 100mA. The enable relay will be open circuit if there is no power on the controller OR a motion error exists on a servo axis OR the user program sets it open with the **WDOG**=OFF command.



WDOG / Analogue Inputs / Outputs

The amplifier 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.





All stepper and servo amplifiers must be inhibited when the amplifier enable output is open circuit

ANALOGUE INPUTS

Two built-in 12 bit analogue inputs are provided which are set up with a scale of 0 to 10V. External connection to these inputs is via the 2-part terminal strip I/O connector 2.

A 24V d.c. supply must be applied to I/O connector 1 to provide power for the analogue input circuit.

ANALOGUE OUTPUTS

The MC403 has 2 12-bit analogue outputs scaled at +/-10V. Each output is assigned to one servo axis, or in the case where the axis is not used, or is set as a pulse+direction/simulated encoder output, the analogue output may be set to a voltage directly in software.

A 24V d.c. supply must be applied to I/O connector 1 to provide power for the analogue output circuit.

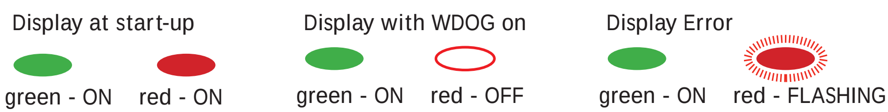
The MC403-Z does not have any analogue outputs.

LED DISPLAY

On power-up, the LEDs flash to show the MC403 version and the SD card status.

P821 2 axis pulse output MC403-Z:	3 flashes of the RED LED.
P822 3 axis pulse output MC403-Z	3 flashes of both LEDs alternately.
P823 3 axis pulse output version:	3 flashes of the RED LED.
P824 2 axis servo version:	3 flashes of both LEDs alternately.
P825 1 axis servo version:	3 flashes of the GREEN LED.
SD card loading system software:	Both LEDs flash together until the system SW load is completed.

During operation, the two LED's show the processor (OK) and system status.



MC403 FEATURE SUMMARY

Size	122 mm x 135 mm x 35 mm (HxWxD).
Weight	325g
Operating Temp.	0 - 45 degrees C.
Control Inputs	Forward Limit, Reverse Limit, Datum Input, Feedhold Input.
Communication Ports	RS232 channel: up to 128k baud. RS485 channel: up to 128k baud. CANbus port (DeviceNet and CANopen compatible). Ethernet: 10/100 BaseT multiple port connection.
Position Resolution	64 bit position count.
Speed Resolution	32 bits. Speed may be changed at any time. Moves may be merged.
Servo Cycle	125µs minimum, 1ms default, 2ms max.
Programming	Multi-tasking TrioBASIC system and IEC 61131-3 programming system. Maximum 6 user processes.
Interpolation modes	Linear 1-3 axes, circular, helical, spherical, CAM Profiles, speed control, electronic gearboxes.
Memory	8 Mbyte user memory. 512,000 x 64 bit TABLE memory. Automatic flash EPROM program and data storage.
VR	4096 global VR data in FLASH memory (automatic-store).
SD Card	Standard micro-SD Card compatible to 16Gbytes. Used for storing programs and/or data.
Power Input	24V d.c., Class 2 transformer or power source. 18..29V d.c. at 300mA + IO supply.
Amplifier Enable Output	Normally open solid-state relay rated 24V ac/dc nominal. Max load 100mA. Max Voltage 29V.
Analogue Inputs	2 isolated, 12 bit, 0 to 10V.
Serial / Encoder Power Output	5V at 150mA. (Max)
Analogue Outputs	2 isolated 12 bit, +/- 10V (MC403 only)
Digital Inputs	8 Opto-isolated 24V inputs.
Digital I/O	4 Opto-isolated 24V outputs. Current sourcing (PNP) 250 mA. (max. 1A per bank of 4).
Product Codes	P821 : MC403-Z 2 axis stepper output / 2 encoder input P822 : MC403-Z 3 axis stepper output / 3 encoder input P823 : MC403 3 axis stepper output / 3 encoder input P824 : MC403 2 axis servo + 1 encoder / 3 axis stepper P825 : MC403 1 axis servo + 1 encoder / 2 axis stepper

MC403 AXIS CONFIGURATION SUMMARY

CONFIGURATION	P823	P824	P825	P821	P822
Axis 0	Core	Extended+AS	Extended+AS	Core	Core
Axis 1	Core	Extended+AS		Core	Core
Axis 2	Core	Extended	Core		Extended
AXES					
# of axes (max)	3	3	2	2	3
# of virtual axes (max)	16	16	16	16	16
DRIVE INTERFACES					
Stepper (Step & Direction)	Yes	Yes	Yes	Yes	Yes
Servo (±10V & Encoder)	No	Yes	Yes	No	No
ENCODER PORTS					
Feedback input	No	Yes	Yes (1 axis)	No	No
Reference input	Yes	Yes	Yes	Yes	Yes
Pulse + direction output	Yes	Yes	Yes	Yes	Yes
Incremental (A+B) output	Yes	Yes	Yes	Yes	Yes
BUILT-IN I/O					
Inputs 24Vdc	8	8	8	8	8
Bi-directional I/O 24Vdc	4	4	4	4	4
0-10V analogue inputs	2x12bit	2x12bit	2x12bit	2x12bit	2x12bit
±10V analogue Outputs	2x12bit	2x12bit	2x12bit	No	No
# registration inputs	6	6	6	6	6
Registration input speed	20μs	20μs	20μs	20μs	20μs

CONFIGURATION KEY

CORE FUNCTIONALITY

CORE AXES - can be configured in software as pulse and direction outputs with stepper or servo drives. They can also be configured for incremental encoder feedback.

Core functionality is a set of ATYPEs (Axis TYPEs) that are available on all controllers. They are based on pulse outputs and incremental encoder feedback.

ATYPE	Description
43	Pulse and direction output with enable output
45	Quadrature encoder output with enable output
63	Pulse and direction output with Z input
64	Quadrature encoder output with Z input

76	Incremental encoder with Z input
78	Pulse and direction with **VFF _ GAIN** and enable output 1

EXTENDED FUNCTIONALITY

EXTENDED AXES – in addition to the Core functionality these axes can also be configured for absolute encoders and closed loop servos (requires voltage output).

ANALOGUE SERVO - Only axes marked as **AS** have an analogue output and can be used for closed loop control.

All Extended Axes can use these **ATYPE**'s as feedback.

If you want to just use the feedback and not complete a closed loop servo system set **SERVO** = OFF

ATYPE	Description
30	Analogue feedback Servo
44	Incremental encoder Servo with Z input
46	Tamagawa absolute Servo
47	Endat absolute Servo
48	SSI absolute Servo
60	Pulse and direction feedback Servo with Z input
77	Incremental encoder Servo with enable output

Motion Coordinator MC405

OVERVIEW

The *Motion Coordinator* MC405 is based on Trio's high-performance ARM11 double-precision technology and provides 4 axes of servo plus a master encoder axis, or 5 axes of pulse+direction control for stepper drives or pulse-input servo drives. Trio uses advanced FPGA techniques to reduce the size and fit the pulse output and servo circuitry in a compact DIN-rail mounted package. The MC405 is housed in a rugged plastic case with integrated earth chassis and incorporates all the isolation circuitry necessary for direct connection to external equipment in an industrial environment. Filtered power supplies are included so that it can be powered from the 24V d.c. logic supply present in most industrial cabinets.

It is designed to be configured and programmed for the application using a PC running Trio's *Motion* Perfect application software, and then may be set to run "standalone" if an external computer is not required for the final system. Programs and data are stored directly to FLASH memory, thus eliminating the need for battery backed storage.

The Multi-tasking version of TrioBASIC for the MC405 allows up to 10 TrioBASIC programs to be run simultaneously on the controller using pre-emptive multi-tasking. In addition, the operating system software includes a the IEC 61131-3 standard run-time environment (licence key required).

PROGRAMMING

The Multi-tasking ability of the MC405 allows parts of a complex application to be developed, tested and run independently, although the tasks can share data and motion control hardware. The 10 available tasks can be used for TrioBASIC or IEC 61131-3 programs, or a combination of both can be run at the same time, thus allowing the programmer to select the best features of each.

I/O CAPABILITY

The MC405 has 8 built in 24V inputs and 8 bi-directional I/O channels. These may be used for system interaction or may be defined to be used by the controller for end of travel limits, registration, datuming and feedhold functions if required. Each of the Input/Output channels has a status indicator to make it easy to check them at a glance. The MC405 can have up 512 external Input and Output channels connected using DIN rail mounted CAN I/O modules. These units connect to the built-in CANbus port.

COMMUNICATIONS

A 10/100 base-T Ethernet port is fitted as standard and this is the primary communications connection to the MC405. Many protocols are supported including Telnet, Modbus TCP, Ethernet IP and TrioPCMotion. Check the Trio website (www.triomotion.com) for a complete list.

The MC405 has one built in RS232 port and one built in duplex RS485 channel for simple factory

communication systems. Either the RS232 port or the RS485 port may be configured to run the Modbus or Hostlink protocol for PLC or HMI interfacing.

If the built-in CAN channel is not used for connecting I/O modules, it may optionally be used for CAN communications. E.g. DeviceNet, CANopen etc.

REMOVABLE STORAGE

The MC405 has a micro-SD Card slot which allows a simple means of transferring programs, firmware and data without a PC connection. Offering the OEM easy machine replication and servicing.

The memory slot is compatible with a wide range of micro-SD cards up to 2Gbytes using the FAT32 compatible file system.

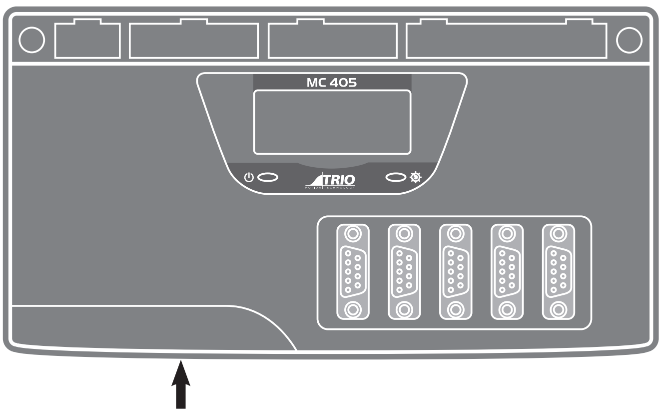
AXIS POSITIONING FUNCTIONS

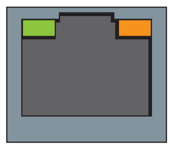
The motion control generation software receives instructions to move an axis or axes from the TrioBASIC or IEC 61131-3 language which is running concurrently on the same processor. The motion generation software provides control during operation to ensure smooth, coordinated movements with the velocity profiled as specified by the controlling program. Linear interpolation may be performed on groups of axes, and circular, helical or spherical interpolation in any two/three orthogonal axes. Each axis may run independently or they may be linked in any combination using interpolation, CAM profile or the electronic gearbox facilities.

Consecutive movements may be merged to produce continuous path motion and the user may program the motion using programmable units of measurement (e.g. mm, inches, revs etc.). The module may also be programmed to control only the axis speed. The positioner checks the status of end of travel limit switches which can be used to cancel moves in progress and alter program execution.

CONNECTIONS TO THE MC405

ETHERNET PORT CONNECTION





Physical layer: 10/100 base_T

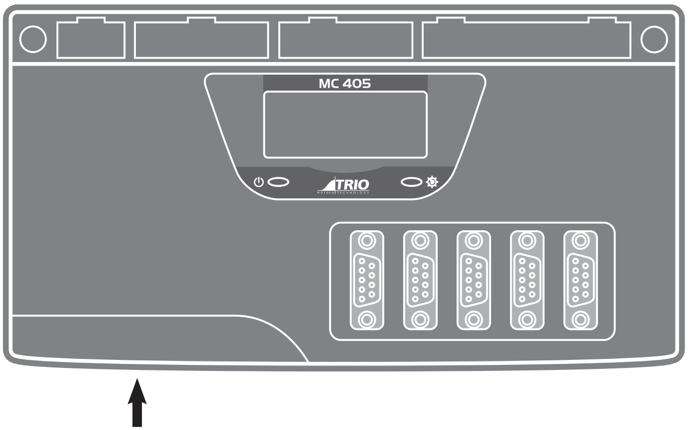
CONNECTOR: RJ45

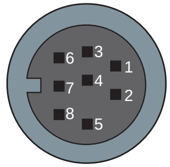
The Ethernet port is the default connection between the *Motion Coordinator* and the host PC running the *Motion* Perfect development application.

MC405 SERIAL CONNECTIONS

The MC405 features two serial ports. Both ports are accessed through a single 8 pin connector.

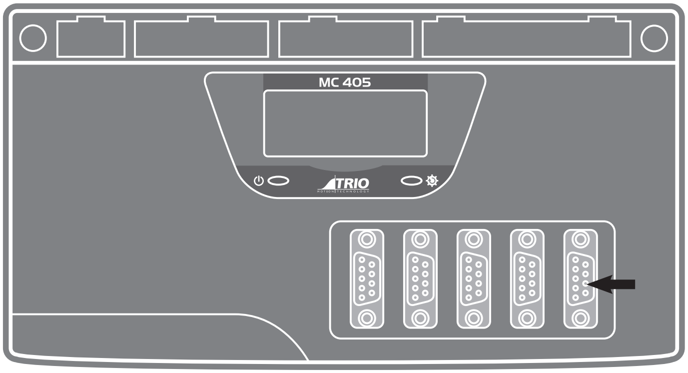
SERIAL CONNECTOR

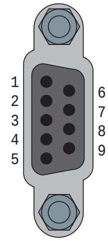




Pin	Function	Note
1	RS485 Data In A Rx+	Serial Port #2
2	RS485 Data In B Rx-	
3	RS232 Transmit	Serial Port #1
4	0V Serial	
5	RS232 Receive	Serial Port #1
6	Internal 5V	5V supply is limited to 150mA, shared with encoder ports
7	RS485 Data Out Z Tx-	Serial Port #2
8	RS485 Data Out Y Tx+	Serial Port #2

MC405 PULSE+DIRECTION OUTPUTS / ENCODER INPUTS





The MC405 is designed to support any combination of servo and pulse driven motor drives on the standard controller hardware. There are 2 versions of the MC405; the servo version and the pulse output only version. In the pulse output only version, only axis 4 can be configured as an encoder input.

Each of the first four axes (0-3) can be enabled as servo(1), pulse output or encoder(1) according to the user's requirements by setting the axis **ATYPE** parameter. Axis 4 can be set as either pulse output, encoder output or encoder input on all versions.

The function of the 9-pin 'D' connectors will be dependent on the specific axis configuration which has been defined. If the axis is setup as a servo, the connector will provide the encoder input(1). If the axis is configured as a pulse output, the connector provides differential outputs for step/direction or simulated encoder, and enable signals.

The encoder port also provides a current-limited 5V output capable of powering most encoders. This simplifies wiring and eliminates external power supplies.

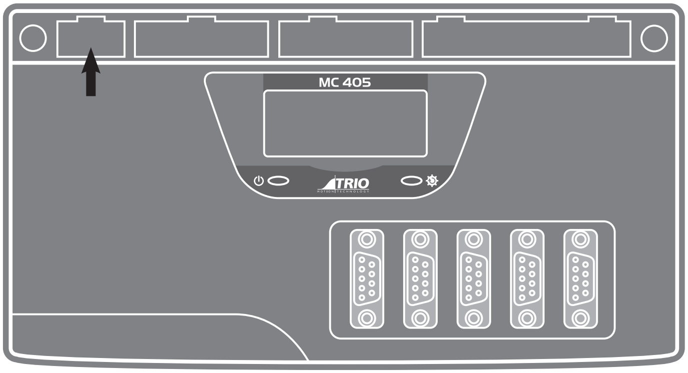
(1) Servo version of the MC405 only.

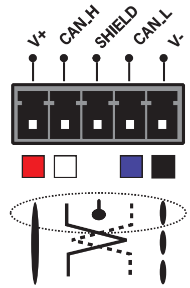
Pin	Encoder in/out	Pulse + Direction	Absolute Encoder
1	Enc. A	Step+	Clock+
2	Enc. /A	Step-	Clock-
3	Enc. B	Direction+	N/C
4	Enc. /B	Direction-	N/C
5	0V Encoder	0V Pulse+direction	0V Encoder
6	Enc. Z	Enable+	Data+
7	Enc. /Z	Enable-	Data-
8	5V *	5V*	5V*
9	N/C	N/C	N/C
*5V supply is limited to 150mA (shared with serial port)			

REGISTRATION

Each MC405 encoder port has 2 available registration events. These are assigned in a flexible way to any of the 8 digital inputs or can be used with the Z mark input on the encoder port.

5-WAY CONNECTOR



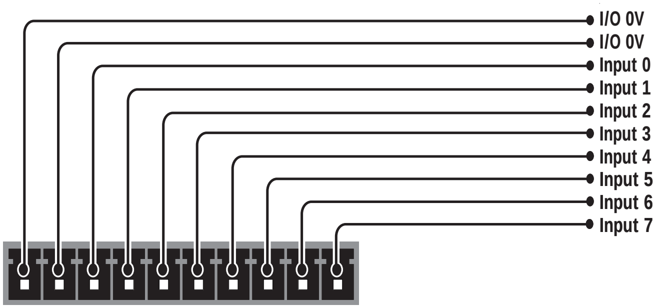


This is a 5 way 3.5 mm pitch connector. The connector is used both to provide the 24 Volt power to the MC405 and provide connections for I/O expansion via Trio’s digital and analogue CAN I/O expanders. 24 Volts must be provided as this powers the unit.

This 24 Volt input is internally isolated from the I/O 24 Volts and the +/-10V voltage outputs.

24V d.c., Class 2 transformer or power source required for UL compliance. The MC405 is grounded via the metal chassis. It MUST be installed on an unpainted metal plate or DIN rail which is connected to earth. An earth screw is also provided on the rear of the chassis for bonding the MC405 to ground.

I/O CONNECTOR 1

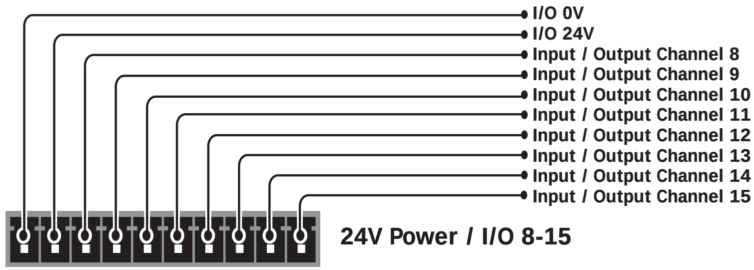


I/O CONNECTOR 2

24V INPUT CHANNELS

The MC405 has 8 dedicated 24V Input channels built into the master unit. A further 256 inputs can be provided by the addition of CAN I/O modules. The dedicated input channels are labelled channels 0..7. Two terminals marked IN- are provided for the input 0V common connections.

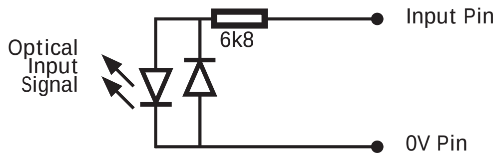
Inputs 0 to 7 can be used as registration inputs for axes 0 to 4, using the **REGIST** command.



I/O POWER INPUTS

The I/O 0 Volts (I/O-) and I/O 24 Volts (I/O+) are used to power the 24 Volt digital IO and the analogue I/O, including the servo DAC outputs.

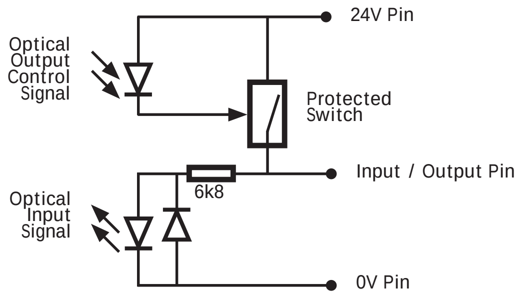
The digital I/O connections are isolated from the module power inputs. The analogue inputs and outputs are isolated from the digital I/O and the module power inputs.



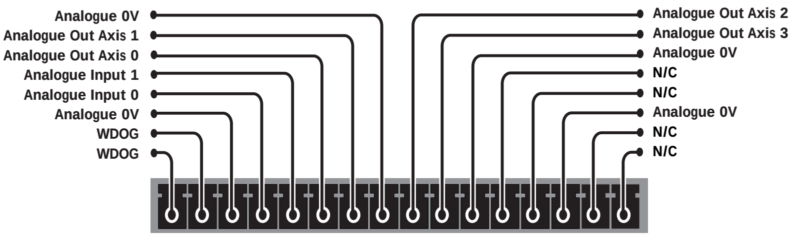
24V I/O CHANNELS

Input/output channels 8..15 are bi-directional. The inputs have a protected 24V sourcing output connected to the same pin. If the output is unused it may be used as an input in the program. The input circuitry is the same as on the dedicated inputs. The output circuit has electronic over-current protection and thermal protection which shuts the output down when the current exceeds 250mA.

Care should be taken to ensure that the 250mA limit for each output circuit is not exceeded, and that the total load for the group of 8 outputs does not exceed 1 amp.



I/O CONNECTOR 3

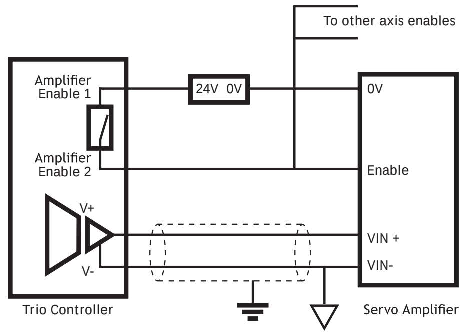


WDOG / Analogue Inputs / Analogue Outputs

AMPLIFIER ENABLE (WATCHDOG) RELAY OUTPUTS

An internal relay contact is available to enable external amplifiers when the controller has powered up correctly and the system and application software is ready. The amplifier enable is a solid-state relay with an ON resistance of 25Ω at 100mA. The enable relay will be open circuit if there is no power on the controller OR a motion error exists on a servo axis OR the user program sets it open with the **WDOG**=OFF command.

The amplifier 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.

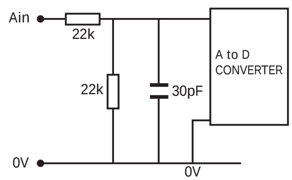


All stepper and servo amplifiers must be inhibited when the amplifier enable output is open circuit

ANALOGUE INPUTS

Two built-in 12 bit analogue inputs are provided which are set up with a scale of 0 to 10V. External connection to these inputs is via the 2-part terminal strip I/O connector 3.

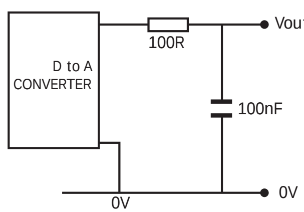
A 24V d.c. supply must be applied to I/O connector 2 to provide power for the analogue input circuit.



ANALOGUE OUTPUTS

The MC405 has 4 12-bit analogue outputs scaled at +/-10V. Each output is assigned to one servo axis, or in the case where the axis is not used, or is set as a pulse+direction/simulated encoder output, the analogue output may be set to a voltage directly in software.

A 24V d.c. supply must be applied to I/O connector 2 to provide power for the analogue output circuit.



BACKLIT DISPLAY

On power-up, the information display area shows bt during the boot process, then the MC405 version is displayed, showing P826 for the 5 axis pulse output version and P827 for the 4 axis servo version. The IP address and subnet mask is shown on power-up and whenever an Ethernet cable is first connected to the MC405.

During operation, this display shows run, Off or Err to indicate the MC405 status. Below the main status display are the **ERROR** and **ENABLE** indicators.

ERROR: An error has occurred (see Error Display Codes table below for details).

ENABLE: When illuminated, WDOG is ON.

A bank of 8 indicators at the left side shows the Digital Input States and a similar bank on the right shows the state of I/O8 to I/O15. The I/O displayed can be altered using the **DISPLAY** command.

Two LED's are provided to show the processor (OK) and system status.

Error Display Codes		
Ann	Axis error on axis nn	
Caa	Configuration error on unit aa	ie: too many axes
Exx	System error	E00 - RAM error 8bit BB - RAM (VR) E01 - RAM error 16 bit BB - RAM (TABLE) E03 - Battery Error E04 - VR/TABLE corrupt entry E05 – Invalid MC_CONFIG file E06 – Started in SAFE mode

MC405 FEATURE SUMMARY

Size	122 mm x 186 mm x 35 mm (HxWxD).
Weight	476g
Operating Temp.	0 - 45 degrees C.
Control Inputs	Forward Limit, Reverse Limit, Datum Input, Feedhold Input.
Communication Ports	RS232 channel: up to 128k baud. RS485 channel: up to 128k baud. CANbus port (DeviceNet and CANopen compatible) Ethernet: 10/100 BaseT multiple port connection.
Position Resolution	64 bit position count.
Speed Resolution	32 bits. Speed may be changed at any time. Moves may be merged.
Servo Cycle	125μs minimum, 1ms default, 2ms max.
Programming	Multi-tasking TrioBASIC system and IEC 61131-3 programming system. Maximum 10 user processes.
Interpolation modes	Linear 1-5 axes, circular, helical, spherical, CAM Profiles, speed control, electronic gearboxes.
Memory	8 Mbyte user memory. 512,000 x 64 bit TABLE memory. Automatic flash EPROM program and data storage.
Real Time Clock	Capacitor backed for 10 days or power off.
VR	4096 global VR data in FLASH memory. (automatic-store)
SD Card	Standard micro-SD Card compatible to 2Gbytes. Used for storing programs and/or data.
Power Input	24V d.c., Class 2 transformer or power source. 18..29V d.c. at 350mA + IO supply.
Amplifier Enable Output	Normally open solid-state relay rated 24V ac/dc nominal. Maximum load 100mA. Maximum Voltage 29V.
Analogue Inputs	2 isolated, 12 bit, 0 to 10V.
Serial / Encoder Power Output	5V at 150mA.
Digital Inputs	8 Opto-isolated 24V inputs.
Digital I/O	8 Opto-isolated 24V outputs. Current sourcing (PNP) 250 mA. (max. 1A per bank of 8).
Product Code	P826 : MC405, 5 axis stepper P827 : MC405, 4 axis servo / 5 axis stepper

Motion Coordinator Euro404 /408

OVERVIEW

The *Motion Coordinator* Euro404 and Euro408 are Eurocard stepper/servo positioners with the built-in ability to control up to 8 servo or stepper motors in any combination. The Euro404 / 408 is designed to provide a powerful yet cost-effective control solution for OEM machine builders who are prepared to mount the unit and provide the power supplies required. It is designed to be configured and programmed for the application with TrioBASIC or IEC61131-3 standard languages using a PC. It may then may be set to run "standalone" if an external computer is not required for the final system. The Multi-tasking version of TrioBASIC for the Euro404 / 408 allows up to 10 TrioBASIC programs to be run simultaneously on the controller using pre-emptive multi-tasking.

PROGRAMMING

The Multi-tasking ability of the Euro404 / 408 allows parts of a complex application to be developed, tested and run independently, although the tasks can share data and motion control hardware.

I/O CAPABILITY

The Euro404 / 408 has 16 built in 24V inputs and 8 built-in output channels. These may be used for system interaction or may be defined to be used by the controller for end of travel limits, datuming and feedhold functions if required. 8 status LEDs are available which can be set to display the status of banks of inputs or outputs. The Euro404 / 408 can have up to 512 external Input/Output channels, up to 32 analogue input channels and up to 16 analogue output channels connected using DIN rail mounted I/O modules. These units connect to the built-in CAN channel of the Euro404 / 408.

COMMUNICATIONS

The Euro404 / 408 has one Ethernet port for primary communications, one RS-232 port and one RS-485 built in.

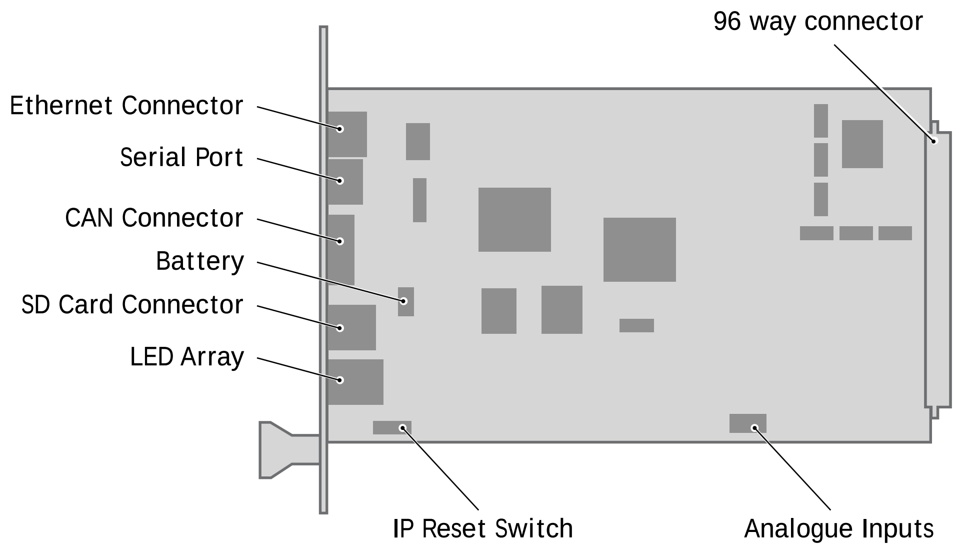
The Ethernet port, RS-232 port or the RS485 port may be configured to run the **MODBUS** protocol for PLC or HMI interfacing. If the built-in CAN channel is not used for connecting I/O modules, it may optionally be used for CAN communications or DeviceNet.

REMOVABLE STORAGE

A micro SD card can be used with the Euro404 / 408 allows a simple means of transferring programs without a PC connection. Offering the OEM easy machine replication and servicing. The Euro404 / 408 supports SD cards up to 16Gbytes. Each Micro SD Card must be pre-formatted using a PC to FAT32 before it can be used in the SD Card Adaptor.

AXIS CONFIGURATION

The Euro404 / 408 is available in 2 configurations. Either as an 8 axis pulse output card or as the full axis servo card.



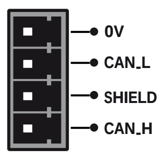
Connections to the Euro404 / 408

5 VOLT POWER SUPPLY

The minimum connections to the Euro404 / 408 are just the 0V and 5V pins. The Euro404 / 408 is protected against reverse polarity on these pins. Application of more than 5.25 Volts will permanently damage the *Motion Coordinator* beyond economic repair. All the 0V are internally connected together and all the 5v pins are internally connected together. The 0V pins are, in addition, internally connected to the **AGND** pins. The Euro404 / 408 has a current consumption of approximately 500mA on the 5V supply. The supply should be filtered and regulated within 5%.

BUILT-IN CAN CONNECTOR

The Euro404 / 408 features a built-in CAN channel. This is primarily intended for Input/ Output expansion via Trio's CAN I/O modules. It may be used for other purposes when I/O expansion is not required.



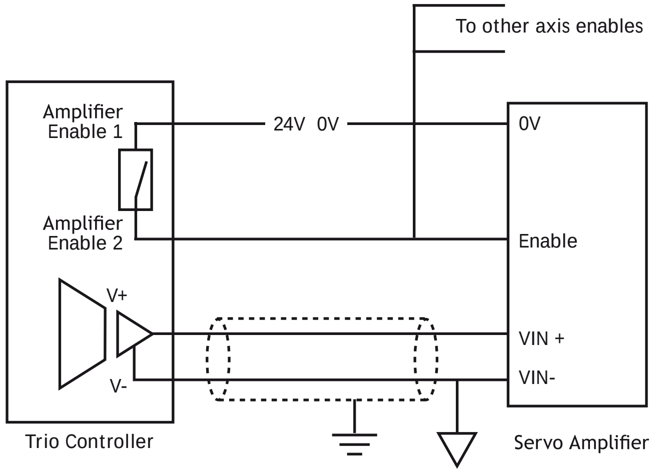
EURO404 / 408 BACKPLANE CONNECTOR

Most connections to the Euro404 / 408 are made via the 96 Way DIN41612 backplane Connector.

Euro408	C	B	A
1	5V	5V	5V
2	5V	5V	5V
3	0V	0V	0V
4	IO GND	OP13	OP10
5	OP9	OP12	OP15
6	OP8	OP11	OP14
7	IO 24V	IN0 / R0	IN1 / R1
8	IN2 / R2	IN3 / R3	IN4 / R4
9	IN5 / R5	IN6 / R6	IN7 / R7
10	IN8	IN9	IN10
11	IN11	IN12	N13
12	IN14	0V	IN15
13	A7- / STEP7-	B7- / DIR7-	Z7- / ENABLE7-
14	A7+ / STEP7+	B7+ / DIR7+	Z7+ / ENABLE7+
15	A6- / STEP6-	B6- / DIR6-	Z6- / ENABLE6-
16	A6+ / STEP6+	B6+ / DIR6+	Z6+ / ENABLE6+
17	A5- / STEP5-	B5- / DIR5-	Z5- / ENABLE5-
18	A5+ / STEP5+	B5+ / DIR5+	Z5+ / ENABLE5+
19	A4- / STEP4-	B4- / DIR4-	Z4- / ENABLE4-
20	A4+ / STEP4+	B4+ / DIR4+	Z4+ / ENABLE4+
21	A3- / STEP3-	B3- / DIR3-	Z3- / ENABLE3-
22	A3+ / STEP3+	B3+ / DIR3+	Z3+ / ENABLE3+
23	A2- / STEP2-	B2- / DIR2-	Z2- / ENABLE2-
24	A2+ / STEP2+	B2+ / DIR2+	Z2+ / ENABLE2+
25	A1- / STEP1-	B1- / DIR1-	Z1- / ENABLE1-
26	A1+ / STEP1+	B1+ / DIR1+	Z1+ / ENABLE1+
27	A0- / STEP0-	B0- / DIR-	Z0- / ENABLE0-
28	A0+ / STEP0+	B0+ / DIR+	Z0+ / ENABLE0+
29	VOUT7	VOUT6	VOUT5
30	AGND	VOUT4	VOUT3
31	VOUT2	VOUT1	VOUT0
32	ENABLE1	ENABLE2	Earth

Euro404	C	B	A
1	5V	5V	5V
2	5V	5V	5V
3	0V	0V	0V
4	IO GND	OP13	OP10
5	OP9	OP12	OP15
6	OP8	OP11	OP14
7	IO 24V	IN0 / R0	IN1 / R1
8	IN2 / R2	IN3 / R3	IN4 / R4
9	IN5 / R5	IN6 / R6	IN7 / R7
10	IN8	IN9	IN10
11	IN11	IN12	N13
12	IN14	0V	IN15
13	N/C	N/C	N/C
14	N/C	N/C	N/C
15	N/C	N/C	N/C
16	N/C	N/C	N/C
17	N/C	N/C	N/C
18	N/C	N/C	N/C
19	N/C	N/C	N/C
20	N/C	N/C	N/C
21	A3- / STEP3-	B3- / DIR3-	Z3- / ENABLE3-
22	A3+ / STEP3+	B3+ / DIR3+	Z3+ / ENABLE3+
23	A2- / STEP2-	B2- / DIR2-	Z2- / ENABLE2-
24	A2+ / STEP2+	B2+ / DIR2+	Z2+ / ENABLE2+
25	A1- / STEP1-	B1- / DIR1-	Z1- / ENABLE1-
26	A1+ / STEP1+	B1+ / DIR1+	Z1+ / ENABLE1+
27	A0- / STEP0-	B0- / DIR-	Z0- / ENABLE0-
28	A0+ / STEP0+	B0+ / DIR+	Z0+ / ENABLE0+
29	N/C	N/C	N/C
30	AGND	N/C	VOUT3
31	VOUT2	VOUT1	VOUT0
32	ENABLE1	ENABLE2	Earth

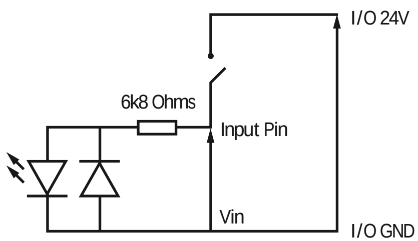
AMPLIFIER ENABLE (WATCHDOG) RELAY OUTPUT



An internal relay contact is used to enable external amplifiers when the controller has powered up correctly and the system and application software is ready. The amplifier enable is a solid-state relay on the Euro404 / 408 with normally open "contacts". The enable relay 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 amplifier 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.

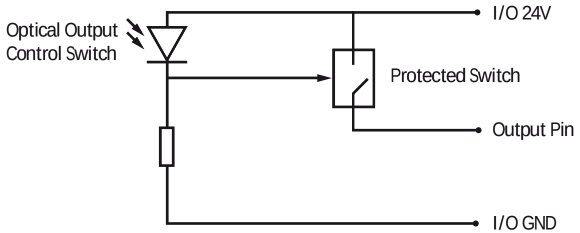
Note: all stepper and servo amplifiers MUST be inhibited when the amplifier enable output is open circuit

24V INPUT CHANNELS



The *Motion Coordinator* has 16 24V Input channels built into the master unit. These may be expanded to 256 Inputs by the addition of CAN-16 I/O modules.

24V OUTPUT CHANNELS



8 output channels are provided. These channels are labelled 8..15 for compatibility with other *Motion Coordinators*, but are NOT bi-directional as on some *Motion Coordinators*. Each channel has a protected 24v sourcing output. The output circuit has electronic over-current protection and thermal protection which shuts the output down when the current exceeds 250mA. Care should still be taken to ensure that the 250mA limit for the output circuit is not exceeded, and that the total load for the group of 8 outputs does not exceed 1 amp. Up to 256 further Outputs may be added by the addition of CAN-16I/O modules).

REGISTRATION INPUTS

The registration inputs are 24 Volt isolated inputs that are shared with digital inputs 0 to 7. The Euro404 / 408 can be programmed to capture the position of an encoder axis in hardware when a transition occurs on the registration input.

DIFFERENTIAL ENCODER INPUTS

The encoder inputs on the Euro404 / 408 are designed to be directly connected to 5 Volt differential output encoders. Incremental or absolute encoders can be connected to the ports.

The encoder ports are also bi-directional so that when axes are set to pulse and direction, the encoder port for that axis becomes a Differential output.

Encoder ports and pulse direction ports on the Euro404 / 408 are NOT electrically isolated.

VOLTAGE OUTPUTS

The Euro404 can generate up to 4 +/-10Volt analogue outputs and the Euro408 can generate up to 8 +/-10Volt analogue outputs for controlling servo-amplifiers. Note that for servo operation the card must be configured as a 4 or 8 axis servo. However, the voltage outputs can be used seperately via the DAC command in TrioBASIC even when the servo axis is not enabled.

ANALOGUE INPUTS

Two built-in 12 bit analogue inputs are provided which are set up with a scale of 0 to 10 Volts. In order to make connection to these inputs, there is a 2 part molex connector behind the front panel. Pin 1 is nearest the front panel.

Pin 1	AIN(32)	Mating MOLEX connector part number
Pin 2	AIN(33)	Connector housing: 22-01-2035
Pin 3	0V	Crimp receptacles : 08-50-0032 (3 required)

USING END OF TRAVEL LIMIT SENSORS

Each axis of the *Motion Coordinator* system may have a 24v Input channel allocated to it for the functions:

FORWARD Limit	Forward end of travel limit
REVERSE Limit	Reverse end of travel limit
DATUM Input	Used in datuming sequence
FEEDHOLD Input	Used to suspend velocity profiled movements until the input is released

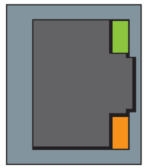
Switches used for the **FORWARD**/**REVERSE**/**DATUM**/**FEEDHOLD** inputs may be normally closed or normally open but the NORMALLY CLOSED type is recommended.

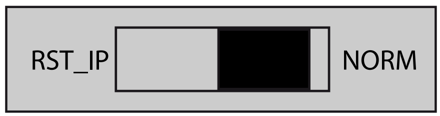
Each of the functions is optional and may be left unused if not required. Each of the 4 functions are available for each axis and can be assigned to any input channel iincluding remote CAN I/O. An input can be assigned to more than one function if desired.

The axis parameters: **FWD _ IN**,**REV _ IN**, **DATUM _ IN** and **FH _ IN** are used to assign input channels to the functions. The axis parameters are set to -1 if the function is not required.

ETHERNET PORT CONNECTION

Pysical layer: 10/100 baseT
Connector: RJ-45
Connection and activity LED indicators
Fixed IP address
User settable subnet mask and default gateway
DHCP client: Not available (fixed IP only)





A switch is provided on the board to reset the IP address to a known value. To reset to the default value of 192.168.000.250, slide the switch to the left (RST_IP) and power up the Euro404 / 408. Make connection with the Euro404 / 408 using *Motion* Perfect on the default address and use the **IP _ ADDRESS** command to set the required address. e.g. for 192.168.000.123 set **IP _ ADDRESS=192.168.0.123**.

NOTE: The switch also sets the following:

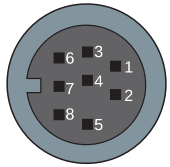
subnet mask to 255.255.255.0
default gateway to 192.168.0.255

Once the IP address has been set, slide switch 1 to NORM and power down the Eurocard. Next time the Euro404 / 408 is powered up, the new IP address can be used.

SERIAL CONNECTOR B:

Euro404 / 408 Serial Port Connections

Pin	Function	Note
1	RS485 Data In A Rx+	Serial Port #2
2	$\overline{\text{RS485 Data In}}$ B Rx-	
3	RS232 Transmit	Serial Port #1
4	Serial 0V	
5	RS232 Receive	
6	5V OUT	Serial Port #2
7	$\overline{\text{RS485 Data Out}}$ Z Tx-	
8	RS485 Data Out Y Tx+	



EURO404 / 408 - FEATURE SUMMARY

Size	170 mm x 129 mm Overall (160mm x 100 mm PCB) 25mm deep
Weight	160 g
Operating Temp.	0 - 45 degrees C
Control Inputs	Forward Limit, Reverse Limit, Datum Input, Feedhold Input.
Communication Ports	RS232 channel: up to 128k baud. RS485 channel: up to 128k baud. CANbus port (DeviceNet and CANopen compatible) Ethernet: 10/100 BaseT multiple port connection.
Position Resolution	64 bit position count
Speed Resolution	32 bits. Speed may be changed at any time. Moves may be merged.
Interpolation modes	Linear 1-8 axes, circular, helical, CAM Profiles, speed control, electronic gearboxes.
Programming	Multi-tasking TrioBASIC system, maximum 10 user tasks. IEC61131-3 programming languages.
Servo Cycle	125µs minimum, 1ms default, 2ms max.
Memory	8 Mbyte user memory. 512,000 x 64 bit TABLE memory. Automatic flash EPROM program and data storage.
Real Time Clock	Capacitor backed for 10 days or power off.
VR	4096 global VR data in FLASH memory. (automatic-store)
Expansion Memory	Socket for Micro SD Card. Used for storing programs and/or data. Format: FAT32, up to 16 GBytes.
Power Input	600mA at 5V d.c.
Amplifier Enable Output	Normally open solid-state relay. Maximim load 100mA, maximum voltage 29V.
Analogue Outputs	4 Isolated 12 bit +/-10V or 8 isolated 12 bit +/-10V.
Analogue Inputs	2 x 12 bit 0 to 10V
Digital Inputs	16 Opto-isolated 24V inputs
Registration Inputs	8 shared with inputs 0 to 7.
Encoder Inputs	4 / 8 differential 5V inputs, 6MHz maximum edge rate
Stepper Outputs	4 / 8 differential step / direction outputs 2MHz max rate
Digital Outputs	8 Opto-isolated 24V outputs. Current sourcing (PNP) 250 mA. (max. 1A per bank of 8)
Product Code	P831 : Euro404, 4 axis stepper P832 : Euro404, 4 axis servo P833 : Euro408, 8 axis stepper P834 : Euro404, 8 axis servo

MC464 EXPANSION MODULES

3

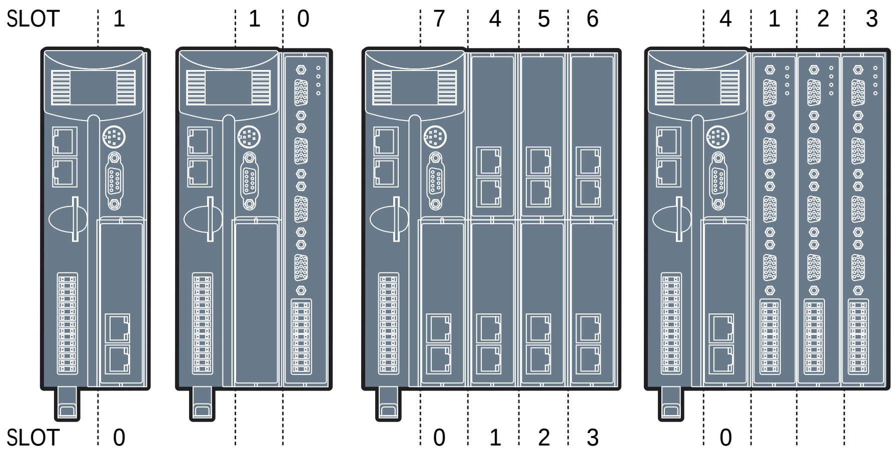
MC664 / MC464 Expansion Modules

Assembly

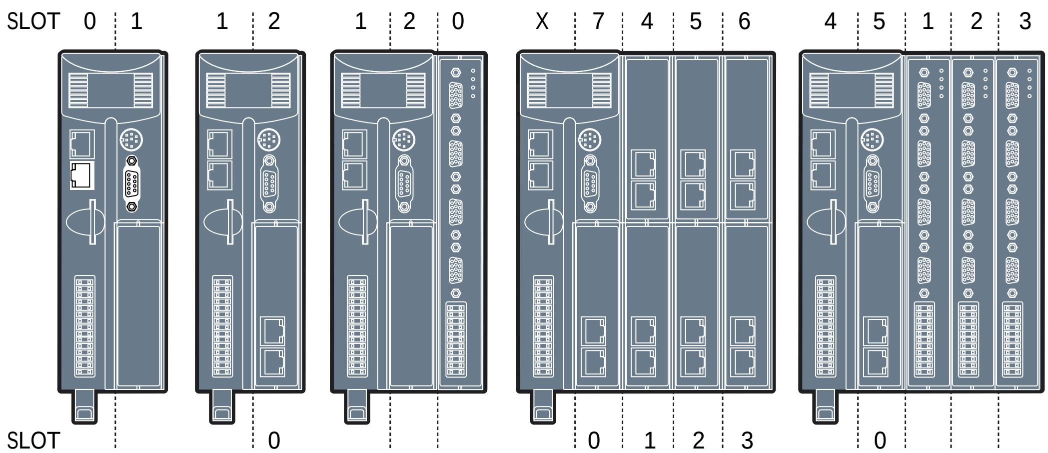
A maximum of 7 half height modules or 3 full height modules may be fitted to the MC664 and MC464. A system may be made using any combination of half and full height modules providing that the full height modules are the last to be attached.

MODULE SLOT NUMBERS

SLOT Numbers are allocated by the system software in order, left to right, starting with the lower bus. Lower modules are allocated slots 0 to m, then the upper modules become slots m+1 to n. Finally, the Sync Encoder Port is allocated slot n+1. The Sync Encoder Port has SLOT number -1 in addition to the one allocated (1) in this sequence.



MC464



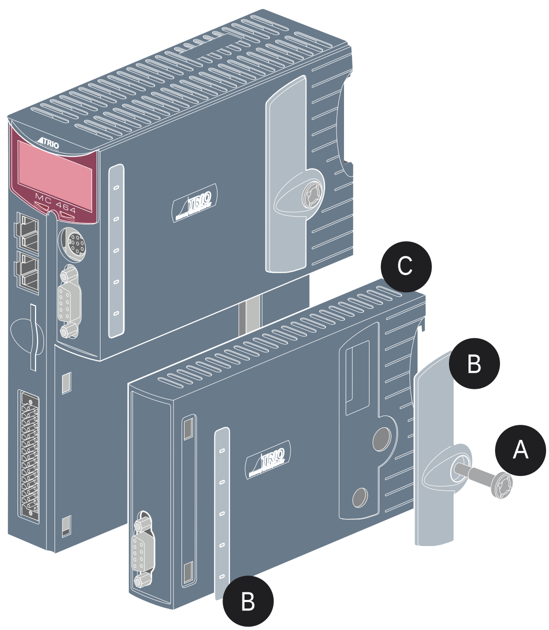
MC664 / MC664-X

FITTING EXPANSION MODULES

- Remove the 2 covers (B) if fitted to the MC664 or MC464 or to the previous expansion module (C).
- Locate the 2 hooks at the front of the module, while holding the rear out at an angle
- Push forward to engage the hooks and at the same time swing the rear of the module in so as to locate the connector.
- Press the connector “home” once it is located.
- Tighten the screw (A) using the tool provided or a small coin
- Clip the provided covers (B) in place as shown.

Removing modules is the reversal of the above procedure.

If the system is to be panel mounted, a kit (P8) comprising 2 x panel mounting brackets and 2 x countersunk screws may be purchaced separately from your Trio distributor.



RTEX Interface (P871)

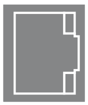
For use with Panasonic amplifiers supporting the Panasonic Real Time Express (RTEX) network. Allows Plug & Play interconnection with Shielded twisted pair (TIA/EIA-568B CAT5e or more) Ethernet cables.

A single interface supports up to 32 axes on the RTEX network. The module comes with 2 axes enabled. Further axes can be enabled with Trio's Feature Enable Codes.

REALTIME EXPRESS

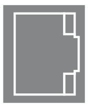
The P871 communicates with up to 32 servo amplifiers using Ethernet Real Time Express. The physical layer is standard Ethernet connected in a ring. Each node has a transmit socket and a receive socket to allow easy connection. The maxium cable length between any 2 nodes is 60 meters and the overall network length is limited to 200 meters.

RJ45 CONNECTOR (TX)



(Top connector)100Mbps Panasonic RTEX transmit - connect to receive of first drive.

RJ45 CONNECTOR (RX)



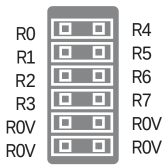
(Bottom connector) 100Mbps Panasonic RTEX receive – connect to transmit of last drive.

TIME BASED REGISTRATION

Time based registration uses a 10MHz clock to record the time of a registration event which is then referenced to time stamps on the axis position from the digital drive network. An accurate registration position is then calculated. The 10MHz clock gives a time resolution of 100nsec. The position and speed of the axis are recorded so that the user can compensate for any fixed delays in the registration circuit.

Any time based registration input can be assigned to any Digital or Virtual axis. This makes the registration very flexible and enables multiple registration channels per axis. Each registration channel can be armed independently and assigned to an axis at any time.

REGISTRATION CONNECTOR



R0-R7 registration inputs (24V).
0V common 0V return.
Registration inputs can be allocated to any axis by software.

LED FUNCTIONS

LED	LED colour	LED function
ok	Green	ON=Module Initialised Okay
0	Red	ON=Module Error
1	Yellow	Status 1
2	Yellow	Status 2

Sercos Interface (P872)

The sercos interface module is designed to control up to 16 servo amplifiers using the standard sercos fibre-optic ring. Benefits of this system include full isolation from the amplifiers and greatly reduced wiring.

For use with any sercos IEC61491 compliant drive. The module allows control of up to 16 axes via sercos with cycle times down to 250usec. Multiple sercos interface modules can be used to increase axes count to 64.

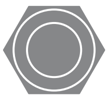
2, 4, 8 and 16 Mbit / sec

Software settable intensity

SERCOS CONNECTIONS

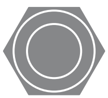
Sercos is connected by 1mm polymer or glass fibre optic cable terminated with 9mm FSMA connectors. The sercos ring is completed by connecting TX to RX in a series loop. The maximum fibre cable length between 2 nodes is 40m for plastic optical fibre (POF) and 200m for hard clad silica (HCS). The total length for POF is 680m and 3,400 for HCS.

CONNECTOR (RX)



(Top connector) sercos fibre-optic transmit. 9mm FSMA.

CONNECTOR (TX)



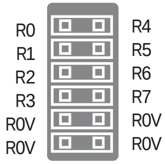
(Bottom connector) sercos fibre-optic receive. 9mm FSMA.

TIME BASED REGISTRATION

Time based registration uses a 10MHz clock to record the time of a registration event which is then referenced to time stamps on the axis position from the digital drive network. An accurate registration position is then calculated. The 10MHz clock gives a time resolution of 100nsec. The position and speed of the axis are recorded so that the user can compensate for any fixed delays in the registration circuit.

Any time based registration input can be assigned to any Digital or Virtual axis. This makes the registration very flexible and enables multiple registration channels per axis. Each registration channel can be armed independently and assigned to an axis on the fly.

REGISTRATION CONNECTOR



R0 - R7 registration inputs (24V).
R0V registration common 0V return.
Registration inputs can be allocated to any axis by software.

LED FUNCTIONS

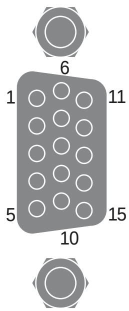
LED	LED colour	LED function
ok	Green	ON=Module Initialised Okay
0	Red	ON=Module Error
1	Yellow	Status 1
2	Yellow	Status 2

sercos phase	LED 1	LED 2
0	OFF	FLASH
1	OFF	ON
2	FLASH	OFF 1
3	ON	OFF 2
4	ON	ON

SLM Interface (P873)

For use with drives supporting the Control Techniques SLM protocol. Each module supports 6 axes which can be individual drives or two drives using the CT Multiax concept.

SLM CONNECTOR



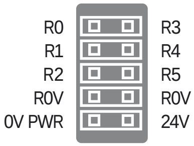
Pin	Upper D-Type	Lower D-Type
1	Com Axis 0	Com Axis 3
2	/Com Axis 0	/Com Axis 3
3	Hardware Enable	Hardware Enable
4	0V Output	0V Output
5	24V Output	24V Output
6	Com Axis 1	Com Axis 4
7	/Com Axis 1	/Com Axis 43
8	No Connection	No Connection
9	No Connection	No Connection
10	No Connection	No Connection
11	24V Output	24V Output
12	0V Output	0V Output
13	Com Axis 2	Com Axis 5
14	/Com Axis 2	/Com Axis 5
15	Earth / Shield	Earth / Shield

TIME BASED REGISTRATION

Time based registration uses a 10MHz clock to record the time of a registration event which is then referenced to time stamps on the axis position from the digital drive network. An accurate registration position is then calculated. The 10MHz clock gives a time resolution of 100nsec. The position and speed of the axis are recorded so that the user can compensate for any fixed delays in the registration circuit.

Any time based registration input can be assigned to any Digital or Virtual axis. This makes the registration very flexible and enables multiple registration channels per axis. Each registration channel can be armed independently and assigned to an axis on the fly.

REGISTRATION CONNECTOR



R0 - R5	registration inputs (24V).
0VR	common 0V return.
0V PWR	Power input for SLM system.
24V	Power input for SLM system.

LED FUNCTIONS

LED	LED Colour	LED Function
ok	Green	ON = Module initalised ok
0	Red	ON = Module error
1	Yellow	Status 1
2	Yellow	Status 2

FlexAxis Interface (P874 / P879)

For use with Stepper, Analogue Servo & Piezo motors. The FlexAxis Interface is available in 4 axes (P879) and 8 axes (P874) versions.

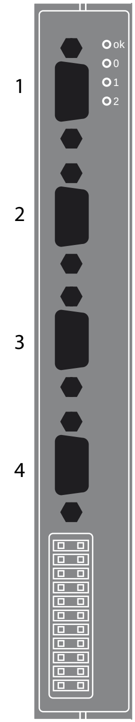
Each axis provides a 16 bit analogue output, up to 8 x 24Vdc high speed registration inputs and a 6MHz encoder input. The encoder port can be configured to drive a stepper motor or an encoder simulation port, both at 2MHz.

ENCODER CONNECTOR

Pin	Incremental Encoder	Pulse + Direction	Absolute Encoder
1	Enc. A n	Step+ n	Clock+ n
2	Enc. /A n	Step- n	Clock- n
3	Enc. B n	Direction+ n	n/c
4	Enc. /B n	Direction- n	n/c
5	0V Enc	0V Enc	0V Enc
6	Enc. Z n	Enable+ n	Data+ n
7	Enc. /Z n	Enable- n	Data- n
8	5V*	5V*	5V*
9	Enc A n+4	Step+ n4	Clock+ n+4
10	Enc /A n+4	Step- n4	Clock- n+4
11	Enc B n+4	Direction+ n+4	n/c
12	Enc /B n+4	Direction- n+4	n/c
13	Enc Z n+4	Enable+ n+4	Data+ n+4
14	Enc /Z n+4	Enable- n+4	Data- n+4
15	0V Enc	0V Enc	0V Enc
*5V supply is limited to 150mA per axis.			

Absolute encoder is only available on axes 4-7 on the P874 and on axes 2-3 on P879.

Connector	8 Axes (P874)	4 Axes (P879)
1	0 and 4	0
2	1 and 5	1
3	2 and 6	2
4	3 and 7	3



MULTIFUNCTION CONNECTOR

The 22 pin multifunction connector provides terminals for 8 registration inputs, 8 voltage outputs and 4 hardware PSWITCH outputs.

ANALOGUE OUTPUTS

8 +/-10V 16Bit analogue outputs are available for servo axis control (4 in the P879). Connect V0 as the velocity command signal for the first axis, V1 for the second axis and so on. The maximum load per axis together is 10mA.

POSITION BASED REGISTRATION

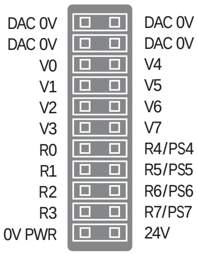
Position based registration uses the encoder signal. When the registration event occurs the encoder position is latched in hardware. The speed of the axis is also recorded so that the user can compensate for any fixed electronic delays in the registration circuit. Flexible allocation of registration inputs to axes is provided. Each axis can have a number of registration events assigned to it and the source of these events can be from any of the registration channels.

The Flex Axis module has 8 registration inputs in addition to the Z mark for each axis. The first axis has 8 registration events which can be assigned to use any of the registration inputs or its own Z mark. The remaining axes have 2 registration events which can be assigned to use any of the registration inputs or their own Z mark.

PSWITCH OUTPUTS

Inputs R4 to R7 are bi-directional and can be used as outputs for high accuracy PSWITCH operation. When used in this mode, the outputs are controlled by the position value of an axis within the same P874 / P879 module.

MULTIFUNCTION CONNECTOR PIN OUT



0V	DAC common 0V return
V0 - V7	Voltage outputs
R0 - R3	24V Registration Inputs
R4/PS4 - R7/PS7	Bidirectional 24V registration In/24V: PSWITCH outputs
Inputs / 24V	PSwitch outputs
0V PWR	Power Input
24V	Power Input

4 axis version uses voltage outputs V0 - V3 only.

Special versions are available for the 8 axis **SSI** and BiSS encoders.

LED FUNCTIONS

LED	LED Colour	LED Function
ok	Green	ON = Module initalised ok
0	Red	ON = Module Error
1	Yellow	Status 1
2	Yellow	Status 2

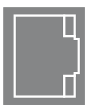
EtherCAT Interface (P876)

For use with EtherCAT compliant drives, this module allows control of up to 64 axes via standard shielded twisted pair (TIA/EIA-568B CAT5e or more) Ethernet cables.
Multiple EtherCAT Interface Modules can be used.

EtherCAT is an open, high performance ethernet based fieldbus system, which has been integrated into several IEC standards (IEC 61158, IEC 61784 and IEC61800). It is a high performance, deterministic protocol, with high bandwidth usage, low latency and low communication jitter. Various network topologies are supported, including line, tree or star. The EtherCAT compliant servo amplifiers from any number of vendors may be included in a network.

The module supports both the CANopen and servo drive (sercos, IEC 61491) EtherCAT profiles, along with the mailbox transfer protocol to exchange configuration, status and diagnostic information between the master and slave.

RJ45 CONNECTOR



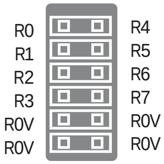
100 base-T Ethernet (EtherCat Master).

TIME BASED REGISTRATION

Time based registration uses a 10MHz clock to record the time of a registration event which is then referenced to time stamps on the axis position from the digital drive network. An accurate registration position is then calculated. The 10MHz clock gives a time resolution of 100nsec. The position and speed of the axis are recorded so that the user can compensate for any fixed delays in the registration circuit.

Any time based registration input can be assigned to any Digital or Virtual axis. This makes the registration very flexible and enables multiple registration channels per axis. Each registration channel can be armed independently and assigned to an axis on the fly.

REGISTRATION CONNECTOR



R0 - R7: registration inputs (24V).

R0V: registration common 0V return.

Registration inputs can be allocated to any axis by software.

LED FUNCTIONS

LED	LED colour	LED function
ok	Green	ON=Module Initialised Okay
0	Red	Quick Flash = Module Error Slow Flash = Not in operational state
1	Yellow	Status 1
2	Yellow	Network Activity

Anybus-CC Module (P875)

Open communications is an important aspect to any control system. This module adds support for the Anybus CompactCom device modules.

Anybus-CC is a plug-in module supporting all major Fieldbus and Ethernet networks. Its innovative design and versatile functionality offers the Anybus-CC optimal flexibility for OEM manufacturers.

The Anybus modules can be found at: www.anybus.com

Anybus CompactCom Module shown for illustration only. Anybus **CC** Modules may be purchased seperately.

Anybus CC Modules support (firmware v2.0263).

- AB6211 **CC**-Link
- AB6201 DeviceNet
- AB6200 Profibus
- AB6216 EtherCAT
- AB6224 Ethernet/**IP** 2 port
- AB663 Modbus **TCP** 2 port
- AB6221 Prifinet-**IO** 2 port

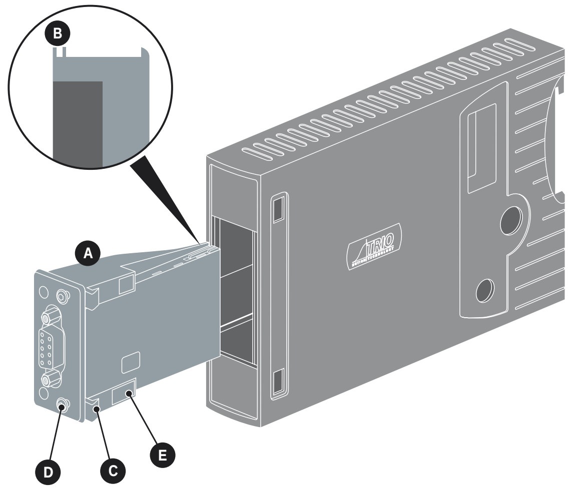
ANYBUS MODULE FITTING

Push the Anybus® module (A) into the Trio Expansion Interface taking care to keep its base in contact with the PCB and align guide slots (B) with the connector rails inside.

Ensure that the moulded hooks (C) on the lower front edge of the Anybus® module locate under the P875 PCB at the front.

When the module is flush with the face of the Trio Expansion Interface, tighten the two “Torx” head screws (D) to locate the two lugs (E) and secure the Anybus® module.

To remove the module, reverse this procedure.



I/O EXPANSION MODULES

4

General Description of I/O Modules

Trio Motion Technology’s range of digital and analogue input/output expansion modules are designed to enable simple and scalable I/O extension for Trio’s *Motion Coordinator*s. In addition to 24V input, output and bi-directional modules, there are relay and analogue I/O modules.

The *Motion Coordinator* I/O expansion system uses CANbus to reduce wiring and allow input/output modules to be distributed remotely. Up to 32 Digital modules and up to 4 Analogue modules may be added to the system.

All CAN Input, Output and I/O modules are DIN rail mounted with the I/O connections located conveniently on the front face. They have been designed with a spaced-saving footprint only 26mm wide so allowing large amounts of Digital and Analogue I/O to be packed in an area no bigger than the average PLC. Address selection is simply done by setting DIP switches that are neatly located under the pull-up flap. LEDs show the I/O state and indicate an error code for straight forward system commissioning and de-bugging.

To install CAN modules, see "Installing the CAN I/O Modules" on page 5-10.

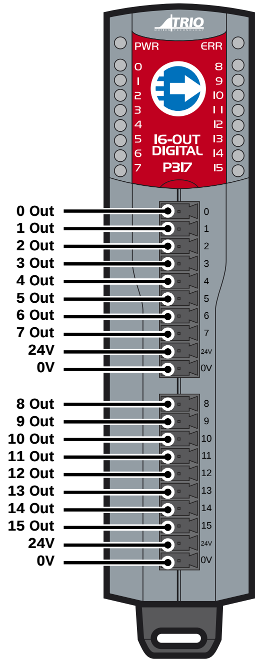
CANbus is used for communication and control between the *Motion Coordinator* and the CAN I/O modules. CANbus is a tried and tested, well known industrial data link which is reliable, noise immune and flexible. All CAN I/O modules are compatible with any *Motion Coordinator* that has a CANbus port and they support various CAN protocols.

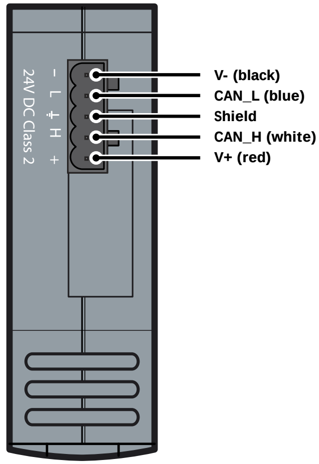
PRODUCT CODE:

CAN 16-Output Module	P317
CAN 16-Input Module	P318
CAN 16-I/O Module	P319
CAN Analogue I/O Module	P326
CAN 8-Relay Module	P327

CAN 16-Output Module (P317)

The Trio CAN 16 Output module offers a compact DIN rail mounted relay input expansion capability for all Trio *Motion Coordinators*. Using remote I/O on the Trio CANbus can significantly reduce the machine wiring.





Up to 16 output modules may be connected to the CAN network which may be up to 100m long. This provides up to 256 distributed output channels at 24Vdc level. All outputs are short-circuit proof and completely isolated from the CANbus. P317 modules may be mixed on the same bus, with other types of Trio CAN I/O modules on the same network to build the I/O configuration required for the system.

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

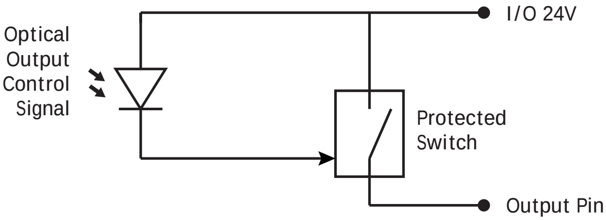
CANBUS

The CANbus port has over voltage and reverse polarity protection. Various protocols can selected using the configuration switches.

24V OUTPUT CHANNELS

The P317 has two banks of eight outputs, both banks of outputs are electrically isolated and require their own 24V and 0V. Output channels have a protected 24V sourcing output connected to the output pin. The output circuit has electronic over-current protection and thermal protection which shuts the output down when the current exceeds 250mA.

Care should be taken to ensure that the 250mA limit for the output circuit is not exceeded, and that the total load for the group of 8 outputs does not exceed 1 amp.



With no load, the outputs may 'float' up to 24V even when off. Fit a load resistor, for example 10k, when bench testing the P317.

LED INDICATORS

The green power (PWR) LED and red error (ERR) LED display the status of the CAN I/O module. The actual status displayed will depend on the protocol selected.

The status LEDs marked 0 - 15 represent the output channels 0 - 15 of the module. The actual outputs as seen by the *Motion Coordinator* software will depend on the modules' address.

CONFIGURATION SWITCHES

The switches are hidden under the display window. These can be adjusted to set the module address, protocol and data rate.

SPECIFICATION P317

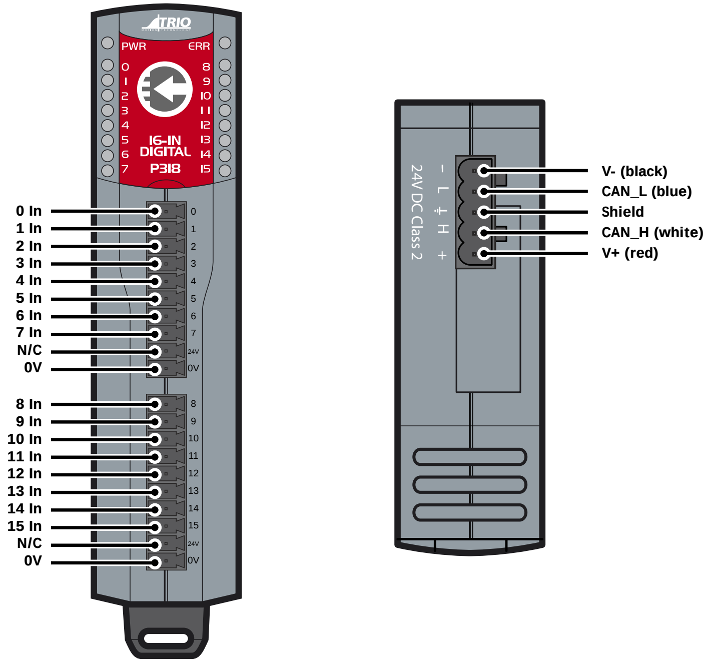
Outputs:	16 24 Volt output channels with 2500V isolation
Configuration:	16 output channels
Output Capacity:	1A per bank of 250mA / channel
Protection:	Outputs are overcurrent and over temperature protected
Indicators:	Individual status LED's
Address Setting:	Via DIP switches
Power Supply:	24V dc, Class 2 transformer or power source 18 ... 29V dc / 1.5W.
Mounting:	DIN rail mount
Size:	26mm wide 85mm deep 130mm high
Weight:	128g
CAN:	500kHz, Up to 256 expansion I/O channels
EMC:	EN 61000-6-2 : 2005 Industrial Noise Immunity / EN 61000-6-4 : 2007 Industrial Noise
CAN protocol:	Trio CAN I/O or CANopen DS401.

CAN 16-Input Module (P318)

The Trio CAN 16 Input module offers a compact DIN rail mounted relay input expansion capability for all Trio *Motion Coordinators*. Using remote I/O on the Trio CANbus can significantly reduce the machine wiring.

Up to 16 input modules may be connected to the CAN network which may be up to 100m long. This provides up to 256 distributed input channels at 24Vdc level. All input points are high level (24V in = ON) and completely isolated from the CANbus. P318 modules may be mixed on the same bus, with other types of Trio CAN I/O modules on the same network to build the I/O configuration required for the system.

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

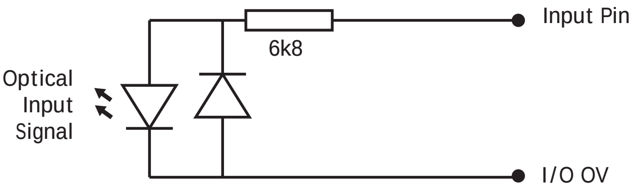


CANBUS

The CANbus port has over voltage and reverse polarity protection. Various protocols can selected using the configuration switches.

24V INPUT CHANNELS

The P318 has two banks of eight inputs, both banks of outputs are electrically isolated and have independent 0V. Input channels are opto-isolated 24V, which are designed to be ON when the input voltage is greater than 18 Volts and OFF when the signal voltage is below 2V. The input has a 6k8 resistor in series and so provides a load of approximately 3.5mA at 24V.



LED INDICATORS

The green power (PWR) LED and red error (ERR) LED display the status of the CAN I/O module. The actual status displayed will depend on the protocol selected.

The status LEDs marked 0 - 15 represent the input channels 0 - 15 of the module. The actual input as seen by the *Motion Coordinator* software will depend on the modules' address.

CONFIGURATION SWITCHES

The switches are hidden under the display window. These can be adjusted to set the module address, protocol and data rate.

SPECIFICATION P318

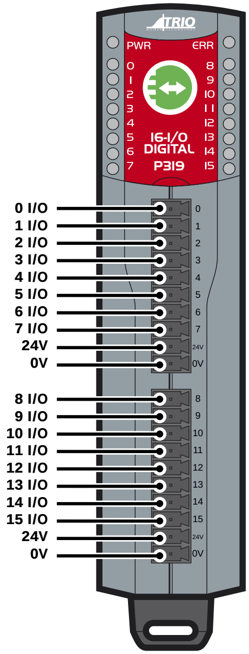
Inputs:	16 24 Volt input channels with 2500V isolation
Configuration:	16 input channels
Protection:	Inputs are reverse polarity protected
Indicators:	Individual status LED’s
Address Setting:	Via DIP switches
Power Supply:	24V dc, Class 2 transformer or power source 18 ... 29V dc / 1.5W.
Mounting:	DIN rail mount
Size:	26mm wide 85mm deep 130mm high
Weight:	128g
CAN:	500kHz, Up to 256 expansion I/O channels
EMC:	EN 61000-6-2 : 2005 Industrial Noise Immunity / EN 61000-6-4 : 2007 Industrial Noise Emissions
CAN protocol:	Trio CAN I/O or CANopen DS401.

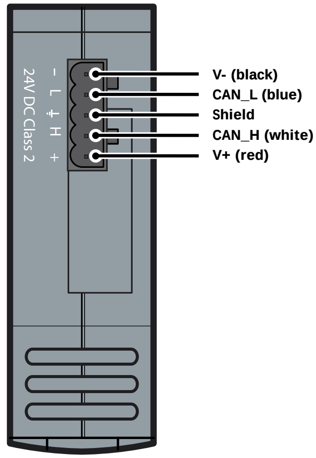
CAN 16-I/O Module (P319)

The Trio CAN 16 Input/ Output module offers a compact DIN rail mounted relay input expansion capability for all Trio *Motion Coordinators*. Using remote I/O on the Trio CANbus can significantly reduce the machine wiring.

Up to 16 I/O modules may be connected to the CAN network which may be up to 100m long. This provides up to 256 distributed bi-directional input/output channels at 24Vdc level. All input points are high level (24V in = ON) all outputs are short-circuit proof and the I/O is completely isolated from the CANbus. P319 modules may be mixed on the same bus, with other types of Trio CAN I/O modules on the same network to build the I/O configuration required for the system.

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





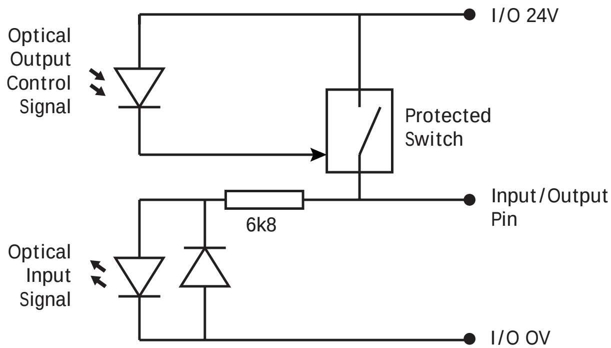
CANBUS

The CANbus port has over voltage and reverse polarity protection. Various protocols can selected using the configuration switches.

24V INPUT/ OUTPUT CHANNELS

The P319 has two banks of eight bi-directional input/ outputs, both banks are electrically isolated and require their own 24V and 0V. Input/output channels are bi-directional, so can be used as an input or output. Bi-directional inputs have a protected 24V sourcing output connected to the same pin. If the output is unused, the pin may be used as an input in the program. The output circuit has electronic over-current protection and thermal protection which shuts the output down when the current exceeds 250mA.

Care should be taken to ensure that the 250mA limit for the output circuit is not exceeded, and that the total load for the group of 8 outputs does not exceed 1 amp.



LED INDICATORS

The green power (PWR) LED and red error (ERR) LED display the status of the CAN I/O module. The actual status displayed will depend on the protocol selected.

The status LEDs marked 0 - 15 represent the I/O channels 0 - 15 of the module. The actual I/O as seen by the *Motion Coordinator* software will depend on the modules' address.

CONFIGURATION SWITCHES

The switches are hidden under the display window. These can be adjusted to set the module address, protocol and data rate.

SPECIFICATION P319

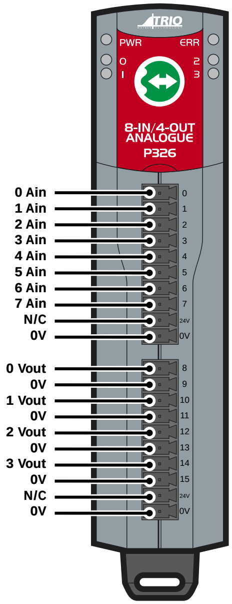
Inputs:	16 24 Volt input channels with 2500V isolation
Outputs:	16 24 Volt output channels with 2500V isolation
Configuration:	16 input/output channels
Output Capacity:	Outputs are rated at 250mA/channel. (1 Amp total/bank of 8 I/O's)
Protection:	Outputs are overcurrent and over temperature protected
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:	26mm wide 85mm deep 130mm high
Weight:	128g
CAN:	500kHz, Up to 256 expansion I/O channels
EMC:	EN 61000-6-2 : 2005 Industrial Noise Immunity / EN 61000-6-4: 2007 Industrial Noise
CAN protocol:	Trio CAN I/O or CANopen DS401.

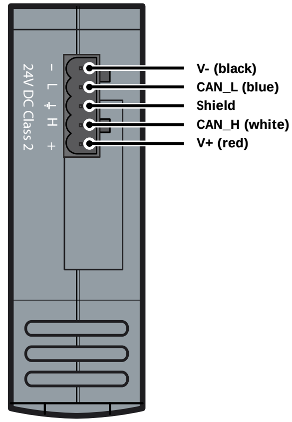
CAN Analogue I/O Module (P326)

The Trio CAN Analogue I/O module offers a compact DIN rail mounted relay output expansion capability for all Trio *Motion Coordinator*s. Using remote I/O on the Trio CANbus can significantly reduce the machine wiring.

Up to 4 analogue modules may be connected to the CAN network which may be up to 100m long. This provides up to 32 distributed analogue inputs and 16 analogue outputs . Each module provides 8 channels of 12-bit analogue inputs (+/-10v) and 4 channels of 12-bit (+/-10v) analogue outputs. All analogue I/O are completely isolated from the CANbus. P326 modules may be mixed on the same bus, with other types of Trio CAN I/O modules on the same network to build the I/O configuration required for the system.

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



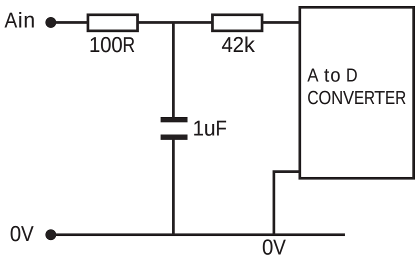


CANBUS

The CANbus port has over voltage and reverse polarity protection. Various protocols can selected using the configuration switches.

INPUT TERMINALS

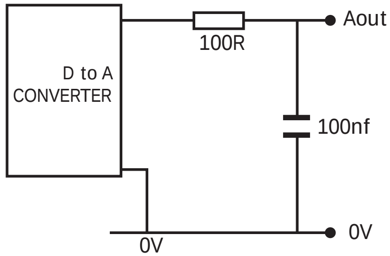
The 8 analogue inputs are single-ended and have a common 0V. Analogue input nominal impedance = 42k Ohm.



OUTPUT TERMINALS

The 4 analogue outputs are single-ended and have a common 0V. Analogue output nominal impedance = 100 Ohm.

The recommended minimum load resistance on the output is 2k Ohm.



LED INDICATORS

The green power (PWR) LED and red error (ERR) LED display the status of the CAN I/O module. The actual status displayed will depend on the protocol selected.

The status LEDs marked 0 - 3 are only used to display an error.

CONFIGURATION SWITCHES

The switches are hidden under the display window. These can be adjusted to set the module address, protocol and data rate.

SPECIFICATION P326

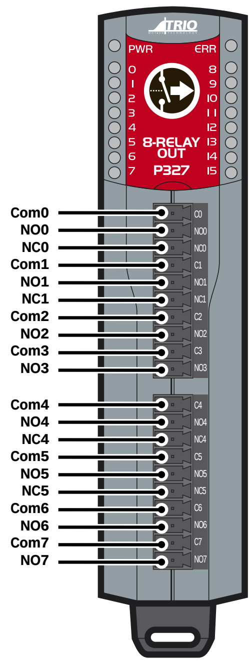
Analogue Inputs:	8 +/-10 Volt inputs with 500V isolation from CAN bus.
Resolution:	12 bit.
Protection:	Inputs are protected against 24V over voltage.
Analogue Outputs:	4 +/-10 Volt outputs with 500V isolation from CAN bus.
Resolution:	12Bit.
Address Setting:	Via DIP switches.
Power Supply:	24V dc, Class 2 transformer or power source. 18 ... 29V dc / 1.5W.
Mounting:	DIN rail mount.
Size:	26mm wide 85mm deep 130mm high.
Weight:	128g
CAN:	500kHz, Up to 32 analogue input channels and 16 analogue output channels.
EMC:	EN 61000-6-2 : 2005 Industrial Noise Immunity / EN 61000-6-4 : 2007 Industrial Noise Emissions.
CAN Protocol:	Trio CAN I/O or CANopen DS401.

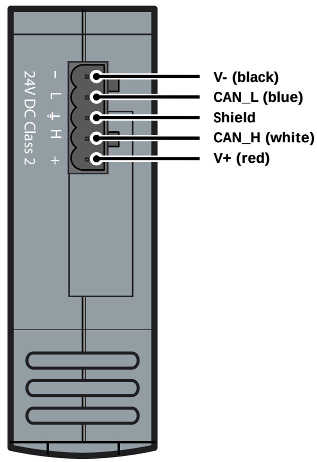
CAN 8-Relay Module (P327)

The Trio CAN 8 Relay module offers a compact DIN rail mounted relay output expansion capability for all Trio *Motion Coordinators*. Using remote I/O on the Trio CANbus can significantly reduce the machine wiring.

Up to 16 relay modules may be connected to the CAN network which may be up to 100m long. This provides up to 128 distributed low power relay channels at up to 30Vdc or 49Vac. Four of the 8 channels in each module are change-over contact and the remaining four are normally-open contacts. All output points are voltage-free contacts and are completely isolated from the CANbus. P327 modules may be mixed on the same bus, with other types of Trio CAN I/O modules on the same network to build the I/O configuration required for the system.

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





Do not connect 24V and 0V to the bottom two pins (Com3, NO3 and Com7, NO7) on the connectors as the pin connections are different to the details molded into the plastic case.

CANBUS

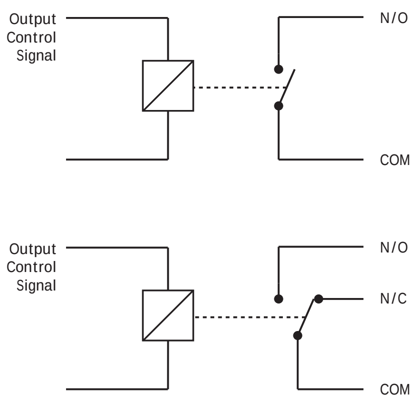
The CANbus port has over voltage and reverse polarity protection. Various protocols can selected using the configuration switches.

RELAY CHANNELS

Each relay channel is an independent isolated voltage free set of contacts. Channels 0, 1, 4 and 5 are change-over contacts and channels 2, 3, 6 and 7 are normally open contacts only. Each contact is rated at 30Vdc (24 Watts) or 49Vac (62.5 VA). Absolute maximum current for any one contact is 1A under all conditions.

Relay contacts do not have built-in suppression so external EMC suppression components must be fitted as required.

Using the Relay in a circuit where the Relay will be ON continuously for long periods (without switching) can lead to unstable contacts, because the heat generated by the coil itself will affect the insulation, causing a film to develop on the contact surfaces. Be sure to use a fail-safe circuit design that provides protection against contact failure or coil burnout.



LED INDICATORS

The green power (PWR) LED and red error (ERR) LED display the status of the CAN I/O module. The actual status displayed will depend on the protocol selected.

The status LEDs marked 0 - 7 represent the relay channels 0 - 7 of the module. The actual output as seen by the *Motion Coordinator* software will depend on the modules' address.

CONFIGURATION SWITCHES

The switches are hidden under the display window. These can be adjusted to set the module address, protocol and data rate.

SPECIFICATION P327

Outputs:	8 relays 30Vdc / 49Vac
Configuration:	4 NO relays and 4 change-over relays
Output Capacity:	Maximum switching power per contact: 62.5 VA, 24W (dc) Max current 1 Amp.
Protection:	Outputs to CAN circuit isolation, 1,500V dc.
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:	26mm wide 85mm deep 130mm high

Weight:	174g
CAN:	500kHz, Up to 128 expansion relay channels
EMC:	EN 61000-6-2: 2005 Industrial Noise Immunity / EN 61000-6-4: 2007 Industrial Noise
CAN protocol:	Trio CAN I/O or CANopen DS401

Controller I/O mapping

DIGITAL I/O ORDER

The controller has different sources of I/O which it has to map to IN and OP. This includes I/O from built in I/O, module I/O and CAN I/O. All of these sources are mapped in blocks of 8, some modules have more than 8 I/O so will take up multiple blocks. Any modules using less than 8 will consume a block of 8 and the remainder of the block will be virtual I/O.

By default built in controller I/O is mapped first followed by module I/O then CAN I/O. **MODULE _ IO _ MODE** is used to configure a different order or to disable the module I/O. When mapping the blocks of separate input and outputs the controller will overlap any inputs and outputs. Please note that bi-directional I/O cannot be split so can cause gaps in the I/O map.

All supported CAN protocols are mapped into the CAN section. For example a system with a MC464, FlexAxis 8, 1 CAN input and 1 CAN output module would be mapped as follows.

I/O source	Inputs	Outputs	I/O
MC464 I/O	0-7		8-15
FlexAxis 8	16-19		20-23
CAN address 0	24-40	24-40	

The FlexAxis is mapped to one block of I/O, as only 4 pins are bi-directional, outputs 16-19 are now virtual.

A different system using a MC464, EtherCAT, 1 CAN input and 1 CAN output module would be mapped as follows.

I/O source	Inputs	Outputs	I/O
MC464 I/O	0-7		8-15
Ethercat	16-23		
CAN address 0	24-40	16-23	

You can see that the EtherCAT inputs and CAN Output module are mapped to the same numbers. It is important to remember that the IN and OP are separate unless they are combined in a bi-directional I/O point.

ANALOGUE I/O ORDER

Up to 32 CAN analogue inputs can be added to the system these are mapped to AIN in order of the module address. Analogue inputs are mapped as follows:

AIN	Source
0 to 31	CAN analogue inputs
32-33	Built in analogue inputs
33+	Module analogue inputs

Analogue outputs are mapped to AOUT in order of the module address starting at 0.

TrioCANv2 Protocol

GENERAL DESCRIPTION

The MC4xx range controllers by default will use TrioCANv2 protocol, this has various enhancements of previous versions of TrioCAN. The protocol allows for a combination of current and older CAN I/O modules though not all features of TrioCANv2 will be available if a P325, P315 or P316 module is used.

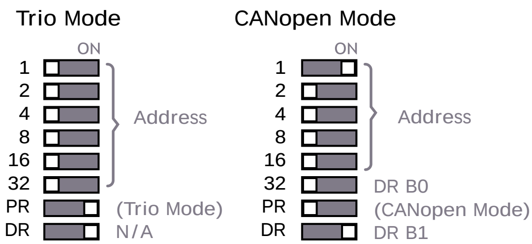
Enhancements to the protocol allow for the following:

- Increase the number of CAN inputs to 256
- Increase the maximum number of CAN outputs to 256
- Increase the total sum of Inputs and Outputs to 512 (bi-directional I/O counts as 1 input and 1 output)
- Allow new analogue output functionality
- Recognise digital input modules
- Recognise digital output modules
- Allow up to 32 digital modules by overlapping input and output addresses.
- Allow expansion module registration inputs and hardware PSWITCH outputs to be used as I/O
- Improved error handling – any error on the network is reported to the controller

If you need to revert to TrioCANv1 protocol you can set **CANIO _ MODE** and **MODULE _ IO _ MODE**. When using **CANIO _ MODE=1** all digital input, output and relay modules are treated as bi-directional I/O modules.

These changes will impact how you address the CAN I/O modules and how the I/O is mapped into the controller.

PROTOCOL SELECTION



TrioCAN (all versions) can be selected on the CAN I/O modules using the protocol (PR) switch. When the controller initialises the CAN network it will tell the module to either use TrioCANv1 or TrioCANv2. It is recommended to leave the controller using TrioCANv2 however TrioCANv1 can be manually set in the controller using **CANIO _ MODE.**

The data rate is fixed to 500kHz for TrioCANv2 Protocol, the data rate (DR) switch has no function.
It is not possible to mix the CAN I/O modules which are running the TrioCANv2 protocol with DeviceNet equipment or CANopen devices on the same network

CONTROLLER SETUP

All *Motion Coordinator*s are configured by default to look for a TrioCAN network, MC4xx range controllers will automatically use TrioCANv2 if the modules on the network all support it. To force the controller to always use TrioCANv1 you can set **CANIO _ MODE.**

To automatically search the CAN bus for TrioCAN modules on power up, **CANIO _ ADDRESS** must be set to 32. There is no need to set this as it is the default value.

There are various system parameters available on the controller to check and change the status of the TrioCAN network, these include **CANIO _ STATUS, CANIO _ ADDRESS, CANIO _ ENABLE and CANIO _ MODE.**

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

- Inputs 0 - 63 ONLY are available for use with system parameters which specify an input, such as **FWD _ IN**, **REV _ IN**, **DATUM _ IN** etc.
- The built-in I/O channels have the fastest operation <1mS
- CAN input modules with addresses 0-3 have the next fastest operation up to 2mS
- The remaining CAN input modules operate up to 20mS
- Outputs are set on demand.

UPDATE RATES

DIGITAL I/O

The digital I/O are cascaded through the modules, this means that lower address modules have a higher update rate.

Function	Update rate
Inputs address 0-3	2ms, no more than 50ms when state unchanged
Inputs address 4-11	10ms, no more than 50ms when state unchanged
Inputs address 12-15	20ms, no more than 50ms when state unchanged
Output address 0-3	5ms or on change of state
Output address 4-7	6ms
Output address 8-11	6ms (offset by 2ms from outputs address 4-7)
Output address 12-15	6ms (offset by 4ms from outputs address 4-7)

ANALOGUE I/O

Analogue inputs have a standard operation which is enabled by default. Some applications require higher speed updates for example when using the analogue inputs as feedback into a servo loop.

Function	Update rate
Analogue Inputs, standard mode	10ms
Analogue Inputs fast mode	2ms
Analogue outputs	On state change

Standard operation is selected by default by the analogue module on power up. Fast operation has to be selected by executing the following **BASIC** in a configuration or startup program:

```
CAN(-1, 5, 4, $50, 8 ,1)
CAN(-1, 7, 4, $04, module_address, $00, $20, $00, $00, $00, $01)
```

DIGITAL CAN I/O ADDRESSING

To enable up to 32 modules on the TrioCANv2 network and up to 512 I/O points Inputs and Outputs are addressed separately. There are 16 addresses (0-15) available for input modules and 16 addresses (0-15) available for outputs. Bi-directional modules take the same address from both the input and output range. There must be no gaps in the input address range, but gaps are allowed in the output address range.

Relay modules are addressed as per digital outputs, they use a block of 16 outputs even though they only have 8.

The total number of digital outputs, digital inputs and total digital I/O are reported by the system parameters NIN, NOP, NIO. The digital configuration is also reported in the startup message.

It is important to remember that **IN** and **OP** are only connected if you are using a bi-directional module. When using Input and Output modules with the same address **IN**(x) and **OP**(x) can be physically different I/O. If you need to read the state of an output you should use **READ _ OP(x)**.

For example a system with 5 CAN 16-Input, 2 16-IO, 7 16-Output and one Relay module could be mapped as per the table below. The CAN I/O start at 16 as the controller has 16 I/O built-in and no module I/O. The start position will move depending on the number of built in I/O and module I/O.

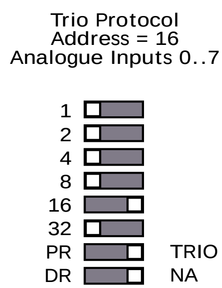
I/O source	Inputs	Outputs	Relay	I/O
Controller I/O	0-7			8-15
CAN address 0				16-31
CAN address 1				32-47
CAN address 2	48-63	48-63		
CAN address 3	64-79	64-79		
CAN address 4	80-95	80-95		
CAN address 5	96-111	96-111		
CAN address 6	112-127	112-127		
CAN address 7			128-135 (136-143 virtual)	
CAN address 8		144-159		
CAN address 9		160-175		

You can see from this chart how the input and output modules are allowed to have overlapping addresses. Bi-directional I/O modules must have a unique address. The relay module only has 8 outputs but uses 1 bank of 16 outputs.

TrioCAN (v1) treats all modules as bi-directional I/O and so every module must have a unique address. The total number of I/O points is limited to 256 and the network is limited to 15 modules.

ANALOGUE I/O ADDRESSING

The address switches on the analogue I/O modules will affect the order in which the I/O is mapped into AIN and **AOUT**. The first analogue module should be address 16 the second to 17 etc, there should be no gaps in the analogue I/O addressing. The addresses are set as a binary sum so for address 17 both switch 16 and 1 must be ON.



The total number of analogue outputs, analogue inputs and total analogue I/O are reported by the system parameters **NAOUT**, **NAIN**, **NAIO**. The analogue configuration is also reported in the startup message.

The analogue I/O are addressed as per the following table.

Address	AIN	AOUT
16	0-7	0-3
17	8-15	4-7
18	16-23	8-11
19	24-31	12-15

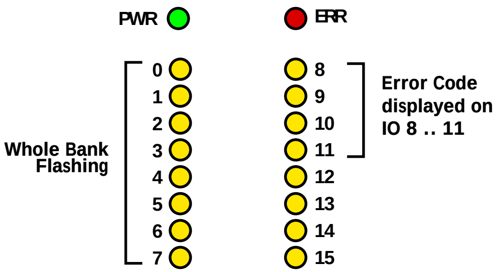
ERROR CODES

When there is a problem with the TrioCANv2 network an error code is displayed on the LED's. All CAN I/O modules have a power LED (PWR) and an error LED (ERR). The power led should be illuminated while the 24V is applied to the CAN connector and the error LED will turn ON when there is an error. The actual error can be read from the status LED's

You can detect which modules have errors by reading **CANIO _ STATUS** in the motion *coordinator*

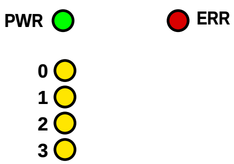
DIGITAL INPUT, OUTPUT AND I/O MODULES

When there is an error the left bank of LED's will flash and the ERR LED will be ON. The error code will be displayed as a binary number on LED's 8-11

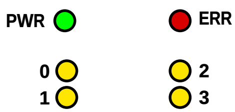


Relay module and Analogue I/O module

When there is an error the error code will be displayed as a binary number on LED's 0-3 and the ERR LED will be ON.



P327



P326

ERROR CODES

Once the binary number has been read from the CAN I/O LED's then the error is as per the table below. Please note that only the error LED's are shown.

Code	P317, P318, P319, P327 LEDs	P326 LEDs	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

TROUBLESHOOTING

If the network configuration is incorrect 2 indications will be seen: The CAN module will indicate an error and the *Motion Coordinator* will report the wrong number of digital or analogue I/O.

If the error is 'uninitialised' then please check:

- Terminating 120 Ohm Network Resistors fitted?
- 24Volt Power to Network?
- Are the addresses correct?

- Have you power cycled the I/O modules after setting the address?
- Cable used is the correct CAN bus specification?
- Is **CANIO _ ADDRESS**=32?

If the network is OK but you are having I/O problems please check:

- 24Volt Power to each I/O bank required?
- You are using the correct I/O in the controller?
- **MODULE _ IO _ MODE** is set as you expect?
- **CANIO _ MODE** is set as you expect?

If the network stops during use please check:

- Terminating 120 Ohm Network Resistors fitted?
- The CAN cable is shielded with the shields correctly connected to earth

Cable used is the correct CAN bus specification?

- Connectors/ wires are not loose

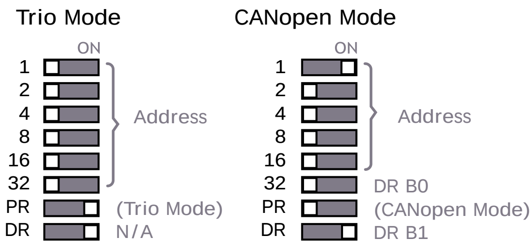
CANopen DS401

GENERAL DESCRIPTION

The CAN modules can support CANopen DS401 so that they can be used with another manufacturers master or with a Trio *Motion Coordinator* and another manufacturer's module on the network.

PROTOCOL SELECTION

CANopen is selected on the CAN I/O modules using the protocol (PR) switch on the module.



CANopen allows the use of different data rates, this is selected by setting the switches marked 32 and DR. Switch 32 sets bit 0 of the data rate and DR sets bit 1.

(DR B1)	DR (DR B0)	Data Rate
0	0	115K
0	1	250KB

(DR B1)	DR (DR B0)	Data Rate
1	1	500KB
1	1	1Mb

CONTROLLER SETUP

To use CANopen DS402 an initialisation program must be run that configures the network. Examples of this program can be found on the Trio website. Once The CANopen network is configured then you can use the CAN I/O with the standard **IN, OP, READ _ OP, AIN** and **AOUT** *commands as normal.

(*Future software release)

MODULE ADDRESSING

Each CAN I/O module becomes a node on the CANopen network. The address switches are used to assign a unique node number to the module.

ERROR CODES

The power (PWR) and error (ERR) LEDs display the modules current state as per the tables below.

LED STATE DEFINITIONS

LED state	Description
LED on	The LED constantly on.
LED off	The LED constantly off.
LED flickering	The LED flashes on and off with a frequency of approximately 10 Hz.
LED blinking	The LED flashes on and off with a frequency of approximately 2.5Hz: on for approximately 200ms followed by off for approximately 200ms.
LED single flash	The LED indicates one short flash.
LED double flash	The LED indicates a sequence of two short flashes.
LED triple flash	The LED indicate a sequence of three short flashes.
LED quadruple flash	The LED indicates a sequence of four short flashes.

PWR LED ERROR CODE

The PWR LED is used as the ‘CANopen run LED’ as recommended by CANopen. Its state displays the following:

CAN Run LED	State	Description
Flickering	AutoBitrate/LSS	The auto-bitrate detection is in progress or LSS services are in progress (alternately flickering with error LED)
Blinking	PRE-OPERATIONAL	The device is in state PRE-OPERATIONAL
Single flash	STOPPED	The device is in state STOPPED
Double flash	Reserved for further use	
Triple flash	Program/ Firmware download	A software download is running on the device
On	OPERATIONAL	The device is in state OPERATIONAL

ERR LED ERROR CODE

The ERR LED is used as the 'CANopen error LED' as recommended by CANopen. Its state displays the following:

ERR LED	State	Description
Off	No error	The device is in working condition
Flickering	AutoBitrate/LSS	The auto-bitrate detection is in progress or LSS services are in progress (alternately flickering with run LED)
Blinking	Invalid Configuration	General configuration error
Single flash	Warning limit reached	At least one of the error counters of the CAN controller has reached or exceeded the warning level (too many error frames)
Double flash	Error control event	A guard event (NMT-slave or NMTmaster) or a heartbeat event (heartbeat consumer) has occurred
Triple flash	Sync error	The sync message has not been received within the configured communication cycle period time out.
Quadruple flash	Event-timer error	An expected PDO has not been received before the event-timer elapsed
On	Bus off	The CAN controller is bus off

5 INSTALLATION

Installing Hardware

Installing the MC664 / MC464

PACKAGING

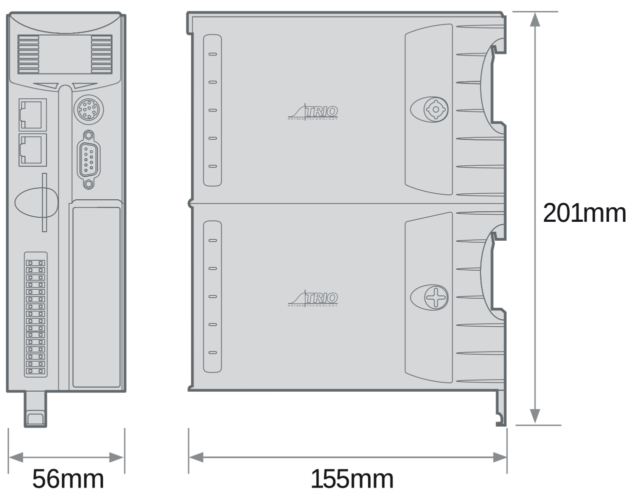
The *Motion Coordinator* MC664 / MC464 is designed to be mounted on a DIN rail or, by use of optional mounting clips, it can be screwed to a backplate.

A cast metal chassis provides mechanical stability and a reliable earth connection to aid EMC immunity.

The rugged plastic case includes ventilation holes, top and bottom, and a removable cover to access the memory battery.

EXPANDABLE DESIGN

System expansion is done by adding either single or double height modules. These are clipped to the MC664 / MC464 and secured by a bolt which also acts as the earth connection between the MC664 / MC464 and the module.



MC664 / MC464 Dimensions

ITEMS SUPPLIED WITH THE MC664 / MC464

CONNECTORS:

- 9 way D-Type plug
- Quick connect I/O connector (30 way)

PANEL MOUNTING SET:

- 2 x Mounting bracket
- 1 x M3 x 10mm Countersunk screw
- 1 x M3 x 6mm Countersunk screw
- Quick start guide

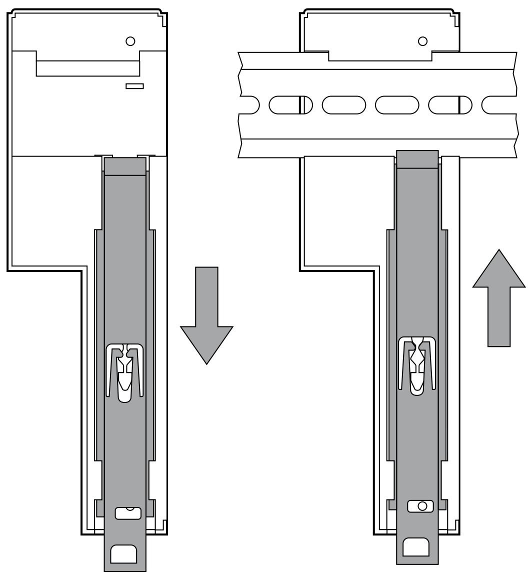
MOUNTING MC664 / MC464

GENERAL

The MC664 / MC464 must be mounted vertically and should not be subjected to mechanical loading. Care must be taken to ensure that there is a free flow of air vertically around the MC664 / MC464.

DIN RAIL

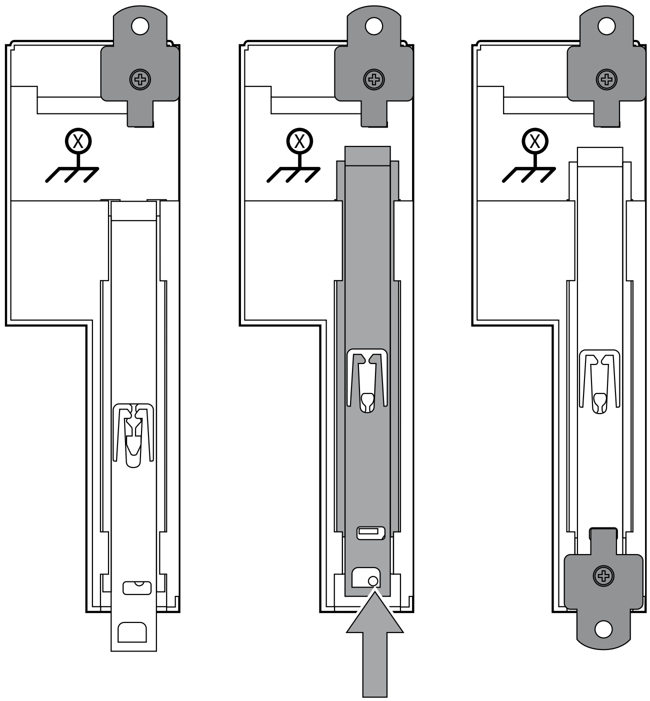
Pull down the clip to allow the MC664 / MC464 to be mounted on a single DIN rail. Push up the clip to lock it to the rail.



Mounting Clips

Remove the 2 mounting clips from their packaging and insert one at the top rear of the case, by fitting the small tab into the rectangular slot and fix with the M3 x 6mm screw provided.

The second clip fits to the bottom of the case rear. Line up the DIN rail lever with the hole and slot in the metal chassis, fit the clip into the slot and fix it with the M3 x 10mm screw.



ENVIRONMENTAL CONSIDERATIONS

The MC664 / MC464 should not be handled whilst the 24 Volt power is connected.

Ensure that the area around the ventilation holes at the top and bottom of the MC664 / MC464 and any additional modules are kept clear. Avoid violent shocks to, of vibration of, the MC664 / MC464, system and modules whilst in use or storage.

IP RATING: IP 20

The MC664 / MC464 and add-on modules are protected against solid objects intruding into the case and against humidity levels that do not induce condensation to occur.

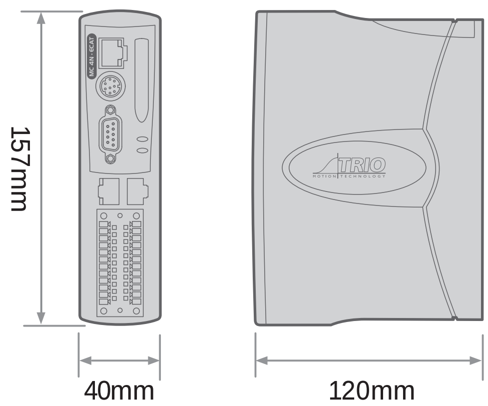
Installing the MC4N

PACKAGING

The *Motion Coordinator* MC4N is designed to be mounted using the 2 mounting holes located on the back-plate.

A cast metal chassis provides superb mechanical stability and a dedicated earth connection point to aid EMC immunity.

The rugged plastic case has conveniently placed access ports for the I/O, encoder inputs, pulse outputs, EtherCAT port, Ethernet and serial connections. A slot is provided for the optional Micro SD card.



ITEMS SUPPLIED WITH THE MC4N

CONNECTORS

- 1 x 9 way D-Type plug and shell
- 1 x 5 way quick dis-connect screw terminal block
- 2 x 12 way quick dis-connect screw I/O connector
- Quick start guide

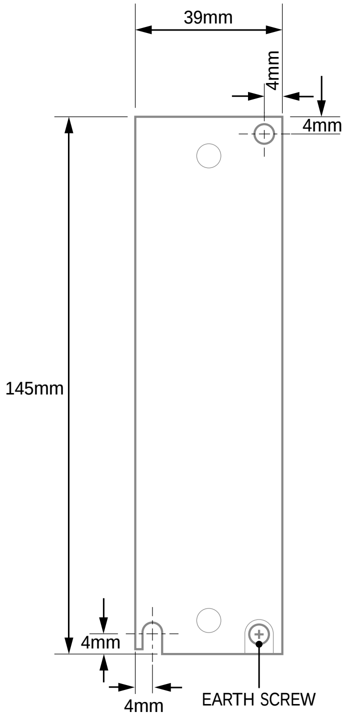
MOUNTING MC4N

GENERAL

The MC4N must be mounted vertically and should not be subjected to mechanical loading. Care must be taken to ensure that there is a free flow of air vertically around the MCN.

SCREW MOUNTING

Drill and tap 2 mounting holes using the dimensions shown below. Use 2 x M4 pan-head screws, (not supplied) of a suitable length, to fix the MC4N to the panel. Screw the lower screw into the panel, leaving the screw head between 4 and 6 mm above the panel surface. Slide the MC4N down on to the screw and insert the upper screw. Tighten both screws.



ENVIRONMENTAL CONSIDERATIONS

The MC4N should not be handled whilst the 24 Volt power is connected.

Ensure that the area around the top and bottom of the MC4N and any additional I/O modules is kept clear. Avoid violent shocks to, of vibration of, the MC4N system and modules whilst in use or storage.

IP RATING: IP 20

The MC4N is protected against solid objects intruding into the case and against humidity levels that do not induce condensation to occur.

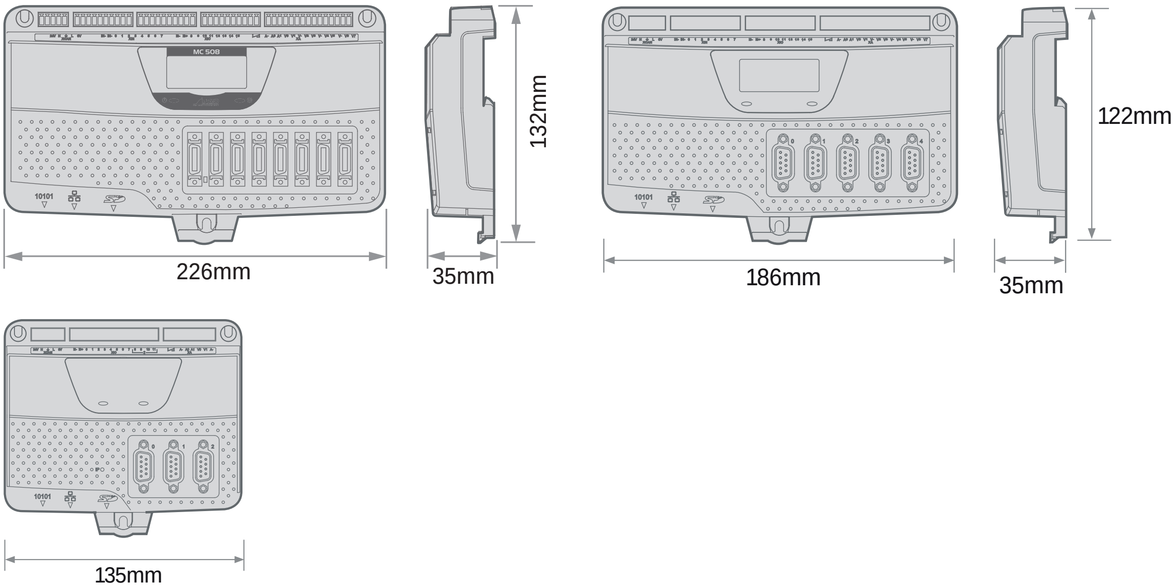
Instaling the MC508 / MC405 / MC403

PACKAGING

The *Motion Coordinator* MC508 / MC405 / MC403 is designed to be mounted on a DIN rail or optionally, using the 3 mounting holes, it can be screwed to a back-plate.

A cast metal chassis provides superb mechanical stability and a dedicated earth connection point to aid EMC immunity.

The rugged plastic case has conveniently placed access ports for the I/O, encoder inputs, pulse outputs,



Ethernet and serial connections. A slot is provided for the optional Micro SD card.

ITEMS SUPPLIED WITH THE MC508 / MC405 / MC403

CONNECTORS

- 3 or 5 x 9 way D-Type plug and shell (MC405 / MC403)
- 2 x MDR type connectors to flying lead cables (MC508)
- 1 x 5 way quick dis-connect screw terminal block
- 8 way and 14 way quick dis-connect screw terminal block (MC403)
- 3 x 10 way and 1 x 16 way quick dis-connect screw terminal block (MC405)
- Quick start guide

MOUNTING MC508 / MC405 / MC403

GENERAL

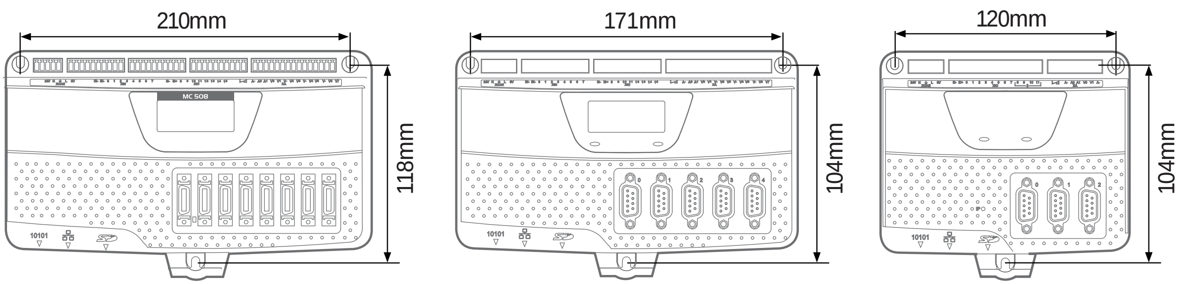
The MC508 / MC405 / MC403 must be mounted vertically and should not be subjected to mechanical loading. Care must be taken to ensure that there is a free flow of air vertically around the MC508 / MC405 / MC403.

DIN RAIL

Pull down the clip to allow the MC508 / MC405 / MC403 to be mounted on a single DIN rail. Release the spring-loaded clip to lock it to the rail.

SCREW MOUNTING

Drill and tap 3 mounting holes using the dimensions shown below. Use 3 x M4 pan-head screws, (not supplied) of a suitable length, to fix the MC508 / MC405 / MC403 to the panel. Screw the upper 2 screws into the panel, leaving the screw head between 4 and 6 mm above the panel surface. Slide the MC508 / MC405 / MC403 up on to the 2 screws and insert the remaining lower screw. Tighten all 3 screws.



ENVIRONMENTAL CONSIDERATIONS

The MC508 / MC405 / MC403should not be handled whilst the 24 Volt power is connected.

Ensure that the area around the top and bottom of the MC508 / MC405 / MC403 and any additional I/O modules is kept clear. Avoid violent shocks to, of vibration of, the Mc508 / MC405 / MC403, system and modules whilst in use or storage.

IP RATING: IP 20

The MC508 / MC405 / MC403 are protected against solid objects intruding into the case and against humidity levels that do not induce condensation to occur.

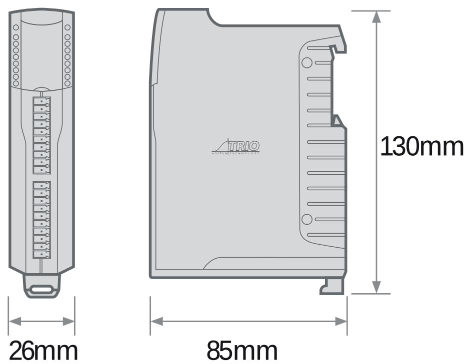
Installing the CAN I/O Modules

PACKAGING

The CAN I/O modules are designed to be mounted on a DIN rail.

The rugged plastic case includes ventilation holes, top and bottom.

The dimensions are shown below.



CAN Module Dimensions

ITEMS SUPPLIED WITH CAN I/O MODULES

- 5 way CAN connector
- 2x 10 way I/O connectors
- 2x 120 Ohm terminating resistors
- Quick start guide

MOUNTING CAN I/O MODULES

The CAN I/O modules should be mounted vertically and should not be subjected to mechanical loading. Care must be taken to ensure that there is a free flow of air vertically around the CAN I/O module.

To mount pull down the sprung loaded clip, slot over the DIN rail and release the clip to lock the module to the rail.

ENVIRONMENTAL CONSIDERATIONS

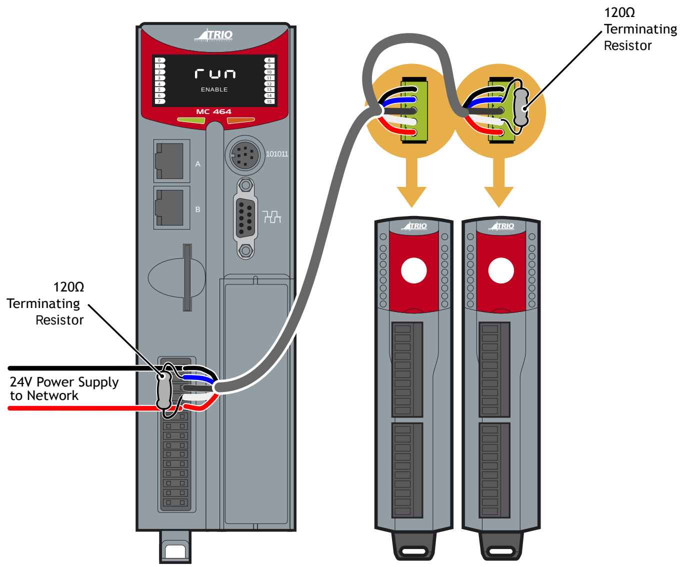
The CAN I/O should not be handled whilst the 24 Volt power is connected.

Ensure that the area around the ventilation holes at the top and bottom of the CAN I/O are kept clear. Avoid violent shocks to, of vibration of, the can i/o modules whilst in use or storage.

IP RATING: IP 20

BUS WIRING

The CAN 16-I/O Modules and the *Motion Coordinator* are connected together on a CAN network 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.

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.

EMC 6

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.

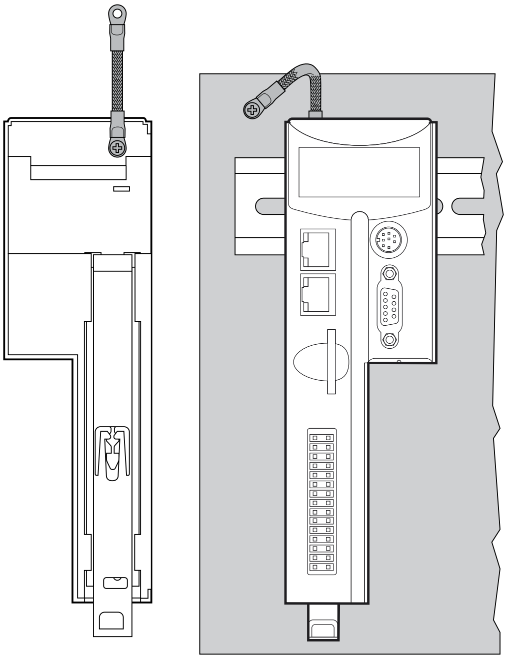
- Mount noise generators such as contactors, solenoid coils and relays as far away as possible from the *Motion Coordinator*.
- Where possible use solid-state contactors and relays.
- Fit suppressors across coils and contacts.
- Route heavy current power and motor cables away from signal and data cables.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

EMC Earth - MC664 / MC464

Best EMC performance is obtained when the MC664 / MC464 is attached to an earthed, unpainted metal panel using the two mounting clips. When screwed directly to the panel, the clips provide the required EMC earth connection.

If the MC664 / MC464 is mounted on a DIN rail, then an additional EMC earth must be attached as shown below. Use a flat braided conductor, minimum width; 4mm. Connect to the earthed metal panel as close to the MC664 / MC464 as possible. Do not use circular cross-section wire. Do not run the conductor to a central star point.

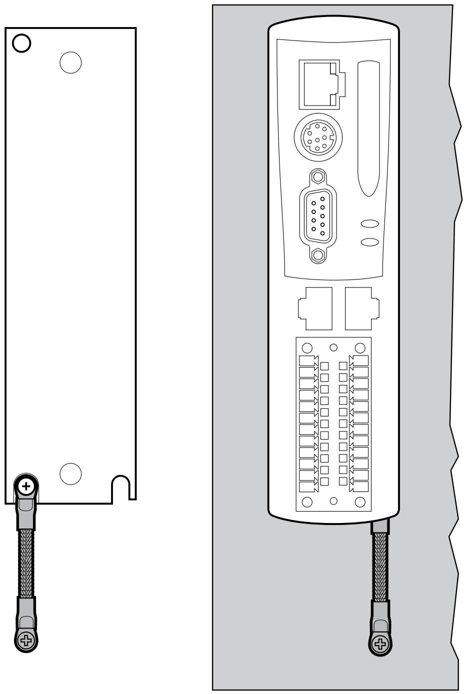


MC664 / MC464 Earth Braid shown rear (left) and front (right)

EMC Earth – MC4N

Best EMC performance is obtained when the MC4N is attached to an earthed, unpainted metal panel using two mounting screws. When screwed directly to the panel, the metal chassis provides the required EMC earth connection.

An additional EMC earth can be attached form the earth screw on the MC4N back plate as shown below. Use a flat braided conductor, minimum width 4mm. Connect to the earthed metal panel as close to the *Motion Coordinator* as possible. Do not use circular cross-section wire. Do not run the conductor to a central star point.

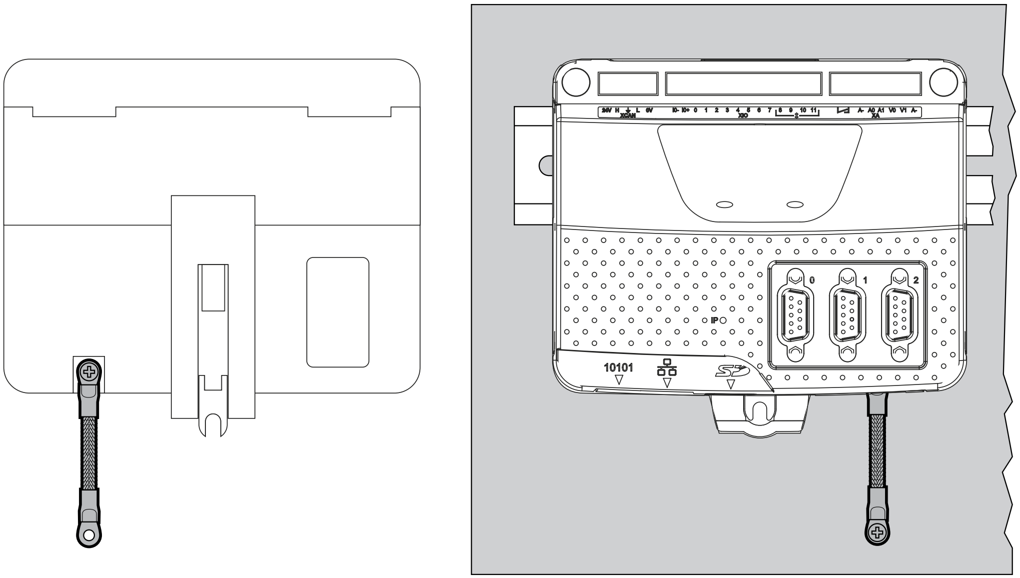


MC4N Earth Braid shown rear (left) and front (right)

EMC Earth – MC508 / MC405 / MC403

Best EMC performance is obtained when the MC508/MC405/MC403 is attached to an earthed, unpainted metal panel using three mounting screws. When screwed directly to the panel, the metal chassis provides the required EMC earth connection.

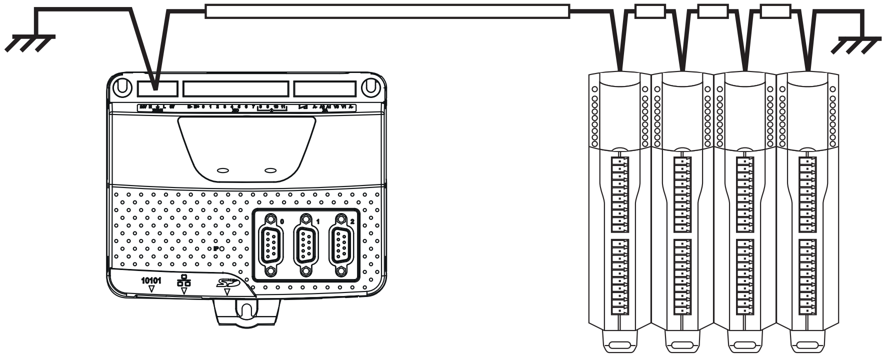
If the MC508/MC405/MC403 is mounted on a DIN rail, then an additional EMC earth must be attached as shown below. Use a flat braided conductor, minimum width 4mm. Connect to the earthed metal panel as close to the *Motion Coordinator* as possible. Do not use circular cross-section wire. Do not run the conductor to a central star point.



MC403 Earth Braid. MC508 / MC405 is Similar

EMC Earth - CAN I/O Modules

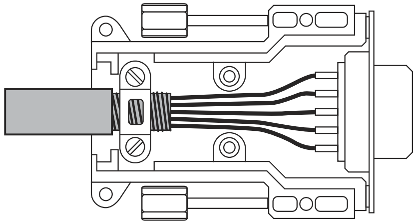
Best EMC performance is obtained when the CAN I/O modules have the screen of the CAN cable connected to the shield pin of the 5 way connector. Both ends of the CAN cable must be connected to an earth point on the back panel of the cabinet. The connection must be as close as possible to the last I/O module. Use a flat braided conductor, minimum width 4mm. Do not use circular cross-section wire. Do not run the conductor to a central star point.



MC403 and CAN I/O Modules

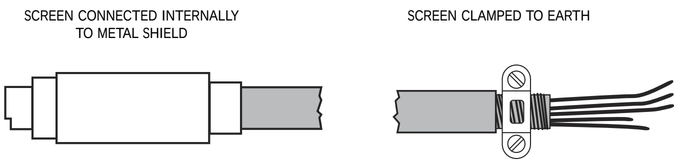
Cable Shields

All encoder cables must be terminated in the correct D-type plug, either 9 way or 15 way as required. For best EMC performance use a metal or metalised plastic cover for the D-type connector. Clamp the screen of the encoder cable where it enters the connector cover. Do not make a "pig-tail" connection from the screen to the plug cover. When plugging the D-type into the MC664 / MC464, use the jack-screws to firmly attach the D-type plug to the socket on the *Motion Coordinator*, axis modules or HMI.



Both ends of the encoder cable's screen must be connected using a 360 degree contact and not a pig-tail connection.

The 0V must be connected separately from the screen. Make sure that encoder cables are specified with one extra wire to carry the 0V.



All serial cables must be terminated in an 8-pin mini-DIN connector. For best EMC performance, clamp the screen of the serial cable where it enters the connector cover. Do not make a "pig-tail" connection from the screen to the plug cover.

Both ends of the serial cable's screen must be connected using a 360 degree contact and not a pig-tail connection.
The 0V must be connected separately from the screen. Make sure that serial cables are specified with one extra wire to carry the 0V. This applies to RS422/RS485 serial connections as well as RS232.

Digital Inputs

*Motion Coordinator*s MC403, MC403-Z, MC405, MC4n and MC464 do not require shielded cables on the digital inputs. Wiring must be designed according to industry best proctise.

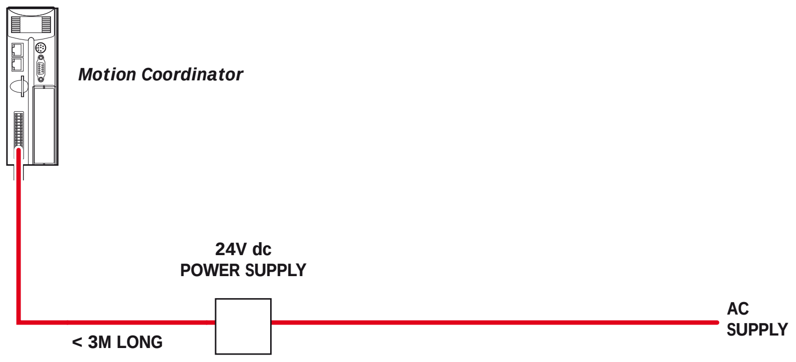
The MC508 and MC664 are fitted with high speed opto-isolated inputs and systems must use shielded cables for all 24V digital inputs to comply with the industry standard.

Surge protection

This section applies to all devices including *Motion Coordinator*s, CAN IO modules and HMIs. The surge protection described is to enable the system components to comply with EMC Generic Immunity for industrial environments standard IEC 61000-6-2:2005.

SINGLE POWER SUPPLY

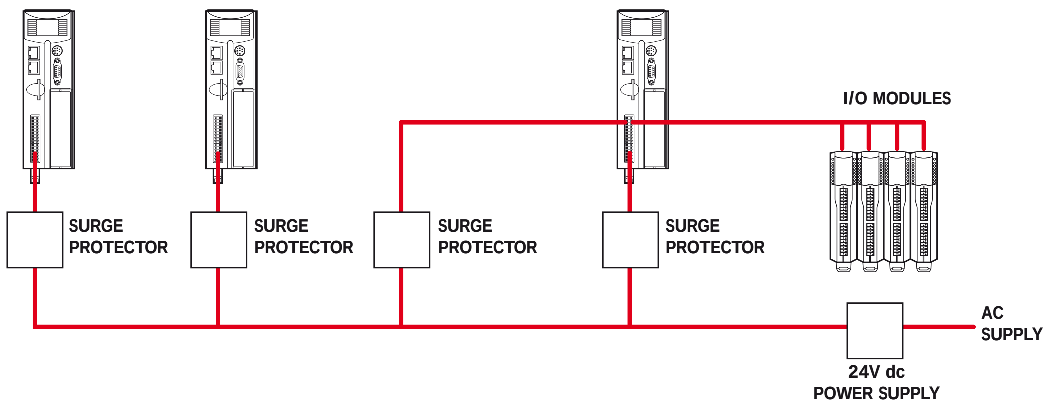
Where the device is supplied with 24V dc from one dedicated 24V power source and the connecting cable is less than 3 metres, there is no need for a separate surge protection device.



Motion Coordinator ***with dedicated power source***

DISTRIBUTED POWER SUPPLY

If the device is connected to a distributed power supply or the cable length between the power source and the device is longer than 3 metres, then a surge protection device must be fitted to comply with the CE EMC directive.



Distributed power supply with surge protection

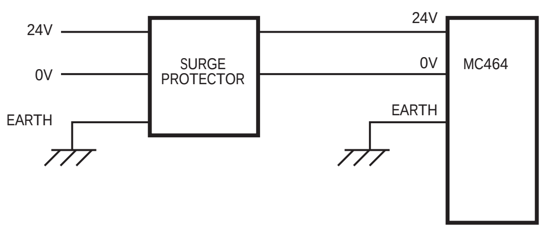
RECOMMENDED PROTECTION DEVICE

If a surge protector is required, a device conforming to the specification below must be installed as close as possible to the 24V power input requiring protection. In addition, the MC508, MC405 and MC403 require 2 x 220 μF electrolytic capacitors to complete the protection circuit.

Protection device - Minimum specification	
Operating Voltage	24V dc
Suppression Begins: Stage Two Stage Three	 30V 35V
Max. Clamp Volts for transients on the line: Stage Two Stage Three	 65V 77V
Surge Current (8/20mSec Pulse) + to - + to Earth - to Earth	 9000A 4000A 4000A
Surge Energy (2mSec Pulse) + to - + to Earth - to Earth	 94 Joules 44 Joules 44 Joules
Response Time	<5 nsec
Resistance to Earth: Max Over-Voltage Operating Voltage	 0.01 Ω > 1 MΩ

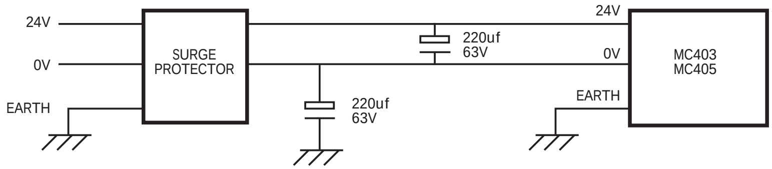
A suggested device is the DC Surge protector TSP-WG6-24VDC-10A-01 from Axiomatic. This protection device is easy to implement with Trio products and is DIN rail mountable. The DC Surge protector and Trio product must be connected to EARTH to make the protection effective.

MC664 / MC464 AND IO DEVICES



Surge protection device

MC403/MC405



Surge protection device

If the I/O power is from a different power source to the main device power, then the I/O power must also have a surge protector fitted.

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:

Equipment must be constructed to ensure that any electromagnetic disturbance it generates allows radio and telecommunications equipment and other apparatus to function as intended.

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 was 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 - EN61000-6-4 +A1: 2007.

The MC4 range of products conform to the Class A limits.

IMMUNITY - EN61000-6-2 : 2005.

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

REQUIREMENTS FOR EMC CONFORMANCE

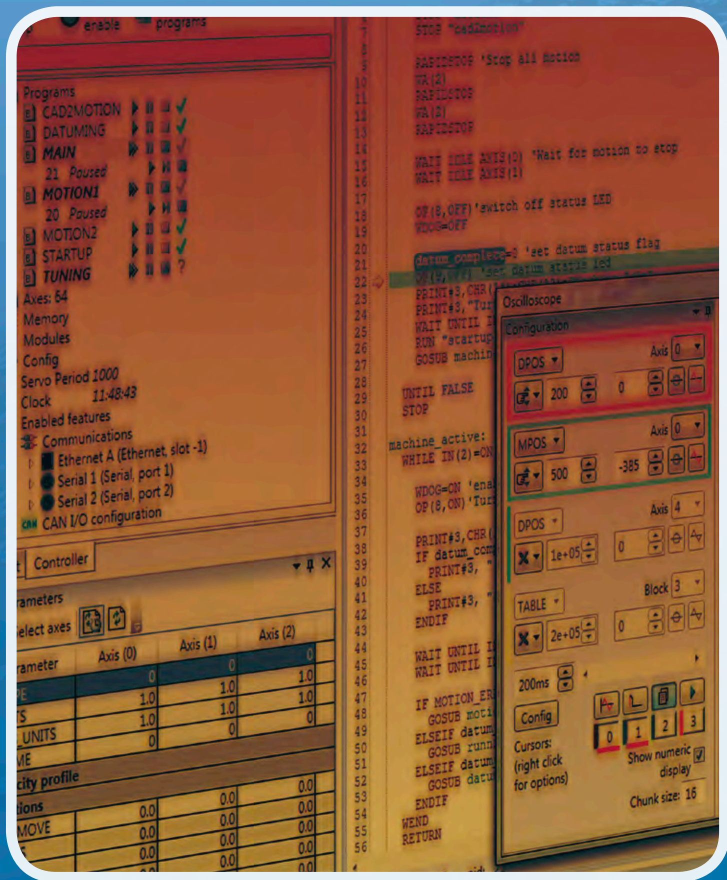
When the Trio products are tested they are wired in a typical system configuration. The wiring practices used in this test system must be followed to ensure the Trio products are compliant within the completed system.

A summary of the guidelines follows:

- The MC664 / MC464 modules must be earthed via the main chassis of the MC4 range using the lower panel mounting clip or an earth strap. This must be done even if DIN rail mounted.
- If any I/O 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 I/O port can act as an antenna for noise.
- Screened cables MUST be used for encoder, stepper and registration input feedback signals and for the demand voltage from the controller to the servo amplifier if relevant. The demand voltage wiring must be less than 1m long and preferably as short as possible. The screen must 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.
- Ethernet cables should be shielded and as a minimum, meet the TIA Cat 5e requirements.
- Connection to the serial ports should be made with a fully screened cable.
- As well as following these guidelines, any installation instructions for other products in the system must be observed.

INDEX

Index



Motion Coordinator - 4xx Range

SOFTWARE REFERENCE MANUAL

Version 7.5

Trio Motion Technology

Motion Coordinator 4xx Range

Software Reference Manual

Seventh Edition • 2014

Revision 5

This manual applies to systems based on the *Motion Coordinator* MC4xx range.

UK
Trio Motion Technology Ltd.
Phone: +44 (0)1684 292333
Fax: +44 (0)1684 297929

USA
Trio Motion Technology LLC.
Phone: + 1 724 540 5018
Fax: +1 724 540 5098

CHINA
Trio Shanghai
Tel: +86 21 5879 7659
Fax: +86 21 5879 4289

INDIA
Trio India
Phone: +91 20 681 149 02

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.

This manual uses the following icons for your reference:

Information that relates to safety issues and critical software information

Information to highlight key features or methods.

Useful tips and techinques.

1 INTRODUCTION

Introduction to Programming

MC4XX *MOTION COORDINATOR* SOFTWARE

The MC4xx range makes a huge advance in programming as well as with its leading hardware design. This manual is a complete reference work covering all the main programming methods, the programming software and the use of remote access methods for Microsoft Windows® packages.

The system designer is free to choose the motors, drives and IO components that best suit the application. Interface options are provided for traditional servo, stepper and piezo control together with and expanding range of digital fieldbus connected drives and IO devices. The MC4xx range can support any number of axes between 1 and 64 in a modular, expandable and cost effective way. Precise and fast motion control is run by 64 bit software developed independently by Trio, benefitting from over a quarter of a century of experience on thousands of real machines world-wide.

The choices available to the system designer now extends to the choice of programming software. Motion *Perfect* 3 and the run-time environment in the *Motion Coordinator* firmware support both TrioBASIC and the industry standard IEC61131-3 programming environment. In addition, there is support for text based languages like HPGL and G-Code within the much extended multi-tasking TrioBASIC. For those applications which need a Windows® PC front-end, the well-established TrioPC Motion ActiveX has been extended and improved and is well suited to high speed connection to the *Motion Coordinator* via Ethernet. For more everyday user interface requirements, *Motion* Perfect v3 includes a complete set of visual programming tools for the Trio Uniplay range of integrated HMIs.

Languages

TrioBASIC has been greatly extended for the MC4xx range. It now includes features such as array variables, string handling, text-file handling and user definable system configuration. The combination of string variable types and the ability to load, save and manipulate text files, is a powerful tool which allows the implementation of text based motion languages like G-Code and HPGL. A new program type, called **MC _ CONFIG**, is used to store all the user defined system configuration changes. This allows the *Motion* Perfect project to store the complete configuration as well as application programs and data. A "must have" for project maintainability.

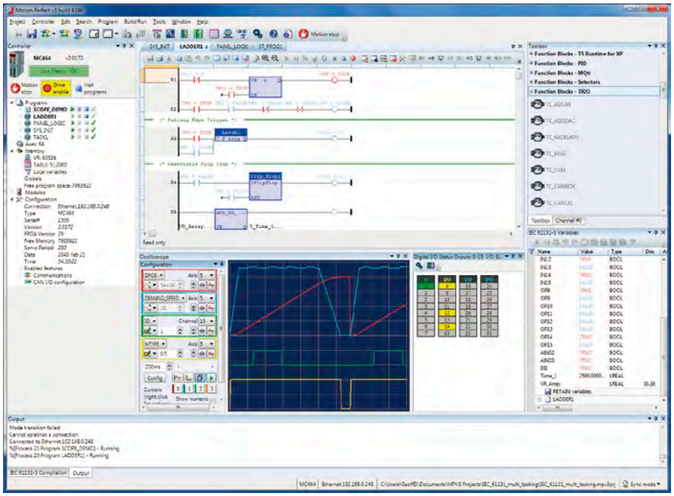
Motion Perfect v3 introduces the option of constructing programs using up to 4 of the IEC61131-3 methods. Ladder (LD), function block (FB), structured text (ST) and sequential function chart (SFC) are all supported through appropriate editor pages and toolbox functions. Only instruction list (IL) is unsupported because its application to motion programming is very limited. All the familiar Trio motion functions are provided as pre-defined function blocks in two special libraries within the MPv3 toolbox.

New to the MC4xx range and *Motion* Perfect v3 is the Uniplay HMI programming system. Create your HMI pages with the MPv3 graphical editor and store them within the *Motion Coordinator* as part of the project. The Uniplay HMI downloads the pages from the *Motion Coordinator* during system startup and interacts with the *Motion Coordinator* during run-time. Uniplay HMI programming does away with the need for a separate programming tool for the HMI. All the machine programming can therefore be stored in one place; the MPv3 project, thus making long term support and software maintenance easier to control.

Setup and Programming

To program the *Motion Coordinator*, a PC is connected via an Ethernet link. The dedicated *Motion* Perfect version 3 Windows® application is normally used to provide a wide range of programming facilities on a PC running Microsoft Windows XP, Vista or Windows 7.

Once connected to the *Motion Coordinator*, the user has direct access to TrioBASIC which 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, synchronised multi axis moves and unsynchronised multi axis moves as well as the control of the digital I/O. Commands for both 2D and 3D interpolated motion are provided as well as transform algorithms for different robot geometries such as SCARA and Delta arrangements.



The MC4xx range of controllers feature a multi-tasking operating system which efficiently allows TrioBASIC and IEC 61131-3 programs to work alongside the motion processing. Multiple TrioBASIC programs plus Ladder Diagram (LD), Function Block (FB), Structured Text (ST) and Sequential Function Chart (SFC) can be constructed and run simultaneously to make programming complex applications much easier.

Motion Perfect version 3 uses the latest .NET technology to provide a more intuitive and familiar user experience. It gives a seamless programming, compilation and debug environment that can work in real-time with the MC4xx range. TrioBASIC support is backwards compatible with *Motion* Perfect 2 projects developed on earlier *Motion Coordinator* platforms. A motion library is provided which enables the familiar Trio Motion Technology commands to be included in IEC 61131-3 programs.

2 TRIOBASIC COMMANDS

Contents

Introduction to TrioBASIC

TrioBASIC is multi-tasking programming language used by the Trio multitasking *Motion Coordinator* range of programmable motion controllers. The syntax is similar to that of other **BASIC** family languages. A PC running the Microsoft Windows™ operating system is used to develop and test the application programs which coordinate all the required motion and machine functions using Trio's *Motion* Perfect software. *Motion* Perfect provides all editing and debugging functionality needed to write and debug applications written in TrioBASIC. The completed application does not require the PC in order to run.

FEATURES

- Fast BASIC language for easy standalone machine programming
- Fully integrated with Trio's *Motion* Perfect application development software
- Comprehensive motion control functions for multiple axes
- Multi-tasking of multiple programs for improved software structure and maintenance
- Support for traditional servo or stepper axes as well as modern digital (**SERCOS**, EtherCAT etc) axes
- A comprehensive set of move types supporting multiple axis coordination as well as simple single axis moves. This includes linear, circular, and spherical interpolation as well as cam profiles and software gearboxes
- Real maths (up to 64 bit) including bit operators and variables
- Support for hardware position capture
- Support for high speed outputs

TrioBASIC has over 300 commands designed to make programming motion functions quick and simple.

How to use this this manual

The TrioBASIC programming reference guide lists all the TrioBASIC keywords used in the MC4xx range of *Motion Coordinators* in alphabetical order. A TrioBASIC keyword can be a simple parameter, or a command with a clearly defined function, such as **FORWARD** or **HALT**, whereas others may take one or more parameters which affect the operation of the command.

This short introduction is intended to provide a guide to using the main programming reference. It identifies the concepts and some words and phrases which have a particular meaning within the context of this manual.

COMMAND REFERENCE ENTRY

Each TrioBASIC keyword is described in the technical reference manual using a standard format. The keyword name is given, what type of TrioBASIC keyword it is, an example of syntax and then a description of its parameters and overall operation. Finally an example of it in a typical program is given when available.

Here is the typical layout.

KEYWORD_NAME

Type:
The keyword type; e.g. **SYSTEM PARAMETER**

Syntax:
The definition of the keyword syntax. Where parameters are optional, they are enclosed in square brackets [].

Description:
A brief description of command or parameter, informing what it does and how it may interact with other parameters or commands.

Parameters:
A table of all the parameters for the command. If the keyword is a parameter itself, then this section will be missed.

Examples:
Example 1:
Where available, at least one example will be shown. When the command is a motion command, the example may be a small sub-set of the sequence needed to show the command working in a realistic application.

See also:
A list of other related keywords so that the reader can easily cross-reference.

KEYWORD TYPES

Keywords are split into groups according to their function, where they may be used and where they are stored in the *Motion Coordinator*. A keyword may have more than one type. For example, a keyword can be a System Variable and be available for use in the **MC_CONFIG** initialisation program.

Below is a table describing all the keyword types.

Axis command	A command sent to a particular axis. An axis command will usually have one or more parameters in parentheses. It will operate on the **BASE** axis that is set, but it can also take the **AXIS** modifier keyword. e.g. **MOVE(100), REGIST(21, 4, 0, 1, 0) AXIS(15)**
Axis Parameter	A parameter which is associated with a particular axis. An axis parameter will operate on the **BASE** axis that is set, but it can also take the **AXIS** modifier keyword. e.g. **P_GAIN = 1.2, x = MPOS AXIS(2)**
Command line only	The command or parameter may be entered in the command line on *Motion* Perfect terminal 0. It may NOT be used within an executable TrioBASIC program.
Constant	The keyword returns a constant value. Used to make common program constants more readable. e.g. **OP(10, ON), WAIT UNTIL MARK = TRUE**

FLASH	The parameter is automatically stored in the flash memory and will therefore be available on the next and all subsequent power ups. Note that parameters stored to Flash from the command line are not referenced in the *Motion* Perfect project and must be documented separately. For this reason, the use of **MC_CONFIG** is recommended even if the parameter is also stored in the Flash.
Mathematical function	The keyword is a typical TrioBASIC mathematical function which can take one or more operands and which returns a result. e.g. **x = COS(y), value = ATAN2(VR(10), VR(11))**
MC_CONFIG	The parameter is available for use in the **MC_CONFIG** script which runs automatically on power up while configuring the system.
Modifier	A modifier keyword is used to modify the target axis, process, port or slot that a command is sent to, or that a parameter is sent to or read from. e.g. **CONNECT(1,3) AXIS(10), x = PROC_STATUS PROC(21), PRINT FPGA_VERSION SLOT(2)**
Process parameter	A parameter which gives the status of a process in the multi-tasking, or which, if written to, has some control function in the multi-tasking. A process parameter operates on process 0 unless the **PROC** modifier is used.
Program Structure	
Slot Parameter	A slot parameter gives some information about the status of the hardware on that slot. Some slot parameters also have a control function when written to. A slot parameter operates on slot 0 unless the **SLOT** modifier is used. e.g. **VR(10) = SERCOS_PHASE SLOT(2), PRINT FPGA_VERSION SLOT(-1)**
System command	A command which operates on the system firmware, or on a part of the *Motion Coordinator* hardware. A system command may have one or more parameters contained within parentheses. **e.g. AUTORUN, SETCOM(19200,8,1,2,2,4)**
System parameter	A parameter which is associated with the system as a whole. A system parameter may control or give the status of something in the operating firmware, or it may be hardware specific. e.g. **NIO, TIME$**

All functions and commands will accept an expression as well as a single variable. For example; a valid expression might be **MOVE(COS(x)*VR(1)/100)**.

KEYWORD SYNTAX

Each entry in the TrioBASIC reference manual shows the syntax of the keyword in a standard form. Syntax, the way you use the keyword, appears in 3 formats in TrioBASIC.

COMMAND

Commands come in 3 types; those which take parameters and those which do not. An example of a command with parameters is shown here.

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

Parameters are contained within parentheses. (round brackets) If there is more than one parameter, then they are separated by a comma. Optional parameters are shown in the syntax description within square brackets. The square brackets are not used when writing the command in a program, so if the optional parameter is used, just insert the comma and the value or expression without square brackets.

Commands which do not have parameters are just entered as the keyword with no parentheses or brackets. For example; **FORWARD**

FUNCTION

Functions can both take a value, or values, and will also return a value. The values given to the function are in parentheses, in the same way as for a command. One or more values may be passed to the function. Mathematical functions are typical of this syntax type;

```
value = COS(expression)
value = ABS(expression)
```

PARAMETER

A parameter carries a value and therefore works in the same way as a variable. A value can be assigned to a parameter or a value can be read from a parameter. Some parameters are read only. This will be shown in the keyword type information.

Some examples of parameter syntax are;

```
P_GAIN = 1.0
VR(10) = PROC_STATUS PROC(3)
IF MPOS AXIS(10) > (ENDMOVE AXIS(10) – 200) THEN
CANIO_ADDRESS = 40
```

CONSTANT

Some keywords are provided to make common constants available to the programmer. These are, of course, read-only. Constants, for the purpose of syntax, can be thought of as a sub-set of the parameter type. Some examples are;

```
    circumference = PI * diameter
    IF result = FALSE THEN
    WHILE TRUE
    OP(30,OFF)
    bit3 = ON
```

VARIABLES

Variables that may be used in expressions or as parameters within a command or function can be stored in volatile RAM, in non-volatile battery backed RAM or in non-volatile Flash memory. A variable may also be local or global.

Local variable

A local variable is given a user defined name. The name can contain letters, numbers and the underscore "_" character. It can be of any length, but only the first 32 characters are used to identify the unique variable name. The value of a local variable is known only to the process that it was defined in.

Local variables are volatile and will be lost at power down.

e.g. **elapsed_time = -TICKS/1000**

Global variables

Global variables, otherwise known as **VR** variables, are held in non-volatile memory. In the MC464 this is maintained by a lithium battery. In the MC403/MC405, the global variables are stored in the Flash memory. Global variables can be accessed from all processes including the command line in terminal 0.

There are a fixed number of global variables. Each variable is accessed by index number, e.g. **value=VR(123)**. See the relevant hardware manual for the highest index number.

e.g. **batch_size = VR(101)**

TABLE values

Another range of globally accessible values is the **TABLE** memory. This is a large indexed array of variables which has a special purpose in some commands. It can also be used as a general memory for application programs.

Table memory may be either volatile or non-volatile. See the appropriate hardware manual for details.

e.g. **TABLE(100, 1.2, 2.3, 4.5, 6.8, 9.0, 15.4, 23.7)**

VARIABLE SYNTAX

The default data type of all variables is double precision float. However, the floating point data type can also store integers up to 52 bits plus sign. Therefore all variables and most parameters can be referenced as if they are integers, without any need to create a separate integer data type definition.

```
my_variable = 450.023 ‘ decimal float
my_variable = 450 ‘ decimal integer
my_variable = $FF6A ‘ hexadecimal integer
my_variable.5 = 1 ‘ sets bit 5 to 1
```

Versions of firmware released after the middle of 2012 have more advanced data types available. For example the String type can be defined by the use of the DIM statement. See under DIM in the Trio **BASIC** reference manual for further information.

LABELS

A label is a place marker in the program. Labels are given user defined names. The name can contain letters, numbers and the underscore "_" character. It can be of any length, but only the first 32 characters are used to identify the unique variable name. The label position is defined by putting the colon ":" character after the label name. The line containing the label can then be referenced within a **GOTO** or **GOSUB** command.

```
start_of_program:

  raduis1 = 123
  GOSUB calc_circle_radius
```

```
  PRINT #5,area1
  WA(500)
GOTO start_of_program

calc_circle_area:
  area1 = PI * radius1 ^ 2
RETURN
```

EXAMPLES

Each keyword entry shows one or more example of how to use the keyword in a realistic context. Sophisticated commands, like the main motion commands, will show a reasonably complete example with all the other associated commands which are required to make the core of a typical application.

More complete programming solutions can be found in Trio's wide range of application notes and programming guides.

ABS

TYPE:
Mathematical function

SYNTAX:
`value = ABS(expression)`

DESCRIPTION:
The ABS function converts a negative number into its positive equal. Positive numbers are unaltered.

PARAMETERS:

Expression:	Any valid TrioBASIC expression

EXAMPLE:
Check to see if the value from analogue input is outside of the range -100 to 100.

```
IF ABS(AIN(0))>100 THEN
  PRINT "Analogue Input Outside +/-100"
ENDIF
```

ACC

TYPE:
Axis command

SYNTAX:
`ACC(rate)`

DESCRIPTION:
Sets both the acceleration and deceleration rate simultaneously.

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

PARAMETERS:

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

EXAMPLES:

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

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

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

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

ACCEL

TYPE:
Axis parameter

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:
Set the acceleration rate and print it to the terminal

```
ACCEL=130
PRINT “ Acceleration rate= “;ACCEL;”mm/sec/sec”
```

ACOS

TYPE:
Mathematical 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 TrioBASIC expression returning a value between -1 and 1.

EXAMPLE:

Print the arc-cosine of -1 on the command line

```
>>PRINT ACOS(-1)
3.1416
>>
```

+ Add

TYPE:

Mathematical operator

SYNTAX:

```
<expression1> + <expression2>
```

DESCRIPTION:

Adds two expressions

PARAMETERS:

Expression1:	Any valid TrioBASIC expression
Expression2:	Any valid TrioBASIC expression

EXAMPLE:

Add 10 onto the expression in the parentheses and store in a local variable. Therefore ‘result’ holds the value 28.9

```
result=10+(2.1*9)
```

ADD_DAC

TYPE:

Axis Command

SYNTAX:

ADD_DAC(axis)

DESCRIPTION:

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

The **ADD_DAC** command is provided to allow a secondary encoder to be used on a servo axis to implement dual feedback control.

This would typically be used in applications such as a roll-feed where a secondary encoder to compensate for slippage is required.

PARAMETERS:

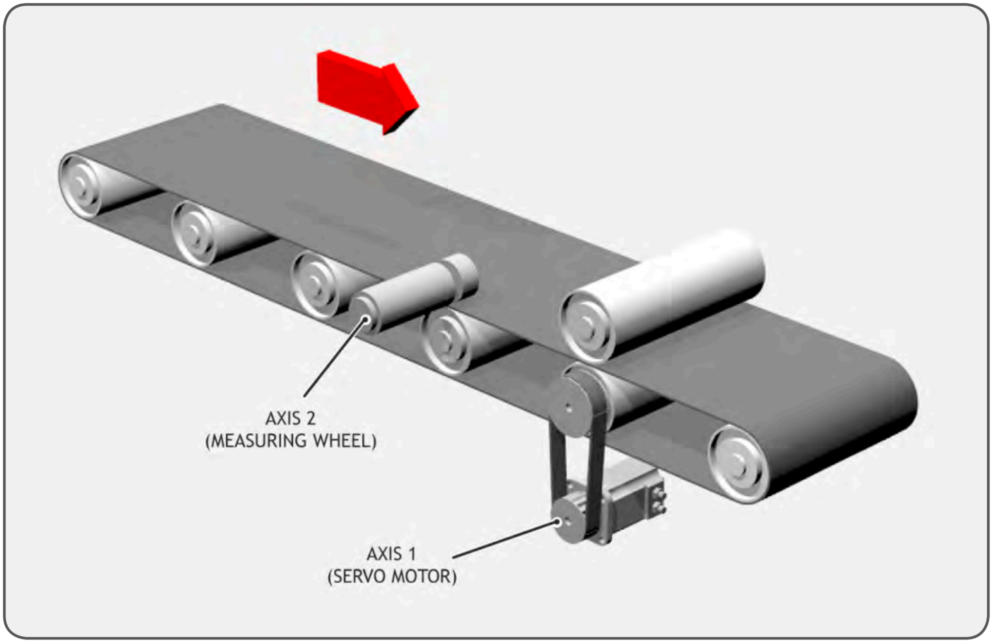
axis:	Number of the second axis, who's output will be added to the base axis. -1 will terminate the **ADD_DAC** link.

EXAMPLE:

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

It is necessary for the two axes to be controlled by a common demand position. Typically this would be achieved by using **ADDAX** to produce a matching **DPOS** on **BOTH** axes. The servo gains are then set up on **BOTH** axes, and the output summed on to one physical output using **ADD_DAC**.

If the required demand positions on both axes are not identical due to a difference in resolution between the 2 feedback devices, **ENCODER_RATIO** can be used on one axis to produce matching **UNITS**.




```
BASE(1)
ATYPE = 44
‘ No need to scale the servo encoder as it is the highest resolution
ENCODER_RATIO(1,1)

‘ Link to the output of the encoders virtual DAC
ADD_DAC(2)
UNITS = 10000

‘ Disable the output from the servo control block by setting PGAIN = 0
P_GAIN = 0
SERVO = ON

BASE(2)
‘ ATYPE must be set to a servo ATYPE to enable the closed position loop
ATYPE = 44

‘ Set the encoder ratio so that it has the same counts per rev as the
servo
ENCODER_RATIO(10000,4096)
```

```
‘ Superimpose axis 1 demand on axis 2
ADDAX(1)
UNITS = 10000

‘ Use servo control block from encoder axis by setting >0 P_GAIN
P_GAIN = 0.5
SERVO = ON

WDOG=ON

BASE(1)
‘ Start movements
MOVE(1200)
WAIT IDLE
```

ADDAX

TYPE:

Axis command

SYNTAX:

ADDAX(axis)

DESCRIPTION:

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

The **ADDAX** command takes the demand position changes from the specified axis and adds them to any movements running on the base axis.

After the **ADDAX** command has been issued the link between the two axes remains until broken and any further moves on the specified axis will be added to the base axis.

The specified axis can be any axis and does not have to physically exist in the system

The **ADDAX** command therefore allows an axis to perform the moves specified on TWO axes added together.

When using an encoder with **SERVO=OFF** the **MPOS** is copied into the **DPOS**. This allows **ADDAX** to be used to sum encoder inputs.

PARAMETER:

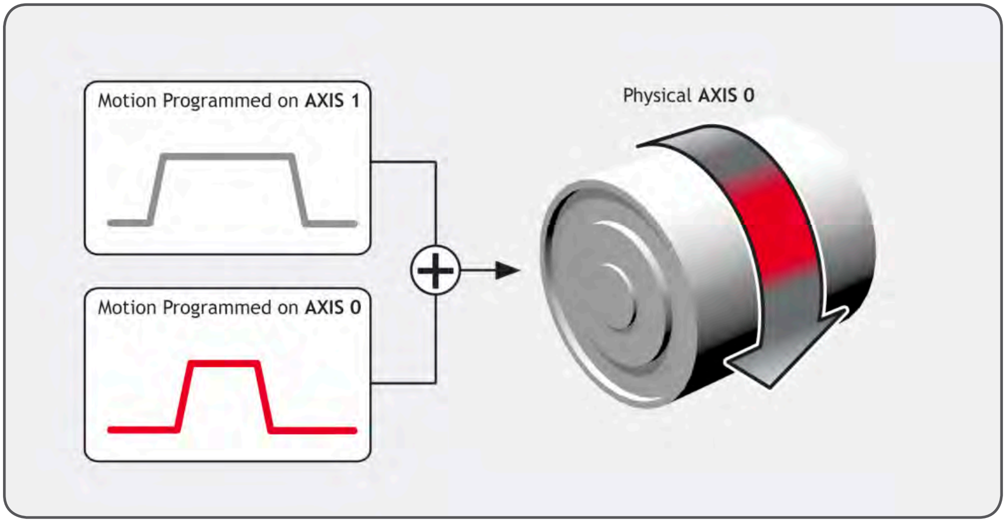
axis:	Axis to superimpose. -1 breaks the link with the other axis.

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

EXAMPLES:

EXAMPLE 1:

Using **ADDAX** on axis with different **UNITS**, Axis 0 will move 1*1000+2*20=1040 edges.




```
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)
```

EXAMPLE 2:

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

ADDAX_AXIS

TYPE:
Axis Parameter (Read Only)

DESCRIPTION:
Returns the axis currently linked to with the **ADDAX** command, if none the parameter returns -1.

EXAMPLE:
Check if an **ADDAX** to axis 2 exists as part of a reset sequence, if it does then cancel it.

```
IF ADDAX_AXIS = 2 then
  ADDAX(-1)
ENDIF
```

ADDRESS

TYPE:
System Parameter

DESCRIPTION:
Sets the RS485 or Modbus multi-drop address for the controller.

VALUE:
Node address, should be in the range of 1..32. If it is set to 255 addressing is not used and all 8 characters from the packet are sent through to the user.

EXAMPLE:
Initialise Modbus as node 5

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

AFF_GAIN

TYPE:
Axis Parameter

DESCRIPTION:

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

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

AIN

TYPE:

System Command

SYNTAX:

```
AIN(channel)
```

DESCRIPTION:

Reads a value from an analogue input. Analogue inputs are either built in to the *Motion Coordinator* or available from the CAN Analogue modules.

The value returned is the decimal equivalent of the binary number read from the A to D converter.

> The built in analogue inputs are updated every servo period.

> The CAN analogue inputs are updated every 10msec

PARAMETERS:

channel:	Analogue input channel number 0...35	
	0 to 31	CAN analogue input channel number
	32 to 35	Built in analogue input channel number

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

EXAMPLE:

Material is to be fed off a roll at a constant speed. There is an ultrasonic height sensor that returns 4V when the roll is empty and 0V when the roll is full. A lazy loop is written in the **BASIC** to control the speed of the roll.

```
MOVE(-5000)
```

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

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

AIN0..3 / AINBI0..3

TYPE:

System Parameter

DESCRIPTION:

These system parameters duplicate the AIN() command.

AIN0..3 is used for single sided analogue inputs.

AINBI0..3 is used for bipolar inputs.

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

If no CAN Analogue modules are fitted, AIN0 and AIN1 will read the first two built-in channels.

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 TrioBASIC expressions.

The AND function between two bits is defined as follows:

AND	0	1

0	0	0
1	0	1

PARAMETERS:

expression1:	Any valid TrioBASIC expression
expression2:	Any valid TrioBASIC expression

EXAMPLES:

EXAMPLE 1:

Using AND to compare two logical expressions, if they are both true then set a local variable.

```
IF (IN(6)=ON) AND (DPOS>100) THEN
  tap=ON
ENDIF
```

EXAMPLE 2:

Use AND as a bitwise operator.

```
VR(0)=10 AND (2.1*9)
```

ANYBUS

TYPE:

System Function

SYNTAX:

ANYBUS(function, slot [, parameters…])

DESCRIPTION:

This function allows the user to configure the active Anybus module and set the network to an operation state. Some networks have limitations on data types and size, please refer the Anybus data sheet for details.

Passive modules require no setup and will appear as a communication channel, they can then be used with **PRINT**, **GET** etc. These modules can be configured using the **SETCOM** command.

PARAMETERS:

function:	0	Configure map
	1	Configure module and start protocol
	2	Stop protocol
	3	Read status byte
	4	Auto configure mapping

FUNCTION = 0:

SYNTAX:

```
value = ANYBUS(0,slot [, map, source [, index, type, count, direction
[,endian]]])
```

DESCRIPTION:

Assigns a **VR** or table point to the memory area that is updated over the network. Individual or all maps can be deleted using the first 4 parameters.

The current mapping can be printed to the terminal using the first 2 parameters.

PARAMETERS:

value:	**TRUE** = the command was successful	
	FALSE = the command was unsuccessful	
slot:	Module slot in which the Anybus is fitted	
map:	Map number, use -1 to delete all maps	
source:	Location for data on the MC464	
	-1	delete map
	0	VR
	1	Table
index:	Start position in data source	

type:	The size and type of data that is sent across the bus	
	0	boolean
	1	signed 8 bit integer
	2	signed 16 bit integer
	3	signed 32 bit integer
	4	unsigned 8 bit integer
	5	unsigned 16 bit integer
	6	unsigned 32 bit integer
	7	character
	8	enumeration
	9-15	(reserved)
	16	signed 64 bit integer
	17	unsigned 64 bit integer
	18	floating point/real number
count:	Number of data types mapped	
direction:	Data direction	
	0	data read into the controller
	1	data transmitted from the controller
endian	0	Use default endian from network (default)
	1	Swap endian

FUNCTION = 1:

SYNTAX:

```
value = ANYBUS(1,slot [, address, baud])
```

DESCRIPTION:

Resets the Anybus module, loads the mapping and then sets the network to operational mode using the parameters provided.

PARAMETERS:

value:	**TRUE**	the command was successful
	FALSE	the command was unsuccessful
slot:	Module slot in which the Anybus is fitted	
address:	Module address, node number, MAC id. etc *(not required for Profinet)*	
baud:	Baud rate CC Link - required	
	0	156 kbps
	1	625 kbps
	2	2.5 Mbps
	3	5 Mbps
	4	10 Mbps
	Baud rate Devicenet – optional	
	0	125 kbps
	1	250 kbps
	2	500 kbps
	3	autobaud (default)
	Baud rate Profibus - automatic, not required	

FUNCTION = 2:

SYNTAX:

```
value = ANYBUS(2,slot)
```

DESCRIPTION:

Stops the cyclic data transfer.

PARAMETERS:

value:	**TRUE**	the command was successful
	FALSE	the command was unsuccessful
slot:	Module slot in which the Anybus is fitted	

FUNCTION = 3:

SYNTAX:
value = **ANYBUS**(3,slot)

DESCRIPTION:
Reads the status byte from the Anybus module.

PARAMETERS:

value:	Anybus status byte:		
	Bits 0-2:	Anybus State:	
		0	**SETUP**
		1	**NW_INIT**
		2	**WAIT_PROCESS**
		3	**IDLE**
		4	**PROCESS_ACTIVE**
		5	**ERROR**
		6	(reserved)
		7	**EXCEPTION**
	Bit 3	Supervisory bit:	
		0	Module is not supervised
		1	Module is supervised by another network device
	Bits 4-7	(reserved)	
slot:	Module slot in which the Anybus is fitted		

FUNCTION = 4:

SYNTAX:
```
value = ANYBUS(4,slot [, address], type, inoff, outoff [,endian])
```

DESCRIPTION:
Auto-configure and start the cyclic network. The mapping can still be read using function 0.

This function only works with Profibus and Profinet. Profinet does not require the address parameter.

PARAMETERS:

value:	**TRUE**	the command was successful
	FALSE	the command was unsuccessful
slot:	Module slot in which the Anybus is fitted	
address:	Module address, node number, MAC id. Etc (Profibus only)	
type:	Data type and location	
	0	VR Integer
	1	Table Integer
	2	VR Float
	3	Table Float
inoff:	Offset for inputs	
outoff:	Offset for outputs	
endian	0	**Use default endian from network (default)**
	1	Swap endian

EXAMPLES:

EXAMPLE 1:

Configure Device Net with 2 16-bit integer inputs and 2 16-bit integer outputs. This data is transmitted cyclically using the ‘Polled Connection’ method. Ensure to configure the master identically to the slave otherwise the data will not transmit.

```
device_net:
  slotnum=0 ‘Local variable with module slot number

‘Map data
  map=FALSE
‘Map received data
  map= ANYBUS(0, slotnum, 1, 0, 0, 2, 4, 0) ‘4*16-bit Int Rx
  IF map=TRUE THEN
    ‘Map transmit data
    map= ANYBUS(0, slotnum, 2, 0, 4, 2, 4, 1) ‘4*16-bit Int Tx
  ENDIF

  IF map=FALSE THEN
    PRINT#term, “Mapping failed”
    STOP
  ENDIF
```

```
'Print mapped data to the terminal
  ANYBUS(0,slotnum)

'Start Network
  map= ANYBUS(1, slotnum, 3, 2)  'MAC ID=3, Baud=500k
  IF map=FALSE THEN
    PRINT#term, "Failed to start network"
    STOP
    ELSE
    PRINT#term, "Network Started"
  ENDIF
  RETURN
```

EXAMPLE 2:

Configure CC-Link with 2 stations, both with 16 bits in, 16 bits out, 2 SINT16 in and 2 SINT16 out. Ensure that the master is configured identically and that the handshaking bits are implemented.

```
cc_link:
'Function 0 - Set up mapping
'station 1
  map = ANYBUS(0, slotnum, 0, 0, 0, 0, 16, 0) '16*BOOL Rx
  map = ANYBUS(0, slotnum, 1, 0, 1, 0, 16, 1) '16*BOOL Tx
  map = ANYBUS(0, slotnum, 2, 0, 2, 2, 2, 0)'2*16-bit Int Rx
  map = ANYBUS(0, slotnum, 3, 0, 4, 2, 2, 1) '2*16-bit Int Tx
'station 2
  map = ANYBUS(0, slotnum, 4, 0, 6, 0, 16, 0) '16*BOOL Rx
  map = ANYBUS(0, slotnum, 5, 0, 7, 0, 16, 1) '16*BOOL Tx
  map = ANYBUS(0, slotnum, 6, 0, 8, 2, 2, 0) '2*16-bit Int Rx
  map = ANYBUS(0, slotnum, 7, 0, 10, 2, 2, 1) '2*16-bit Int Tx

  ANYBUS(0,slotnum) 'print mapping to terminal

'Function 1 - Start Protocol
  IF map = FALSE THEN
  map = ANYBUS(1, slotnum, 1, 2)
  ENDIF
```

EXAMPLE 3:

Configure Profibus using the automated mapping.

```
Profibus:
  vrint=0
  tableint=1
  vrfloat=2
  tablefloat=3
```

```
slotnum=0

‘Function 4, read network mapping, configure and start.
map=ANYBUS(4, slotnum, 5, vrint, 100, 200)

IF map=FALSE THEN
  PRINT#term, «Failed to start network»
  STOP
ENDIF
ANYBUS(0,slotnum) ‘print mapping to terminal
```

EXAMPLE 4:

Configure Profinet using the automated mapping.

```
Profinet:
  vrint=0
  tableint=1
  vrfloat=2
  tablefloat=3
  slotnum=0

  ‘Function 4, read network mapping, configure and start.
  map=ANYBUS(4, slotnum, vrint, 100, 200)

  IF map=FALSE THEN
    PRINT#term, «Failed to start network»
    STOP
  ENDIF
```

AOUT

TYPE:

System Command

SYNTAX:

`AOUT(channel)`

DESCRIPTION:

Writes a value to an analogue output. Analogue outputs available from the CAN Analogue module.

The value sent is the decimal equivalent of the binary number to be written to the D to A converter.

PARAMETERS:

channel:	Analogue output channel number 0...15

EXAMPLE:

An output is to be set to the speed input of an open-loop inverter drive. 10V is 1500 rpm and the required speed is 300 rpm.

```
value = 300 * 2048 / 1500
AOUT(1) = value
```

The analogue output voltage is set to 2V.

The voltage is approximate and the output must be calibrated by the user if high accuracy is required.

AOUT0..3

TYPE:

System Parameter

DESCRIPTION:

These system parameters duplicate the **AOUT** command.

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

ASC

TYPE:

String Function

SYNTAX:

```
value = ASC("string")
```

DESCRIPTION:

ASC returns the **ASCII** value of the first character in the provided **STRING** parameter. If the **STRING** is empty then 0 will be returned.

PARAMETERS:

string:	Any valid **STRING**
value:	An integer value

EXAMPLES:

EXAMPLE 1:
Print the **ASCII** value of character ‘A’ contained within a longer **STRING**.

```
>>PRINT ASC(“ABCDEF”)
65
>>
```

EXAMPLE 2:
Print the **ASCII** value of character ‘9’.

```
>> PRINT ASC(“9”)
57
>>
```

SEE ALSO:
PRINT, STRING, CHR

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.

PARAMETERS:

Expression:	Any valid TrioBASIC expression returning a value between -1 and 1.

EXAMPLE:

Print the arc-sine of -1 on the command line

```
>>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 TrioBASIC expression

EXAMPLE:

Print the arc-tangent of -1 on the command line

```
>>PRINT ATAN(1)
0.7854
```

ATAN2

TYPE:

Mathematical Function

SYNTAX:

```
ATAN2(expression1,expression2)
```

DESCRIPTION:

The ATAN2 function returns the arc-tangent of the ratio expression1/expression2. The result in radians is in the range -PI.. +PI

Use **ATAN2** when calculating vectors as it is quicker to execute than **ATAN**(x/y)

PARAMETERS:

Expression1:	Any valid TrioBASIC expression.
Expression2:	Any valid TrioBASIC expression.

EXAMPLE:

Print the arc-tangent of 0 divided by 1 on the command line

```
>>PRINT ATAN2(0,1)
0.0000
```

ATYPE

TYPE:

Axis Parameter (**MC_CONFIG**)

DESCRIPTION:

The **ATYPE** axis parameter indicates the type of axis fitted. By default this will be set to match the hardware, but some modules allow configuration of different operation.

If you are setting an **ATYPE**, this must be done during initialisation through the **MC_CONFIG**.bas program.

When using **ATYPE** in MC_**CONFIG** you must use the **AXIS** modifier, **BASE** is not allowed.

VALUE:

The following **ATYPE**'s are currently active values

Value	Description
0	No axis daughter board fitted/ virtual axis
30	Analogue feedback Servo
43	Pulse and direction output with enable output

Value	Description
44	Incremental encoder Servo with Z input
45	Quadrature encoder output with enable output
46	Tamagawa absolute Servo
47	Endat absolute Servo
48	SSI absolute Servo
50	**RTEX** position
51	**RTEX** speed
52	**RTEX** torque
53	Sercos velocity
54	Sercos position
55	Sercos torque
56	Sercos open
57	Sercos velocity with drive registration
58	Sercos position with drive registration
59	Sercos spare
60	Pulse and direction feedback Servo with Z input
61	SLM
62	PLM
63	Pulse and direction output with Z input
64	Quadrature encoder output with Z input
65	EtherCAT position
66	EtherCAT speed
67	EtherCAT Torque
68	EtherCAT Open Speed
69	EtherCAT Reference Encoder
75	SSI 32 Absolute Slave
76	Incremental encoder with Z input

Value	Description
77	Incremental encoder Servo with enable output
78	Pulse and direction with **VFF_GAIN** and enable output
79	Pulse and direction feedback with Z input
84	Quadrature encoder output with **VFF_GAIN** and enable output
85	Used for monitoring difference between 2 axes with **AXESDIFF**
86	Tamagawa absolute (input only)
87	Endat absolute (input only)
88	SSI absolute (input only)

Which **ATYPE**s are supported is controller and module dependent.

EXAMPLES:

EXAMPLE 1:

Set a stepper on axis 0 and SSI encoder on axis 1. The default for a flexible axis is servo

```
BASE(0)
ATYPE = 43
BASE(1)
binary = 1
gray = 0
'Set the number of bits
ENCODER_BITS = 24
'Set gray or binary code
ENCODER_BITS.6 = gray
ATYPE = 48
```

EXAMPLE 2:

Set a the **ATYPE** so a Sercos axis uses velocity mode with drive registration

```
ATYPE AXIS(12)=57
```

EXAMPLE 3:

Setting the **ATYPE** for the first 4 axis in the **MC_CONFIG** file so that the first two axes are SSI and the rest incremental servo.

```
ATYPE AXIS(0) = 48
ATYPE AXIS(1) = 48
ATYPE AXIS(2) = 44
ATYPE AXIS(2) = 44
```

EXAMPLE 4:

Set a EnDAT encoder on **AXIS**(0).

```
ENCODER_BITS=25+256*12
ATYPE=47
```

EXAMPLE 5:

Set a Tamagawa encoder on **AXIS**(0). Remember you may need to change the **FPGA_PROGRAM** to use the Tamagawa encoder.

```
ATYPE=46
```

AUTO_ETHERCAT

TYPE:

System Parameter (**MC_CONFIG**)

DESCRIPTION:

Controls the action of the system software on power up. If present, the EtherCAT network is initialized automatically on power up or soft reset (EX). If this is not required, then setting **AUTO_ETHERCAT** to OFF will prevent the EtherCAT from being set up and it is then up to the programmer to start the EtherCAT network from a **BASIC** program.

This command should not be used in a TrioBASIC program. You must use it in the special MC_**CONFIG** script which runs automatically on power up. This parameter is **NOT** stored in **FLASH**.

VALUE:

Value	Description
0	EtherCAT network does not initialise on power up.
1	EtherCAT network searches for drives and sets up the system automatically.

EXAMPLE:

Prevent the EtherCAT system from starting on power up.

```
' MC_CONFIG script file
AUTO_ETHERCAT = OFF
```

AUTORUN

TYPE:
System Command

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

This command should not be used in a TrioBASIC program. You can use it in the command line or a TRIO**INIT**.bas in a SD card.

EXAMPLE:
Using a **TRIOINIT**.bas file in a SD card to load and run a new project

```
FILE "LOAD_PROJECT" "ROBOT_ARM"
AUTORUN
```

AXESDIFF

TYPE:
Axis command

SYNTAX:
AXESDIFF(axis1, axis2)

DESCRIPTION:
The **AXESDIFF** command is used to configure the monitoring of 2 axes performed on an axis with **ATYPE**=85. An axis of **ATYPE**=85 will produce an **MPOS** output based on the difference between **MPOS** of 'axis2' subtracted from **MPOS** of 'axis1', a DAC output will also be produced.

The specified axis can be any axis and does not have to physically exist in the system

PARAMETER:

Axis1:	First Axis to monitor. -1 breaks the link with the other axis.
Axis2:	Second Axis to monitor. -1 breaks the link with the other axis.

EXAMPLES:

EXAMPLE 1:

To monitor axes 3 & 7.

```
ATYPE=85
AXESDIFF(3,7)
```

AXIS

TYPE:

Modifier (**MC_CONFIG**)

SYNTAX:

AXIS(expression)

DESCRIPTION:

Assigns ONE command, function or axis parameter operation to a particular axis.

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

PARAMETERS:

Expression:	Any valid TrioBASIC expression. The result of the expression should be a valid integer axis number.

EXAMPLES:

EXAMPLE 1:

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

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

EXAMPLE 2:

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

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

EXAMPLE 3:

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

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

SEE ALSO:
BASE()

AXIS_A_OUTPUT

TYPE:
Reserved Keyword

AXIS_ADDRESS

TYPE:
Axis Parameter (**MC_CONFIG**)

DESCRIPTION:
The **AXIS_ADDRESS** parameter holds the address of the drive or feedback device. For example can be used to specify the Sercos drive address or AIN channel that is used for feedback on the base axis.

VALUE:
Drive address / node number or analogue input number

You may require additional Feature Enable Codes before using the remote axis functionality.

EXAMPLE:
Assigning the Sercos drive with the node address 4 to axis 8 in the controller. Then starting it in position mode with drive registration.

```
BASE(8)
AXIS_ADDRES = 4
ATYPE = 58
```

AXIS_B_OUTPUT

TYPE:
Reserved Keyword

AXIS_DEBUG_A

TYPE:
Reserved Keyword

DESCRIPTION:
Use only when instructed by Trio as part of an operational analysis.

AXIS_DEBUG_B

TYPE:
Reserved Keyword

DESCRIPTION:
Use only when instructed by Trio as part of an operational analysis.

AXIS_DISPLAY

TYPE:
Reserved Keyword

AXIS_DPOS

TYPE:
Axis Parameter (Read Only)

ALTERNATE FORMAT:
TRANS_DPOS

DESCRIPTION:

AXIS_DPOS is the axis demand position at the output of the **FRAME** transformation.

AXIS_DPOS is normally equal to **DPOS** on each axis. The frame transformation is therefore equivalent to 1:1 for each axis (**FRAME** = 0). 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. In this situation when **FRAME** is not zero **AXIS_DPOS** returns the demand position for the actual motor.

AXIS_DPOS is set to **MPOS** when **SERVO** or **WDOG** are OFF

VALUE:

The axis demand position at the output of the **FRAME** transformation in **AXIS_UNITS**. Default 0 on power up.

EXAMPLE:

Return the axis position in user **AXIS_UNITS** using the command line.

```
>>PRINT AXIS_DPOS
125.22
>>
```

SEE ALSO:

AXIS_UNITS, FRAME

AXIS_ENABLE

TYPE:

Axis Parameter

DESCRIPTION:

Can be used to independently disable an axis. ON by default, can be set to OFF to disable the axis. The axis is enabled if **AXIS_ENABLE** = ON and **WDOG** = ON.

On stepper axis **AXIS_ENABLE** will turn on the hardware enable outputs.

If the axis is part of a **DISABLE_GROUP** and an error occurs **AXIS_ENABLE** is set to **OFF** but the **WDOG** remains **ON**.

VALUE:

Accepts the values ON or OFF, default is ON.

EXAMPLE:

Re-enabling a group of axes after a motion error

```
DEFPOS(0)         ‘Clear the error
For axis_number = 4 to 8
BASE(axis_number)
AXIS_ENABLE = ON  ‘Enable the axis
NEXT axis_number
```

SEE ALSO:

DISABLE_GROUP

AXIS_ERROR_COUNT

TYPE:

Axis Parameter.

DESCRIPTION:

Each time there is a communications error on a digital axis, the **AXIS_ERROR_COUNT** parameter is incremented. Where supported, this value can be used as an indication of the error rate on a digital axis. Not all digital axis types have the ability to count the errors. Further information can be found in the description of each type of digital communications bus.

VALUE:

The communications error count since last reset.

EXAMPLE:

Initialise the error counter

```
AXIS_ERROR_COUNT = 0
```

In the terminal, check the latest error count value.

```
>>?AXIS_ERROR_COUNT AXIS(3)
10.0000
>>
```

Keep a record of the overall error rate for an axis.

```
TICKS = 600000
AXIS_ERROR_COUNT = 0
REPEAT
  IF TICKS<0 THEN
    VR(10) = AXIS_ERROR_COUNT ‘ number of errors counted in ten minutes
    TICKS = 600000
    AXIS_ERROR_COUNT = 0
  ENDIF
  …
  …
UNTIL FALSE
```

AXIS_FS_LIMIT

TYPE:

Axis Parameter

DESCRIPTION:

An end of travel limit may be set up in software thus allowing the program control of the working range of an axis. This parameter holds the absolute position of the forward travel limit in user **AXIS_UNITS**.

Bit 16 of the **AXISSTATUS** register is set when the axis position is greater than the **AXIS_FS_LIMIT**.

Axis software limits are only enabled when **FRAME**<>0 so that the user can limit the range of motion of the motor/ joint.

When **AXIS_DPOS** reaches **AXIS_FS_LIMIT** the controller will **CANCEL** all moves on the **FRAME_GROUP**, the axis will decelerate at **DECEL** or **FASTDEC**. Any **SYNC** is also stopped. As this software limit uses **AXIS_DPOS** it will require a negative change in **AXIS_DPOS** to move off the limit. This may not be a negative movement on **DPOS** due to the selected **FRAME** transformation..

AXIS_FS_LIMIT is disabled when it has a value greater than **REP_DIST** or when **FRAME**=0.

VALUE:

The absolute position of the software forward travel limit in user **UNITS**. (default = 200000000000)

EXAMPLES:

Set up an axis software limit so that the axis operates between 180 degrees and 270 degrees. The encoder returns 4000 counts per revolution.

```
AXIS_UNITS=4000/360
AXIS_FS_LIMIT=270
AXIS_RS_LIMIT=180
```

SEE ALSO:

AXIS_DPOS, AXIS_RS_LIMIT, AXIS_UNITS, FS_LIMIT, FWD_IN, REV_IN, RS_LIMIT

AXIS_MODE

TYPE:

Axis Parameter

DESCRIPTION:

This parameter enables various different features that an axis can use.

VALUE:

Bit	Description	Value
1	Prevents **CONNECT** from canceling when a hardware or software limit is reached, the ratio is set to 0.	2
2	Enable 3D direction calculations (default 2D)	4
6	Use non sign-extended analogue feedback	64

EXAMPLES:

EXAMPLE 1:

Enable bit 2 so that you can use 3D direction calculations, the AND is used so that only bit 2 is changed.

```
AXIS_MODE AXIS(18) = AXIS_MODE AXIS(18) AND 4
```

EXAMPLE 2:

Enable bit 6 so that you can use a 0 to 10V analogue input as axis feedback. The AND is used so that only bit 6 is changed.

```
BASE(5)
AXIS_MODE = AXIS_MODE AND 64
```

SEE ALSO:

ERRORMASK, DATUM(0)

AXIS_OFFSET

TYPE:

Slot Parameter (**MC_CONFIG** / **FLASH**)

DESCRIPTION:

AXIS_OFFSET is the first axis number that a slot tries to assign its axis to. If the axis is already being used (its **ATYPE** is non zero) then the axis is assigned to the next free axis. The controller will assign the axis depending on their SLOTs and the module type as per the following sequence:

1. EtherCAT and Panasonic axis will be assigned by **SLOT** to the first available axis starting at **AXIS_OFFSET** (plus node address -1 for Ethercat)
2. Then FlexAxis will be assigned by **SLOT** to the first available axis starting at **AXIS_OFFSET**
3. The built in axis is assigned to the first available axis starting at **AXIS_OFFSET**
4. Finally any BASIC axis are assigned as per the BASIC program. This includes **SLM** and **SERCOS** as well as any EtherCAT or Panasonic axis that is configured in BASIC.

The axis assignment is only performed on power up. **AXIS_OFFSET** should be put in the MC_**CONFIG** script to take effect immediately.

VALUE:

The first axis that the module tries to assign its axis to, range = 0 to max axis, default = 0.

EXAMPLES:

EXAMPLE 1:

```
SLOT -1 = built in, AXIS_OFFSET=0
SLOT 0 = EtherCAT, 4 axis, no node addresses set, AXIS_OFFSET=0
AXIS(0-3) Ethercat
AXIS(4) Built in
AXIS_OFFSET SLOT(0)=0
AXIS_OFFSET SLOT(-1)=0
```

This is the default case.

EXAMPLE 2:

```
SLOT -1 = built in, AXIS_OFFSET=2
SLOT 0 = EtherCAT, 4 axis, no node addresses set, AXIS_OFFSET=0
AXIS(0-3) Ethercat
AXIS(4) Built in
AXIS_OFFSET SLOT(0)=0
AXIS_OFFSET SLOT(-1)=2
```

The built in is still last as it is assigned last, the controller tries to assign the built in axis to the ***first*** available axis from 2 which is 4.

EXAMPLE 3:

```
SLOT -1 = built in, AXIS_OFFSET=0
SLOT 0 = EtherCAT, 4 axis, no node addresses set, AXIS_OFFSET=1
AXIS(0) Built in
AXIS(1-4) Ethercat
AXIS_OFFSET SLOT(0)=1
AXIS_OFFSET SLOT(-1)=0
```

The offset pushes the Ethercat out one axis so **AXIS**(0) is still spare when the built in axis is assigned

EXAMPLE 4:

```
SLOT -1 = built in, AXIS_OFFSET=0
SLOT 0 = EtherCAT, 4 axis, node switches on the drives set to 2, 3, 4,
5, AXIS_OFFSET=0
AXIS(0) Built in
```

```
AXIS(1-4) Ethercat
AXIS_OFFSET SLOT(0)=0
AXIS_OFFSET SLOT(-1)=0
```

The EtherCAT axis are set from their node address-1+**AXIS_OFFSET**

EXAMPLE 5:

```
SLOT -1 = built in, AXIS_OFFSET=0
SLOT 0 = EtherCAT, 4 axis, nodes set to 2, 3, 4, 5, AXIS_OFFSET=1
AXIS(0) Built in
AXIS(2-5) Ethercat
AXIS_OFFSET SLOT(0)=1
AXIS_OFFSET SLOT(-1)=0
```

The EtherCAT axis are set from their node address-1+**AXIS_OFFSET**

EXAMPLE 6:

```
SLOT -1 = built in, AXIS_OFFSET=0
SLOT 0 = FlexAxis, 8 axis module, AXIS_OFFSET=1
AXIS(0) Built in
AXES(1-8) FlexAxis
AXIS_OFFSET SLOT(-1)=0
AXIS_OFFSET SLOT(0)=1
```

AXIS_RS_LIMIT

TYPE:

Axis Parameter

DESCRIPTION:

An end of travel limit may be set up in software thus allowing the program control of the working range of an axis. This parameter holds the absolute position of the reverse travel limit in user **AXIS_UNITS**.

Bit 17 of the **AXISSTATUS** register is set when the axis position is less than the **AXIS_RS_LIMIT**.

Axis software limits are only enabled when **FRAME**<>0 so that the user can limit the range of motion of the motor/ joint.

When **AXIS_DPOS** reaches **AXIS_RS_LIMIT** the controller will **CANCEL** all moves on the **FRAME_GROUP**, the axis will decelerate at **DECEL** or **FASTDEC**. Any **SYNC** is also stopped. As this software limit uses **AXIS_DPOS** it will require a positive change in **AXIS_DPOS** to move off the limit. This may not be a positive movement on **DPOS** due to the selected **FRAME** transformation..

AXIS_RS_LIMIT is disabled when it has a value greater than **REP_DIST** or when *FRAME*=0.

VALUE:

The absolute position of the software forward travel limit in user **UNITS**. (default = 200000000000)

EXAMPLES:

An arm on a robots joint can move 90degrees. The encoder returns 400 counts per revolution and there is a 50:1 gearbox

```
AXIS_UNITS=4000*50/360
AXIS_FS_LIMIT=0
AXIS_RS_LIMIT=90
```

SEE ALSO:

AXIS_DPOS, **AXIS_FS_LIMIT**, **AXIS_UNITS**, **FS_LIMIT**, **FWD_IN**, **REV_IN**, **RS_LIMIT**

The built-in axis would normally be put after the Flexaxis. Here the Flexaxis is forced to start at axis 1, therefore the built-in axis can take axis 0.

AXIS_UNITS

TYPE:

Axis Parameter

DESCRIPTION:

AXIS_UNITS is a conversion factor that allows the user to scale the edges/ stepper pulses to a more convenient scale. **AXIS_UNITS** is only used when a **FRAME** is active and only applies to the parameters in the axis coordinate system (after the **FRAME**). This includes **AXIS_DPOS**, **AXIS_FS_LIMIT**, **AXIS_RS_LIMIT** and **MPOS**.

MPOS will use **UNITS** when **FRAME** =0 and **AXIS_UNITS** when **FRAME** <> 0

VALUE:

The number of counts per required units (default =1). Examples:

EXAMPLE:

A motor on a robot has an 18bit encoder and uses an 18bit encoder and 31:1 ratio gearbox. To simplify reading **AXIS_DPOS** the user wants to use radians.

```
encoder_bits = 2^10
```

```
gearbox_ratio = 31
radians_conversion=2*PI
AXIS_UNITS=( encoder_bits * gearbox_ratio)/ radians_conversion
```

SEE ALSO:
AXIS_DPOS, **UNITS**

AXIS_Z_OUTPUT

TYPE:
Reserved Keyword

AXISSTATUS

TYPE:
Axis Parameter (Read Only)

DESCRIPTION:
The **AXISSTATUS** axis parameter may be used to check various status bits held for each axis fitted:

VALUE:
21 bit value, each bit represents a different status bit.

Bit	Description	Value	char
0	Speed limit active	1	l
1	Following error warning range	2	w
2	Communications error to remote drive	4	a
3	Remote drive error	8	m
4	In forward hardware limit	16	f
5	In reverse hardware limit	32	r
6	Datuming in progress	64	d
7	Feedhold active	128	h
8	Following error exceeds limit	256	e
9	**FS_LIMIT** active	512	x

Bit	Description	Value	char
10	**RS_LIMIT** active	1024	y
11	Canceling move	2048	c
12	Pulse output axis overspeed	4096	o
13	**MOVETANG** decelerating	8192	t
15	**VOLUME_LIMIT** active	32768	v
16	**AXIS_FS_LIMIT** active	65536	i
17	**AXIS_RS_LIMIT** active	131072	j
18	Encoder power supply overload	262144	p
19	**HW_PSWITCH FIFO** not empty	524288	n
20	**HW_PSWITCH FIFO** full	1048576	b

Motion Perfect uses the characters to display the error in the Axis Parameters window.

EXAMPLES:

EXAMPLE 1:

Check bit 4 to see if the axis is in forward limit.

```
IF (AXISSTATUS AND 16)>0 THEN
  PRINT “In forward limit”
ENDIF
```

EXAMPLE 2:

Check bit 3 to see if there is a remote drive error.

```
IF AXISSTATUS.3 = ON THEN
  PRINT “Remote drive error”
ENDIF
```

SEE ALSO:

ERRORMASK, DATUM(0)

AXISVALUES

TYPE:
AXIS Command

SYNTAX:
`AXISVALUES(axis,bank)`

DESCRIPTION:
Used by *Motion* Perfect to read a bank of axis parameters.

The data is returned in the format:

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

<Parameter> is the name of the parameter

<type> is the type of the value:

i	integer
F	float
S	string
C	string of upper and lower case letters, where upper case letters mean an error

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

PARAMETERS:

axis:	the axis number where you want to read the parameters	
bank:	the bank of parameters that you wish to read.	
	0	displays the data that is only adjusted through the TrioBASIC
	1	displays the data that is changed by the motion generator.

B_SPLINE

TYPE:
Command

SYNTAX:
`B_SPLINE(mode, {parameters})`

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

PARAMETERS:

mode:	1	Standard B-Spline
	2	Non-uniform Rational B-Spline

MODE = 1:

SYNTAX:
`B_SPLINE(1, data_in, points, data_out, expansion_ratio)`

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

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

PARAMETERS:

data_in:	Location in the **TABLE** where the source profile is stored.
points:	Number of points in the source profile.
data_out:	Location in the **TABLE** where the expanded profile will be stored.

expansion_ratio:	The expansion ratio of the **B_SPLINE** function. Total output points = (Number of points+1) * expansion (i.e. if the source profile is 100 points and the expansion ratio is set to 10 the resulting profile will be 1010 point ((100+1) * 10).

EXAMPLE:

Expands a 10 point profile in **TABLE** locations 0 to 9 to a larger 110 point profile starting at **TABLE** address 200.

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

MODE = 2:

SYNTAX:

```
B_SPLINE(2, dimensions, curve_type, weight_op, points, knots, expansion, in_data, out_data)
```

DESCRIPTION:

Non Uniform Rational B-Splines, commonly referred to as **NURBS**, have become the industry standard way of representing geometric surface information designed by a CAD system

NURBS provide a unified mathematical basis for representing analytic shapes such as conic sections and quadratic surfaces, as well as free form entities, such as car bodies and ship hulls.

NURBS are small for data portability and can be scaled to increase the number of target points along a curve, increasing accuracy. A series of **NURBS** are used to describe a complex shape or surface.

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

PARAMETERS:

dimensions:	Defines the number of axes. Reserved for future use must be 3.
curve_type:	Classification of the type of **NURBS** curve. Reserved for future use must be 3.
weight_op:	Sets the weighting of the knots 0 = All weighting set to 1.
knots:	Number of knots defined.
points:	Number of data points.
expansion:	Defines the number of points the expanded curve will have in the table. Total output points = Number of points * expansion. Minimum value = 3.

in_data:	Location of input data.
out_data:	Table start location for output points stored X0, Y0, Z0 etc.

EXAMPLE:

Starting with 9 sets of X Y Z data point and expanding by 5, resulting with 45 sets of X Y Z data points (135 table points). The profile is then split from the XYZ groups into separate axis so that the profiles can be executed using **CAMBOX**.

```
weight_op=0    ‘0 sets all weights to 1.0
points=9       ‘number of data points
knots=13       ‘number of knots
expansion=5    ‘expansion factor
in_data=100    ‘data points
out_data=1000  ‘table location to construct output

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

‘Knots:
TABLE,0,0,0,0,146.8154,325.6644,536.0555,763.4151,910.1338,1109.0886)
TABLE(137,1109.0886,1109.0886,1109.0886)

‘Expand the curve, generate 5*9=45 XYZ points
‘or 135 table locations

B_SPLINE(2, 3, 3, weight_op, points, knots, expansion, in_data, out_
data)

‘Split the profile into X Y Z
FOR p= 0 TO 44
    TABLE(8000+p,TABLE(1000+(p*3)+0))
    TABLE(10000+p,TABLE(1000+(p*3)+1))
    TABLE(12000+p,TABLE(1000+(p*3)+2))
NEXT p

‘Execute the profile using CAMBOX, synchronised using axis 4
```

```
BASE(0)
DEFPOS(0,0,0,0)
CAMBOX(8000,8044,1,100,4)
BASE(1)
CAMBOX(10000,10044,1,100,4)
BASE(2)
CAMBOX(12000,12044,1,100,4)
BASE(4)
MOVE(100)
```

BACKLASH

TYPE:

Axis Command

SYNTAX:

BACKLASH(enable [,distance, speed, acceleration])

DESCRIPTION:

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

The backlash compensation is applied after a reversal of the direction of change of the **DPOS** parameter.

The backlash compensation can be seen in the **AXIS_DPOS** axis parameter. This is effectively **DPOS** + backlash compensation.

PARAMETERS:

enable:	ON to enable **BACKLASH**
	OFF to disable **BACKLASH**
distance:	The distance to be offset in user units
speed:	The speed at which is the compensation move is applied in user units
acceleration:	The **ACCEL/DECEL** rate at which is compensation move is applied in user units

EXAMPLES

EXAMPLE 1:

```
'Apply backlash compensation on axes 0 and 1:
BACKLASH(ON,0.5,10,50) AXIS(0)
BACKLASH(ON,0.4,8,50) AXIS(1)
```

EXAMPLE 2:

```
'Turn off backlash compensation on axis 3:
BASE(3)
BACKLASH(OFF)
```

SEE ALSO:

AXIS_DPOS

BACKLASH_DIST

TYPE:

Axis Parameter

DESCRIPTION:

Amount of backlash compensation that is being applied to the axis when **BACKLASH** is ON.

EXAMPLE:

Illuminate a lamp to show that the backlash has been compensated for.

```
IF BACKLASH_DIST>100 THEN
  OP (10, ON)  'show that backlash compensation has reached
               'this value
ELSE
  OP (10, OFF)
END IF
```

SEE ALSO:

BACKLASH

BASE

TYPE:

Process Command

SYNTAX:

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

ALTERNATE FORMAT:

`BA(...)`

DESCRIPTION:

The **BASE** command is used to direct all subsequent motion commands and axis parameter read/writes to a particular axis, or group of axes. The default setting is a sequence: 0, 1, 2, 3...

Each process has its own **BASE** group of axes and each program can set **BASE** values independently. So the **BASE** array will be different for each of your programs and the command line.

The values are stored in an array, when you adjust **BASE** the controller will automatically fill in the remaining positions by continuing the sequence and then adding the missed values at the end.

The **BASE** array can be printed on the command line by simply entering **BASE**

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.

The **BASE** array must use ascending values

EXAMPLES:

EXAMPLE 1:

Setting the base array to non sequential values and printing them back on the command line. This example uses a 16 axis controller.

The controller automatically continues the sequence with 10 and then fills in the missed values at the end of the list.

```
>>BASE(1,5,9)
>>BASE
(1, 5, 9, 10, 11, 12, 13, 14, 15, 0, 2, 3, 4, 6, 7, 8)
>>
```

EXAMPLE 2:

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

```
BASE(1)
UNITS=2000      'unit conversion factor
SPEED=100       'Set speed axis 1 (units/sec)
```

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

EXAMPLE 3:

Set up an interpolated move to run on axes; 0 (x), 6 (y) and 9 (z). Axis 0 will move 100 units, axis 6 will move -23.1 and axis 9 will move 1250 units. The axes will move along the resultant path at the speed and acceleration set for axis 0.

```
BASE(0,6,9)
SPEED=120
ACCEL=2000
DECEL=2500
MOVE(100,-23.1,1250)
```

SEE ALSO:

AXIS()

BASICERROR

TYPE:

System Command

DESCRIPTION:

This command is used as part of an ON... **GOSUB** or ON... **GOTO**. This lets the user handle program errors. If the program ends for a reason other than normal stopping then the subroutine is executed, this is when **RUN_ERROR**<>31.

> You should include the BASIC**ERROR** statement as the first line of the program

EXAMPLE:

When a program error occurs, print the error to the terminal and record the error number in a **VR** so that it can be displayed on an HMI through Modbus.

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

error_routine:
  VR(100) = RUN_ERROR
  PRINT “The error “;RUN_ERROR[0];
  PRINT “ occurred in line “;ERROR_LINE[0]
STOP
```

SEE ALSO:
RUN_ERROR, ERROR_LINE

BATTERY_LOW

TYPE:
System Parameter (Read only)

DESCRIPTION:
This parameter returns the condition of the non-rechargeable battery.

VALUE:

0	Battery voltage is OK
1	Battery voltage is low and needs replacing

. Bit number

TYPE:
Mathematical operator

SYNTAX:

```
<expression1>.bit_number
```

DESCRIPTION:
Returns the value of the specified bit of the expression.

As . can be used as a decimal point be careful that you only use it with an expression. There should be no spaced between the expression and the .bit_number.

PARAMETERS:

Expression1:	Any valid TrioBASIC expression
bit_number:	The bit number of the expression to return

EXAMPLES:

EXAMPLE 1:

Check the **AXISSTATUS** for remote drive errors, bit3

```
IF AXISSTATUS.3 = 1 THEN
  PRINT “Remote drive error”
ENDIF
```

EXAMPLE2:

Set **VR**(10) to 54.2, then read bit 2 of 54.

```
VR(10) = 54.2
PRINT (54).2
```

BOOT_LOADER

TYPE:

System Command (command line only)

DESCRIPTION:

Used by *Motion* Perfect to enter the boot loader software.

Do not use unless instructed by Trio or a Distributor.

BREAK_ADD

TYPE:

System Command (command line only)

SYNTAX:

BREAK_ADD “program name” line_number

DESCRIPTION:

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

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

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

PARAMETERS:

program name:	the name of any program existing on your controller
line_number:	the line umber where to insert the breakpoint

EXAMPLE:

Add a break point at line 8 of program "simpletest"

```
BREAK_ADD "simpletest" 8
```

BREAK_DELETE

TYPE:

System Command (command line only)

SYNTAX:

`BREAK_DELETE "program name" line_number`

DESCRIPTION:

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

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

PARAMETERS:

program name:	the name of any program existing on your controller
line_number:	the line umber where to remove the breakpoint

EXAMPLE:

Remove the break point at line 8 of program "simpletest"

```
BREAK_DELETE "simpletest" 8
```

BREAK_LIST

TYPE:
System Command (command line only)

SYNTAX:

```
BREAK_LIST “program name”
```

DESCRIPTION:
Used by *Motion* Perfect to returns a list of all the break points in the given program name. The program name, line number and the code associated with that line is displayed.

PARAMETERS:

program name:	the name of any program existing on your controller

EXAMPLE
Show the breakpoints from a program called “simpletest” with break points inserted on lines 8 and 11.

```
>>BREAK_LIST “simpletest”

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

BREAK_RESET

TYPE:
System Command (command line only)

SYNTAX:

```
BREAK_RESET “program name”
```

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

PARAMETERS:

program name:	the name of any program existing on your controller

EXAMPLE:

Remove all break points from program “simpletest”

```
BREAK_RESET “simpletest”
```

CAM

TYPE:
Axis Command

SYNTAX:
CAM(start point, end point, table multiplier, distance)

DESCRIPTION:
The CAM command is used to generate movement of an axis according to a table of positions which define a movement profile. The table of values is specified with the **TABLE** command. The movement may be defined with any number of points from 3 up to the maximum table size available. The controller performs linier interpolation between the values in the table to allow small numbers of points to define a smooth profile.

The **TABLE** values are translated into positions by offsetting them by the first value and then multiplying them by the multiplier parameter. This means that a non-zero starting profile will be offset so that the first point is zero and then all values are scaled with the multiplier. These are then used as absolute positions from the start position.

Two or more **CAM** commands executing simultaneously can use the same values in the table.

The speed of the CAM profile is defined through the **SPEED** of the **BASE** axis and the distance parameter. You can use these two values to determine the time taken to execute the CAM profile.

As with any motion command the **SPEED** may be changed at any time to any positive value. 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).

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

PARAMETERS:

start point:	The start position of the cam profile in the **TABLE**
end point:	The end position of the cam profile in the **TABLE**
multiplier:	The table values are multiplied by this value to generate the positions.
distance:	The distance parameter relates the speed of the axis to the time taken to complete the cam profile. The time taken can be calculated using the current axis speed and this distance parameter (which are in user units).

EXAMPLES:

EXAMPLE 1:

A system is being programmed in mm and the speed is set to 10mm/sec. It is required to take 10 seconds to complete the profile, so a distance of 100mm should be specified.

```
SPEED = 10    ‘axis SPEED
time = 10     ‘time to complete profile
distance = SPEED* time  ‘distance parameter for CAM
CAM(0, 100, 1, distance)
```

EXAMPLE2:

Motion is required to follow the **POSITION** equation:

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

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

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

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

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

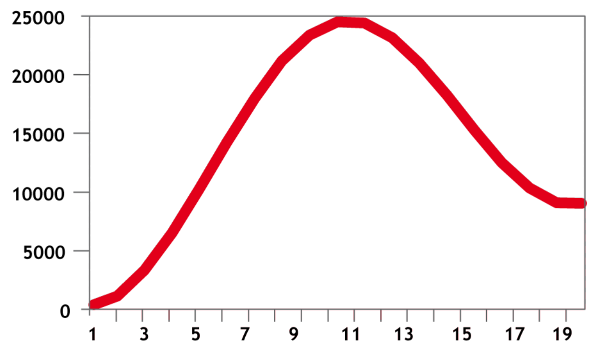


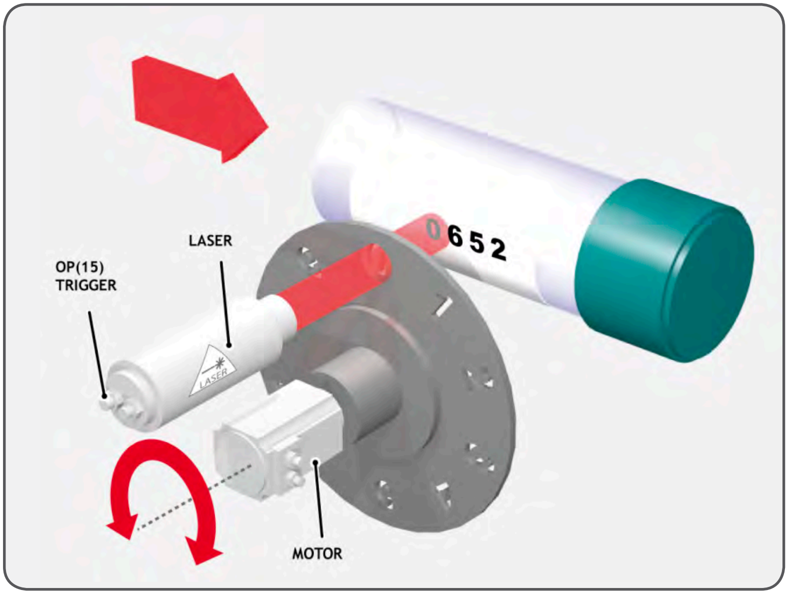
Table Position	Degrees	Value
9	160	23396
10	180	24500
11	200	24396
12	220	23160
13	240	21000
14	260	18236
15	280	15263
16	300	12500
17	320	10340
18	340	9103
19	360	9000

EXAMPLE 3:

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

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

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




```
GOSUB profile_gen
WHILE IN(2)=ON
  WAIT UNTIL KEY#1         'Waits for character on port 1
  GET#1,k
  IF k>47 AND k<58 THEN    'check for valid ASCII character
    position=(k-48)*400    'convert to absolute position
    multiplier=position-offset  'calculate relative movement
    'check if it is shorter to move in reverse direction
    IF multiplier>2000 THEN
      multiplier=multiplier-4000
    ELSEIF multiplier<-2000 THEN
      multiplier=multiplier+4000
    ENDIF
    CAM(0,200,multiplier,2500)   'set the CAM movment
    WAIT IDLE
    OP(15,ON)                    'trigger the laser flash
    WA(20)
    OP(15,OFF)
    offset=(k-48)*400  'calculates current absolute position
  ENDIF
WEND

profile_gen:
```

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

EXAMPLE 4:

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

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

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

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

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

CAMBOX

TYPE:

Axis Command

SYNTAX:

```
CAMBOX(start_point, end_point, table_multiplier, link_distance , link_
axis[, link_options][, link_pos][, offset_start])
```

DESCRIPTION:

The **CAMBOX** command is used to generate movement of an axis according to a table of **POSITIONS** which define the movement profile. The motion is linked to the measured motion of another axis to form a continuously variable software gearbox. The table of values is specified with the **TABLE** command. The movement may be defined with any number of points from 3 up to the maximum table size available. The controller interpolates between the values in the table to allow small numbers of points to define a smooth profile.

The **TABLE** values are translated into positions by offsetting them by the first value and then multiplying them by the multiplier parameter. This means that a non-zero starting profile will be offset so that the first point is zero and then all values are scaled with the multiplier. These are then used as absolute positions from the start position.

Two or more **CAMBOX** commands executing simultaneously can use the same values in the table.

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:

start_point:	The start position of the cam profile in the **TABLE**
end_point:	The end position of the cam profile in the **TABLE**
table_multiplier:	The table values are multiplied by this value to generate the positions.
link_distance:	The distance the link axis must move to complete **CAMBOX** profile.
link_axis:	The axis to link to.

link_options:	Bit value options to customize how your **CAMBOX** operates		
	Bit 0	1	link commences exactly when registration event **MARK** occurs on link axis
	Bit 1	2	link commences at an absolute position on link axis (see link_pos for start position)
	Bit 2	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)
	Bit 3	8	**PATTERN** mode. Advanced use of **CAMBOX**: allows multiple scale values to be used
	Bit 5	32	Link is only active during a positive move on the link axis
	Bit 7	128	Forces the profile to start at a defined point in the link_dist (see offset_start for the position)
	Bit 8	256	link commences exactly when registration event **MARKB** occurs on link axis
	Bit 9	512	link commences exactly when registration event **R_MARK** occurs on link axis. (see link_pos for channel number)
link_pos:	link_option bit 1 - the absolute position on the link axis in user **UNITS** where the **CAMBOX** is to be start. link_option bit 9 – the registration channel to start the movement on		
offset_start:	The position defined on the link_dist where the profile will start		

The link_dist is in the user units of the link axis and should always be specified as a positive distance.

The link options for start (bits 0, 1, 8 and 9) may be combined with the link options for repeat (bits 2 and 5) and direction as well as offset_start (bit 7).

start_pos cannot be at or within one servo period’s worth of movement of the **REP_DIST** position.

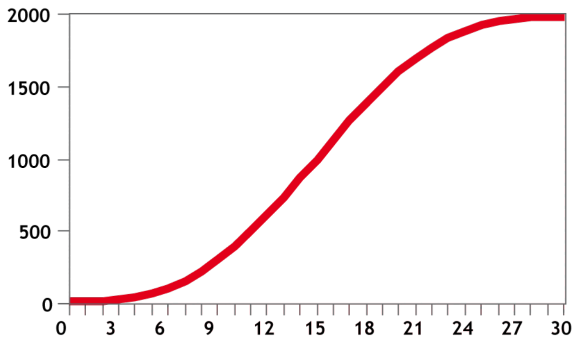
EXAMPLES:

EXAMPLE 1:

A subroutine can be used to generate a **SINE** shaped speed profile. This profile is used in the other examples.

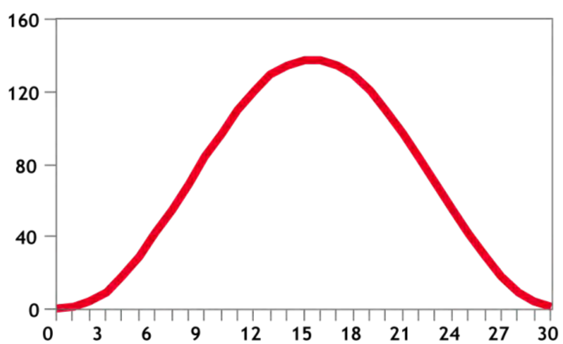
```
      ‘ p is loop counter
      ‘ num_p is number of points stored in tables pos 0..num_p
      ‘ scale is distance travelled scale factor
    profile_gen:
      num_p=30
```

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



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

Speed Curve

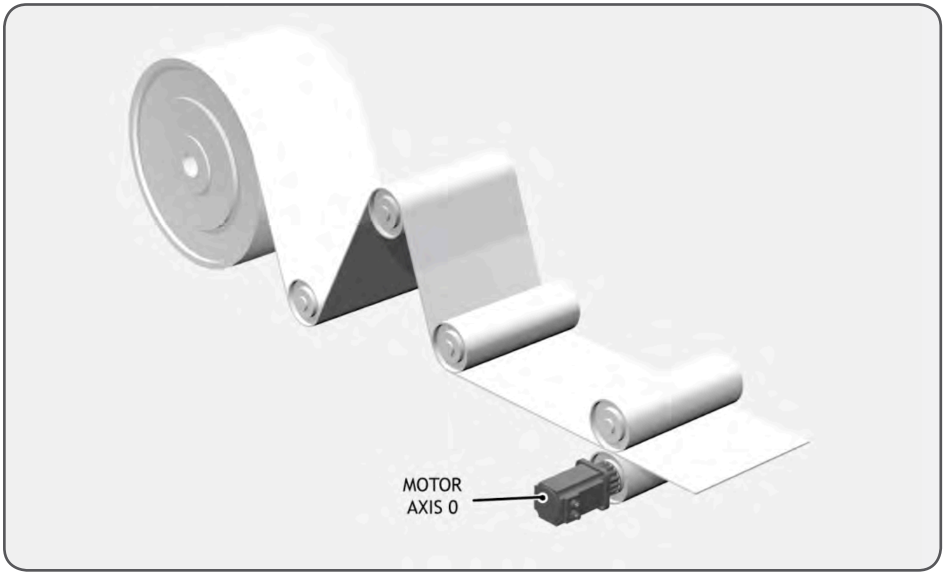


EXAMPLE 2:

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

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.
variable "start"	The motion will execute when the position "start" is reached on axis 15.

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



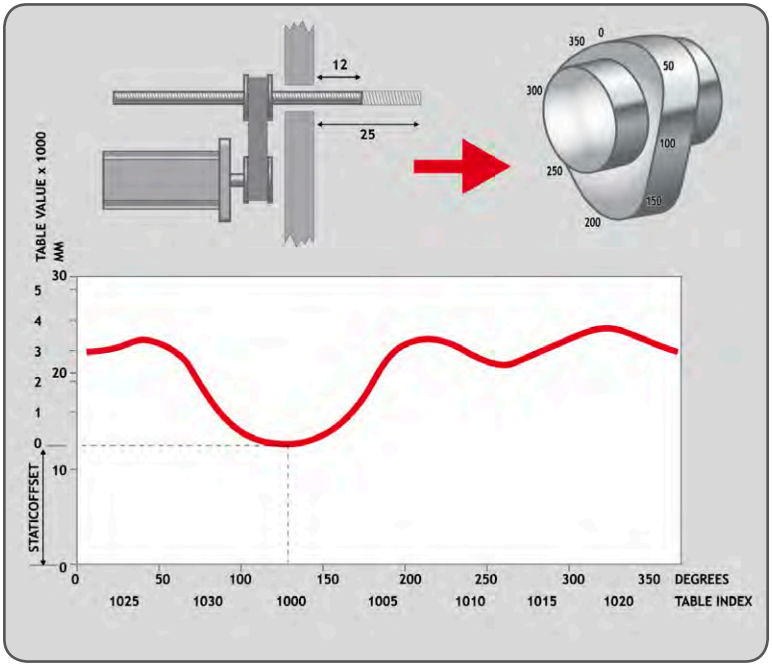
EXAMPLE 3:

A motor on Axis 0 is required to emulate a rotating mechanical CAM. The position is linked to motion on axis 3. The "shape" of the motion profile is held in **TABLE** values 1000..1035.

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

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

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

The firmware resets bit 1 of **REP_OPTION** after the repeating mode has been cancelled.



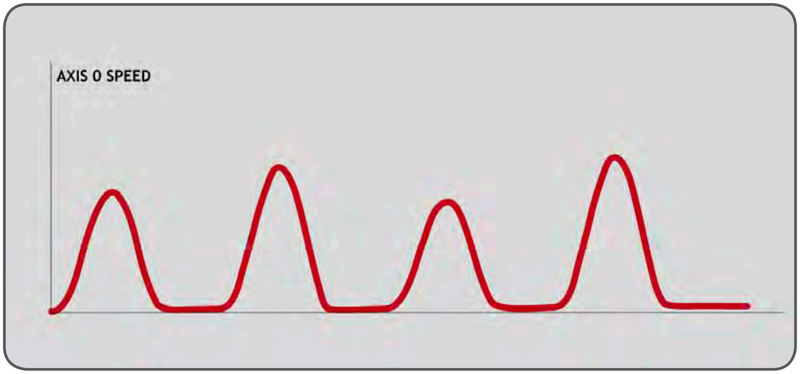
CAMBOX PATTERN MODE:

SYNTAX:

`CAMBOX(start_point, end_point, control_block_pointer, link_dist, link_axis, options)`

DESCRIPTION:

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



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 Control Block pointer. 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 two of the **CONTROL BLOCK** values. The pattern sequence is a sequence of scale factors for the **SHAPE**.

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.

PARAMETERS:

start_point:	The start position of the shape block in the **TABLE**
end_point:	The end position of the shape block in the **TABLE**
control_block_pointer:	The position in the table of the 7 point control block

link_distance:	The distance the link axis must move to complete **CAMBOX** profile.
link_axis:	The axis to link to.
options:	As **CAMBOX**, bit 3 must be enabled

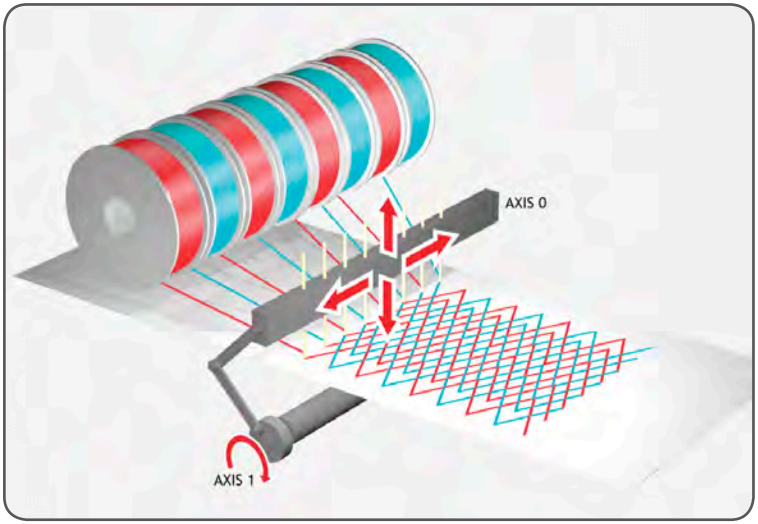
CONTROL BLOCK PARAMETERS

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

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

EXAMPLE:

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



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

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

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

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

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

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

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

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

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

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

SEE ALSO:

REP_OPTION

CAN

TYPE:

System Command

SYNTAX:

`CAN(slot, function[, parameters])`

DESCRIPTION:

This function allows the CAN communication channels to be controlled from the Trio **BASIC**. All *Motion Coordinator's* have a single built-in CAN channel which is normally used for digital and analogue I/O using Trio's I/O modules.

In addition to using the CAN command to control CAN channels, there are specific protocol functions into the firmware. 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.

Channel:	Channel Number:	Maximum Baudrate:
Built-in CAN	-1	1 Mhz

There are 16 message buffers in the controller

PARAMETERS:

slot:	Set to -1 for the built in CAN port	
function:	0	Read Register, do not use unless instructed by Trio or a Distributor.
	1	Write Register, do not use unless instructed by Trio or a Distributor.
	2	Initialise baud rate
	3	Check for message received
	4	Transmit OK
	5	Initialise message
	6	Read message
	7	Write message
	8	Read CANOpen Object
	9	Write CANOpen Object
	11	Initialise 29bit message
	20	CAN mode
	21	Enable CAN driver
	22	Reset CAN message buffer
	23	Specify CAN **VR** map
	24	Enable and configure a Sync telegram

FUNCTION = 2:

SYNTAX:

```
CAN(channel,2,baudrate)
```

DESCRIPTION:

Initialise the baud rate of the CANBus

PARAMETERS:

baudrate:	0	1MHz
	1	500kHz (default value)
	2	250kHz
	3	125kHz

FUNCTION = 3:

SYNTAX:
```
value=CAN(channel, 3, message)
```

DESCRIPTION:
Check to see if there is a new message in the message buffer

PARAMETERS:

message:	message buffer to check	
value:	**TRUE**	new message available
	FALSE	no new message

FUNCTION = 4:

SYNTAX:
```
value=CAN(channel, 4, message)
```

DESCRIPTION:
Checks that it is ok to transmit a message

PARAMETERS:

message:	message buffer to transmit	
value:	**TRUE**	OK to transmit
	FALSE	Network busy

FUNCTION = 5:

SYNTAX:

`CAN(channel#, 5, message, identifier, length, rw)`

DESCRIPTION:

Initialise a message by configuring its buffers size and if it is transmit or receive.

PARAMETERS:

message:	message buffer to initialise	
identifier:	the identifier which the message buffer appears on the CANBus	
length:	the size of the message buffer	
rw:	0	read buffer
	1	write buffer

FUNCTION = 6:

SYNTAX:

`CAN(channel, 6, message, variable)`

DESCRIPTION:

Read in the message from the specified buffer to a **VR** array.

The first **VR** holds the identifier. The subsequent values hold the data bytes from the CAN packet.

PARAMETERS:

message:	the message buffer to read in
variable:	the start position in the **VR** memory for the message to be written

FUNCTION = 7:

SYNTAX:

`CAN(channel, 7, message, byte0, byte1..)`

DESCRIPTION:

Write a message to a message buffer.

PARAMETERS:

message:	the message buffer to write the message in
byte0:	the first byte of the message
byte1:	the second byte of the message
...	

FUNCTION = 8:

SYNTAX:

```
CAN(channel, 8, transbuf, recbuf, object, subindex, variable)
```

DESCRIPTION:

Read a CANOpen object. The first **VR** holds the variable data type. The subsequent values hold the data bytes from the CAN packet.

PARAMETERS:

transbuf:	the message buffer used to transmit
recbuf:	the message buffer used to recieve
object:	the CANOpen object to read
subindex:	the sub index of the CANOpen object to read
variable:	the start position in the **VR** memory for the message to be written

FUNCTION = 9:

SYNTAX:

```
CAN(channel, 9, transbuf, recbuf, format, object, subindex, value,
{valuems})
```

DESCRIPTION:

Write a CANOpen object. This function automatically requests the send so you do not need to use function 4.

PARAMETERS:

transbuf:	the message buffer used to transmit

recbuf:	the message buffer used to recieve
format:	data size in bits 8, 16 or 32
object:	the CANOpen object to write to
subindex:	the sub index of the CANOpen object to write to
value:	the least significant 16 bits of the value to write
valuems:	the most significant 16 bit of the value to write

FUNCTION = 11:

SYNTAX:

```
CAN(channel#, 11, message, identifierms, identifier, length, rw)
```

DESCRIPTION:

Initialise a message by configuring its buffers size and if it is transmit or receive using 29 bit identifiers.

PARAMETERS:

message:	message buffer to initialise	
identifierms:	the most significant 13 bits of the identifier	
identifier:	the least significant 16 bits if the identifier	
length:	the size of the message buffer	
rw:	0	read buffer
	1	write buffer

FUNCTION = 20:

SYNTAX:

```
CAN(channel, 20,mode)
```

DESCRIPTION:

Sets the CAN mode, normally this is done using **CANIO_ADDRESS**

PARAMETERS:

Mode:	0	Disable all CAN operations
	1	CAN command mode
	2	**CANIO** mode (default)
	3	CANopenIO mode (**CANOPEN_OP_RATE** controls the cycle period, default = 5ms)

Unlike **CANIO_ADDRESS** this is **NOT** stored in flash EPROM

FUNCTION = 21:

SYNTAX:

```
CAN(channel, 21,enable)
```

DESCRIPTION:

Provides the ability to reset the CAN driver. Do not use unless instructed by Trio or a Distributor.

PARAMETERS:

Enable:	0	Disable
	1	Enable (default)

FUNCTION = 22:

SYNTAX:

```
CAN(channel, 22, message)
```

DESCRIPTION:

Reset a message buffer

PARAMETERS:

message:	the message buffer to reset

FUNCTION = 23:

SYNTAX:

```
CAN(channel, 23, [message, map, offset, length, order, variable,
direction [,data_type]])
```

DESCRIPTION:

Specify CAN **VR** map for use with CANOpenIO mode

If no parameters provided then current mappings are displayed

PARAMETERS:

message:	message buffer (0..15)
map:	MAP number (0..7)
offset:	CAN buffer byte offset (0..7)
length:	CAN buffer byte length (1..8)
order:	Endian Byte order (0=Little, 1=Big)
variable:	Index of variable in the controller
direction:	Direction (0=Receive, 1=Transmit)
data_type:	0 =inactive 1 = **VR** (default), 2 = Digital IO, 3 = Analogue IO

FUNCTION = 24:

SYNTAX:

CAN(channel, 24, enable, message, period)

DESCRIPTION:

Set up a Cyclic Sync Telegram for CANOpenIO mode. After **CANIO_ENABLE** is set to 1, the firmware will send the sync telegram at the specified period, synchronised with the internal servo cycle of the *Motion Coordinator*.

PARAMETERS:

enable:	1 = enable sync telegram, 0 = disable
message:	message buffer (0..15)
period:	Sync period in milliseconds

EXAMPLE:

```
CAN(-1,5,14,128,0,1) ‘ Set buffer 14 for SYNC CobID=$80 (128)
CAN(-1,24,1,14,4) ‘ sync telegram every 4 msec

CAN(-1,7,15,1,0) ‘ Set the CanOpen slave modules to run state
CANIO_ENABLE=1
```

SEE ALSO:
CANIO_ADDRESS, CANOPEN_OP_RATE

CANCEL

TYPE:
Axis Command

SYNTAX:
CANCEL([mode])

ALTERNATE FORMAT:
CA([mode])

DESCRIPTION:
Used to cancel current or buffered axis commands on an axis or an interpolating axis group. Velocity profiled moves, for example; **FORWARD**, **REVERSE**, **MOVE**, **MOVEABS**, **MOVECIRC**, **MHELICAL**, **MOVEMODIFY**, will be ramped down at the programmed **DECEL** or **FASTDEC** rate then terminated. Other move types will be terminated immediately.

CANCEL can be called manually, but also automatically by software limits, hardware limits and **MOTION_ERROR**s.

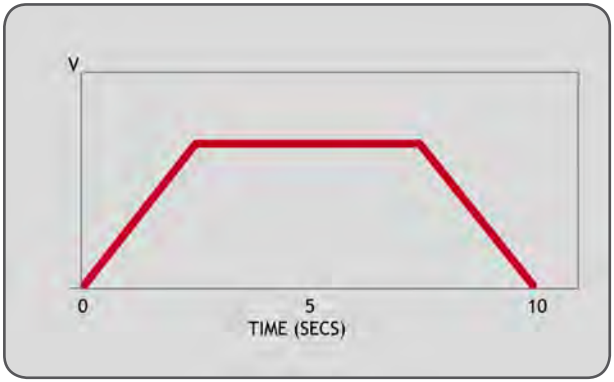
PARAMETERS:

mode:	0	Cancels axis commands from the **MTYPE** buffer. Can be used without the parameter
	1	Cancels all buffered moves on the base axis (excluding the **PMOVE**)
	2	Cancels all active and buffered moves including the **PMOVE** if it is to be loaded on the **BASE** axis

CANCEL will only cancel the presently executing move. If further moves are buffered they will then be loaded and the axis will not stop.

EXAMPLES:

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




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

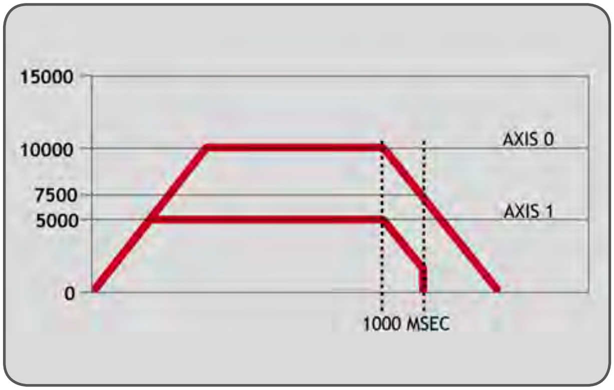
EXAMPLE 2:

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

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

EXAMPLE 3:

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




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

SEE ALSO:

RAPIDSTOP, FASTDEC

CANIO_ADDRESS

TYPE:

System Parameter (**MC_CONFIG** / **FLASH**)

DESCRIPTION:

CANIO_ADDRESS is used to set the operating mode of the CANBus. You can select between Trio CAN, DeviceNet, CANOpen and a user configuration when implementing your own can protocol.

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.

If the value is not set to 32 then you cannot connect to Trio CAN I/O

VALUES:

32	Trio CAN I/O Master 64in/64out
33	DeviceNet
34...39	User range
40	CanOpen I/O Master 64in/64out
41	CanOpen I/O Master 128in/128out
42	CANOpen I/O Master custom mapping

CANIO_BASE

TYPE:
System Parameter (**MC_CONFIG**)

DESCRIPTION:
This parameter sets the start address of any CAN module I/O channels. Together with **MODULEIO_BASE**, DRIVEIO_BASEand **NODE_IO** the I/O allocation scheme can replace and expand the behaviour of **MODULE_IO_MODE**, however MODULE_IO_MODEtakes precedence if its value has been changed to 2 (**CANIO** followed by **MODULE** IO).

VALUE:

-1	No effect (**CANIO** should be disabled using **CANIO_ADDRESS**)
0	CAN I/O allocated automatically (default)
>= 8	CAN I/O is located at this IO point address, truncated to the nearest multiple of 8

EXAMPLE:
A system with MC464, a Panasonic module (slot 0) and a **CANIO** Module will have the following I/O assignment:

CANIO_BASE=0 + **DRIVEIO_BASE**=0 + **MODULEIO_BASE**=0

0-7	Built in inputs
8-15	Built in bi-directional I/O
16-23	Panasonic module inputs
24-39	**CANIO** bi-directional I/O

40-47	Panasonic drive inputs
48-1023	Virtual I/O

CANIO_BASE=100 + **DRIVEIO_BASE**=0 + **MODULEIO_BASE**=0

0-7	Built in inputs
8-15	Built in bi-directional I/O
16-23	Panasonic module inputs
24-31	Panasonic drive inputs
32-95	Virtual I/O
96-103	**CANIO** bi-directional I/O
104-1023	Virtual I/O

SEE ALSO:
MODULEIO_BASE, DRIVEIO_BASE, NODE_IO, MODULE_IO_MODE

CANIO_ENABLE

TYPE:
System Parameter

DESCRIPTION:
CANIO_ENABLE enables the Trio CAN I/O or CANOpen protocol.

When using the Trio I/O protocol it is set automatically by firmware. You have to set **CANIO_ENABLE**=ON manually after configuring CANOpen IO.

VALUE:

ON	Enable the CAN protocol (default when **CANIO_ADDRESS**=32)
OFF	Disable the CAN protocol (default when **CANIO_ADDRESS**<>32)

CANIO_MODE

TYPE:
System Parameter (**MC_CONFIG** / **FLASH**)

DESCRIPTION:
CANIO_MODE is used to set the operating mode of the Trio CAN I/O system. The MC4xx *Motion Coordinators* allow separate Input and Output modules to occupy overlapping addresses. This allows up to 32 Input and Output modules to be connected. Alternatively, the **CANIO_MODE** can be set to force the MC4xx *Motion Coordinator* to work in the same way as the MC2xx series, with only 16 digital modules of any type allowed.

The value is held in flash EPROM and can be set in the **MC_CONFIG** script.

VALUE:

0	MC4xx CAN IO addressing (default)
1	Compatibility mode CAN IO addressing

CANIO_STATUS

TYPE:
System Parameter

DESCRIPTION:
Returns the status of the Trio CAN I/O network. You can set bit 4 to reset the network.

VALUE:

Bit	Description	Value
0	Error from the I/O module 0,3,6 or 9	1
1	Error from the I/O module 1,4,7 or 10	2
2	Error from the I/O module 2,5,8 or 11	4
3	Error from the I/O module 12,13,14 or 15	8
4	Should be set to re-initialise the **CANIO** network	16
5	Is set when initialisation is complete	32
6	Error from Analogue module	64
7	Output error (0-3)	128

Bit	Description	Value
8	Output error (4-7)	256
9	Output error (8-11)	512
10	Output error (12-15)	1024
11	Input error (0-3)	2048
12	Input error (4-7)	4096
13	Input error (8-11)	8192
14	Input error (12-15)	16384

CANOPEN_OP_RATE

TYPE:
System Parameter

DESCRIPTION:
Used to adjust the transmission rate of CanOpen I/O PDO telegrams.

VALUE:
Default is 5msec. Adjustable in 1msec steps.

CHANGE_DIR_LAST

TYPE:
Axis Parameter (read only)

DESCRIPTION:
Returns the difference between the direction of the end of the previous loaded interpolated motion command and the start direction of the last loaded interpolated motion command. If there is no previous loaded command then **END_DIR_LAST** can be written to set an initial direction.

This parameter is only available when using **SP** motion commands such as **MOVESP**, **MOVEABSSP** etc.

VALUE:
Change in direction, in radians between 0 and PI. Value is always positive.

EXAMPLE:

```
Perform a 90 degree move and print the change.
>>MOVESP(0,100)
>>MOVESP(100,0)
>>PRINT CHANGE_DIR_LAST
1.5708
>>
```

SEE ALSO:

END_DIR_LAST, START_DIR_LAST

CHANNEL_READ

TYPE:

System Command

SYNTAX:

```
x = CHANNEL_READ(channel, storage_buffer[, delimiter_buffer[, escape_
character[, crc]]])
```

DESCRIPTION:

CHANNEL_READ will read bytes from the channel and store them into the storage buffer.

If the storage buffer is in **VR** then the first value specifies why the **CHANNEL_READ** stopped: 0 for end of file, 1 for the first delimiter character, 2 for the second delimiter character, etc, and the command returns the number of characters read. The string is null terminated so the **VRSTRING** command can be used to view the buffer as a string.

If the storage buffer is a named string variable then the command returns why the **CHANNEL_READ** stopped. The number of characters read can be obtained using the LEN command on the named string variable.

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

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

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

PARAMETERS:

channel	Communication or file channel.
storage_buffer	1 named string variable, or 2 numerical expressions that specify the **VR** base and length.

delimiter_buffer	1 string expression, or 2 numerical expressions that specify the **VR** base and length.
escape_character	When this character is received the following character is not interpreted.
crc	Position in the **VR** data where the CRC will be stored.

EXAMPLE 1:

Read numbers from a file: one number per line, using **VR** storage and delimiter buffers.

```
‘ create a temp file in RAM that contains the numbers 1 to 10,
‘ one line per number
OPEN #40 AS “ram:test” FOR OUTPUT(1)
FOR i=1 TO 10
    PRINT #40,i
NEXT i
CLOSE #40

‘ set the delimiters
VR(10)=13’carriage return
VR(11)=10’line feed

‘ test vr functionality
OPEN #40 AS “ram:test” FOR INPUT
PRINT “-------------------- START VR ---------------------”
REPEAT
    ‘ read channel 40.
    ‘ VR(100) has the end status
    ‘ VR(101)-VR(199) hold the data
    ‘ VR(10)-VR(11) hold the delimiters
    c=CHANNEL_READ(40,100,100,10,2)

    ‘ if we have characters then print them
    IF (c > 0) THEN
        PRINT c[0], VR(100)[0], VRSTRING(101)
    ENDIF
    IF VR(100) = 1 THEN
        PRINT “--- CARRIAGE RETURN ----”
    ELSEIF VR(100)=2 THEN
        PRINT “--- LINE FEED ----”
    ENDIF

UNTIL NOT KEY#40
PRINT “-------------------- STOP VR --------------------”
CLOSE #40
```

EXAMPLE 2:

Read numbers from a file: one number per line, using string storage and delimiter buffers.

```
‘ create a temp file in RAM that contains the numbers 1 to 10,
‘ one line per number
OPEN #40 AS “ram:test” FOR OUTPUT(1)
```

```
FOR i=1 TO 10
    PRINT #40,i
NEXT i
CLOSE #40

‘ declare the buffers
DIM b AS STRING(100)
DIM d AS STRING(2)

‘ set the delimiters
d=CHR(13)+CHR(10)

‘ test string functionality
OPEN #40 AS “ram:test” FOR INPUT
PRINT “-------------------- START STRING ---------------------”
REPEAT
    ‘ read channel 40.
    s=CHANNEL_READ(40,b,d)
    c=LEN(b)

    ‘ if we have characters then print them
    IF (c > 0) THEN
        PRINT c[0], s[0], b
    ENDIF
    IF s = 1 THEN
        PRINT “--- CARRIAGE RETURN ----”
    ELSEIF s=2 THEN
        PRINT “--- LINE FEED ----”
    ENDIF
UNTIL NOT KEY#40
PRINT “-------------------- STOP STRING ---------------------”
CLOSE #40
```

EXAMPLE 3:

Read numbers from a file: one number per line, using string storage buffer and **VR** delimiter buffer.

```
‘ create a temp file in RAM that contains the numbers 1 to 10,
‘ one line per number
OPEN #40 AS “ram:test” FOR OUTPUT(1)
FOR i=1 TO 10
    PRINT #40,i
NEXT i
CLOSE #40

‘ declare the buffers
DIM b AS STRING(100)

‘ set the delimiters
VR(10)=13’carriage return
VR(11)=10’line feed
```

```
' test string functionality
OPEN #40 AS "ram:test" FOR INPUT
PRINT "------------------- START STRING --------------------"
REPEAT
    ' read channel 40.
    s=CHANNEL_READ(40,b,10,2)
    c=LEN(b)

    ' if we have characters then print them
    IF (c > 0) THEN
        PRINT c[0], s[0], b
    ENDIF
    IF s = 1 THEN
        PRINT "--- CARRIAGE RETURN ----"
    ELSEIF s=2 THEN
        PRINT "--- LINE FEED ----"
    ENDIF
UNTIL NOT KEY#40
PRINT "------------------- STOP STRING --------------------"
CLOSE #40
```

CHECKSUM

TYPE:
Reserved Keyword

CHR

TYPE:
String Function

SYNTAX:
`value = CHR(number)`

DESCRIPTION:
CHR returns the **ASCII** character as a **STRING** which is referred to by the number, this can be assigned to a **STRING** variable or be PRINTed.

Parameters:

number:	Any valid numerical value for an **ASCII** character

value:	A **STRING** containing the character

EXAMPLES:

EXAMPLE 1:
Print the character A on the command line

```
>>PRINT CHR(65)
A
>>
```

EXAMPLE 2:
Print a line of text terminating only with a carriage return

```
PRINT#5, "abcdefghijk"; CHR(13)
```

EXAMPLE 3:
Append a character from the serial port to a **STRING** variable

```
DIM value AS STRING
WHILE KEY#5
  GET#5, char
  value = value + CHR(char)
WEND
```

SEE ALSO:
PRINT, STRING

CLEAR

TYPE:
System Command

DESCRIPTION:
Sets all global (numbered) variables and **VR** values to 0 and sets local variables on the process on which command is run to 0.

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:

```
Setting and clearing VR values.
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_BIT

TYPE:

Logical and Bitwise Command

SYNTAX:

`CLEAR_BIT(bit, variable)`

DESCRIPTION:

CLEAR_BIT can be used to clear the value of a single bit within a **VR**() variable.

PARAMETERS:

bit:	The bit number to clear, valid range is 0 to 52
variable:	The **VR** on which to operate

EXAMPLE:

Set bit 6 in **VR** 23 to zero.

```
CLEAR_BIT(6,23)
```

SEE ALSO

READ_BIT, SET_BIT

CLEAR_PARAMS

TYPE:
System Command (command line only)

DESCRIPTION:
Resets all flash parameters to the default value. This command must only be used on the command line.

You must cycle power after issuing this command to ensure that all parameters take effect.

This will reset the **IP** address to the default value and so you may not be able to connect after cycling power.

You should use the **MC_CONFIG** file to set all **FLASH**/ **MC_CONFIG** parameters so that they are saved as part of the project.

CLOSE

TYPE:
System command

SYNTAX:
CLOSE channel

DESCRIPTION:
CLOSE will close the file on the specified channel.

PARAMETERS:

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

SEE ALSO:
OPEN

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.

VALUE:
Position of the end of the position window in user units.

EXAMPLE:
Set a position window between 10 and 30

```
OPEN_WIN = 10
CLOSE_WIN = 30
```

SEE ALSO:
OPEN_WIN, REGIST

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.

VALUE:
Change in connection ratio per second (default 1000000)

EXAMPLE:
The connection ratio will be changed from 0 to 6 when an input is set. It is required to take 2 second to accelerate the linked axis so the ratio must change at 3 per second.

```
CLUTCH_RATE = 3
```

```
CONNECT(0,0)
WAIT UNTIL IN(1)=ON
CONNECT(6,0)
```

CO_READ

TYPE:
System Command

SYNTAX:
CO_READ(slot, address, index, subindex ,type [,vr_number])

DESCRIPTION:
This function gets a CANopen-over-EtherCAT object from the remote drive or IO device. The Object's index and sub-index are used to request a value and that value is either placed in the **VR** or is displayed in the *Motion* Perfect terminal if the **VR** number is set to -1.

Refer to the remote device's manual for a list of available objects. If the object value is returned successfully, the command returns **TRUE**. (-1) Otherwise, in the case of an error while requesting the value, the command returns **FALSE**.

PARAMETERS:

slot:	Slot number of the EtherCAT module.	
address:	Node address of the remote device on the network	
index:	CANopen Object index	
subindex:	CANopen Object sub-index	
Type:	1	Boolean
	2	Integer 8
	3	Integer 16
	4	Integer 32
	5	Unsigned 8
	6	Unsigned 16
	7	Unsigned 32
	9	Visible String (to terminal only)

vr_number:	VR number between 0 and max **VR** where the result will be stored. (-1 means the value will be printed to the terminal)

EXAMPLES:

EXAMPLE 1:

Read the remote drive mode of operation and display to the terminal

```
>>CO_READ(0, 1, $6061, 0, 2, -1)
8
>>
```

EXAMPLE 2:

Get the remote drive interpolation time, objects $60C2 sub-index 1 and sub-index 2, and place in **VR**(200) and **VR**(201).

```
'read object $60C2:01 unsigned 8
CO_READ(0, 5, $60C2, 1, 5, 200)
'read object $60C2:02 signed 8
CO_READ(0, 5, $60C2, 2, 2, 201)
PRINT "Drive at node 5: "; VR(200)[0];"x 10^";VR(201)[0]
```

CO_READ_AXIS

TYPE:

System Command

SYNTAX:

CO_READ_AXIS(axis_number, index, subindex ,type [,vr_number])

DESCRIPTION:

This function gets a CANopen-over-EtherCAT object from the remote drive or IO device. The Object's index and sub-index are used to request a value and that value is either placed in the **VR** or is displayed in the *Motion* Perfect terminal if the **VR** number is set to -1.

Refer to the remote device's manual for a list of available objects. If the object value is returned successfully, the command returns **TRUE**. (-1) Otherwise, in the case of an error while requesting the value, the command returns **FALSE**.

PARAMETERS:

Axis_number:	Axis number of the EtherCAT drive.
index:	CANopen Object index

subindex:	CANopen Object sub-index	
Type:	1	Boolean
	2	Integer 8
	3	Integer 16
	4	Integer 32
	5	Unsigned 8
	6	Unsigned 16
	7	Unsigned 32
	9	Visible String (to terminal only)
vr_number:	VR number between 0 and max **VR** where the result will be stored. (-1 means the value will be printed to the terminal)	

EXAMPLES:

EXAMPLE 1:

Print the value for object 0x6064 sub-index 00, position actual value. This is a 32 bit long word and so has the CANopen type 4.

```
>>CO_READ_AXIS(3, $6064, 0, 4, -1)
5472
>>
```

EXAMPLE 2:

Get the proportional gain and velocity feedforward gain from the remote drive, and place in **VR**(200) and **VR**(201). Perform a check to make sure the object is supported by the drive.

```
IF CO_READ_AXIS(2, $60FB, 1, 6, 200) = FALSE THEN
  PRINT “Error reading Object $60FB:01”
ELSE
  PRINT “Drive P Gain = “;VR(200)[0]
ENDIF
IF CO_READ_AXIS(2, $60FB, 2, 6, 201) = FALSE THEN
  PRINT “Error reading Object $60FB:02”
ELSE
  PRINT “Drive VFF Gain = “;VR(201)[0]
ENDIF
```

CO_WRITE

TYPE:

System Command

SYNTAX:

CO_WRITE(slot, address, index, subindex ,type, vr_number [,value])

DESCRIPTION:

This function sets a CANopen-over-EtherCAT object in the remote drive or IO device. The Object's index and sub-index are used to write a value to that object. The value can come from a **VR** or is put into the command directly if the **VR** number is set to -1.

Refer to the remote device's manual for a list of available objects. If the object value is set successfully, the command returns **TRUE**. (-1) Otherwise, in the case of an error while writing the value, the command returns **FALSE**.

PARAMETERS:

slot:	Slot number of the EtherCAT module.	
address:	Node address of the remote device on the network	
index:	CANopen Object index	
subindex:	CANopen Object sub-index	
Type:	1	Boolean
	2	Integer 8
	3	Integer 16
	4	Integer 32
	5	Unsigned 8
	6	Unsigned 16
	7	Unsigned 32
	9	Visible String (N/A as this is read only)
vr_number:	VR number between 0 and max **VR** where the result will be stored. (-1 if the next parameter contains the value to be written)	
value:	Optional data value for direct setting of the object	

EXAMPLES:

EXAMPLE 1:
Set the remote drive at EtherCAT address 3 to homing mode.

```
>>CO_WRITE(0, 3, $6060, 0, 2, -1, 6)
>>
```

EXAMPLE 2:
Set the remote drive proportional gain and velocity feed forward gain to the values placed in **VR**(21) and **VR**(22).

```
VR(21) = 2500
VR(22) = 1000
' both objects are unsigned 16 bit (data type 6)
CO_WRITE(0, 1, $60fb, 1, 6, 21)
CO_WRITE(0, 1, $60fb, 2, 6, 22)
```

Always refer to the manufacturer's user manual before writing to a CANopen object over EtherCAT.

CO_WRITE_AXIS

TYPE:
System Command

SYNTAX:
CO_WRITE_AXIS(axis_number, index, subindex, type, vr_number [,value])

DESCRIPTION:
This function sets a CANopen-over-EtherCAT object in the remote drive or IO device. The Object's index and sub-index are used to write a value to that object. The value can come from a **VR** or is put into the command directly if the **VR** number is set to -1.

Refer to the remote device's manual for a list of available objects. If the object value is set successfully, the command returns **TRUE**. (-1) Otherwise, in the case of an error while writing the value, the command returns **FALSE**.

PARAMETERS:

Axis_number:	Axis number of the EtherCAT drive.
index:	CANopen Object index

subindex:	CANopen Object sub-index	
Type:	1	Boolean
	2	Integer 8
	3	Integer 16
	4	Integer 32
	5	Unsigned 8
	6	Unsigned 16
	7	Unsigned 32
	9	Visible String (to terminal only)
vr_number:	VR number between 0 and max **VR** where the result will be stored. (-1 if the next parameter contains the value to be written)	
value:	Optional data value for direct setting of the object	

EXAMPLES:

EXAMPLE 1:

Write a value of 1 to a manufacturer specific object on servo drive at MC464 axis 3. CoE object 0x2802 sub-index 0x00, type 2 (8 bit integer). Get the **TRUE**/**FALSE** success indication and print it to the terminal.

```
>>?CO_WRITE_AXIS(3, $2802, 0, 2, -1, 1)
>>-1.0000
>>
```

EXAMPLE 2:

Write a position controller velocity feedforward gain value to the servo drive at MC464 axis 12. CoE object 0x60FB sub-index 0x02, type 6 (unsigned 16 bit integer).

```
VR(2010)=1000
‘ write the value from VR(2010)
error_flag = CO_WRITE_AXIS(12, $60fb, 2, 6, 2010)

IF error_flag = FALSE THEN
  PRINT “Error writing CANopen Object to Drive”
ENDIF
```

Always refer to the manufacturer’s user manual before writing to a CANopen object over EtherCAT.

: Colon

TYPE:
Special Character

DESCRIPTION:
The colon character is used as a label terminator and as a command separator.

LABEL TERMINATOR

SYNTAX:
`label:`

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

Labels can also be used to aid readability of code.

PARAMETERS:

Label	may be character strings of any length but only the first 32 characters are significant. Labels must be the first item on a line and should have no leading spaces.

EXAMPLE:
Use an **ON...GOTO** structure to assign a value into **VR** 10 depending on a local variable ‘attempts’.

```
ON attempts GOTO label1, label2, label3
GOTO continue

label1:
VR(10)=1
GOTO continue

Label2:
VR(10)=5
GOTO continue

Label3:
VR(10)=2
GOTO continue

continue:
```

COMMAND SEPERATOR

SYNTAX:

`statement: statement`

DESCRIPTION:

The colon is also used to separate TrioBASIC statements on a multi-statement line.

PARAMETERS:

Statement: any valid TrioBASIC statement. 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. Similarly with **GOSUB** because subroutine calls return to the following line.

EXAMPLES:

EXAMPLE 1:

Use of **GOTO** in the line means that any command following it will never be executed. This can be used as a debugging technique but usually happens due to a programming error.

```
PRINT "Hello":GOTO Routine:PRINT "Goodbye"
```

"Goodbye" will not be printed.

EXAMPLE 2:

Set the speed, a position in the table and execute a move all in one line.

```
SPEED=100:TABLE(10,123):MOVE(TABLE(10)
```

‘ Comment

TYPE:

Special Character

SYNTAX:

‘ text

DESCRIPTION:

A single ‘ is used to mark the start of a comment. A comment is a piece of text that is not compiled and just used to give the programmer information. It can be used at the start of a line or after a piece of code.

PARAMETERS:

text	Any notes that you wish to add to your program

EXAMPLE:
Using comments at the start of the program and in line to help document a program

```
'Motion program version 1.35
MOVE(100) 'Move to the start position
```

COMMSERROR

TYPE:
Reserved Keyword

COMMSPOSITION

TYPE:
Slot Parameter

DESCRIPTION:
Returns if the expansion module is on the top or the bottom bus.

VALUE:

-1	built in controller
1	module is on the top bus
0	module is on the bottom bus or no module fitted

COMMSTYPE

TYPE:
Slot Parameter (read only)

DESCRIPTION:
This parameter returns the type of communications daughter board in a controller slot.

VALUE:

Value	Communication type
0	Empty slot
32	SERCOS
37	Panasonic module
39	Sync encoder port
40	FlexAxis 4
41	FlexAxis 8
42	Ethercat module
43	SLM module
44	FlexAxis 8 SSI
62	Anybus module empty/ unrecognised
63	Anybus RS232
64	Anybus RS422
65	Anybus USB
66	Anybus Ethernet
67	Anybus Bluetooth
68	Anybus Zigbee
69	Anybus wireless LAN
70	Anybus RS485
71	Anybus Profibus
72	Anybus CC-Link
73	Anybus DeviceNet
74	Anybus Profinet 1 port
75	Anybus Profinet 2 port

EXAMPLE:

Check that the correct Anybus module is fitted before starting initialisation.

```
IF COMMSTYPE SLOT(3) = 71
```

```
    GOSUB initialise_profibus
  ELSE
    PRINT#5, “No Profibus compact com module detected”
  ENDIF
```

COMPILE

TYPE:

System Command

DESCRIPTION:

Forces compilation of the currently selected program. Program compilation is performed automatically by the system software prior to program RUN or when another program is SELECTed. This command is not therefore normally required.

SEE ALSO:

SELECT, COMPILE_ALL

COMPILE_ALL

TYPE:

System Command

DESCRIPTION:

Forces compilation of all programs. Program compilation is performed automatically by the system software prior to program RUN or when another program is SELECTed. This command is not therefore normally required.

SEE ALSO:

SELECT, COMPILE

COMPILE_MODE

TYPE:

Startup Parameter (**MC_CONFIG**)

DESCRIPTION:

COMPILE_MODE controls whether or not all used variables have to be defined within a DIM statement as a prerequisite before use or not.

The default setting (0) is the traditional compile mode where variables can be used without any need for declaration. However, by changing this parameter to 1, either within **MC_CONFIG** or at any time after startup, means that all new program compilations will require variables to be declared using DIM.

VALUE:

0	Local variables do not require explicit declaration (default)
1	Local variables require explicit declaration using DIM

EXAMPLES:

EXAMPLE 1:

COMPILE_MODE = 0 'No enforced variable declarations

EXAMPLE 2:

COMPILE_MODE = 1 'Force variable declarations via DIM

SEE ALSO:

DIM, COMPILE and COMPILE_ALL

CONNECT

TYPE:

Axis Command

SYNTAX:

CONNECT(ratio, driving_axis)

ALTERNATE FORMAT:

CO(...)

DESCRIPTION:

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

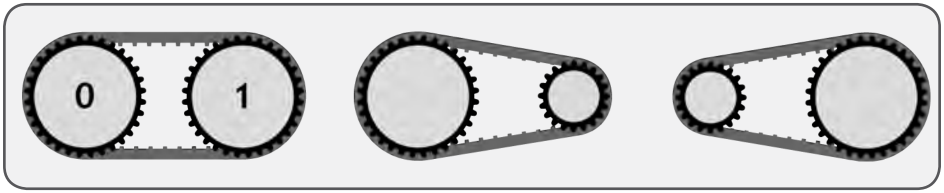
The ratio can be changed at any time by issuing another **CONNECT** command which will automatically update the ratio at **CLUTCH_RATE** without the previous **CONNECT** being cancelled. The command can be cancelled with a **CANCEL** or **RAPIDSTOP** command

You can prevent **CONNECT** from being canceled when a hardware or software limit is reached by setting the bit in **AXIS_MODE**. When this bit is set the ratio is temporarily set to zero while the limit is active so the axis will slow to a stop at the programmed **CLUTCH_RATE**.

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. The ratio is always specified as an encoder edge ratio.
driving_axis:	This parameter specifies the axis to link to.

As **CONNECT** uses encoder data it is not affected by *UNITS*, if you need to change the scale of your encoder feedback you should use *ENCODER_RATIO*



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

EXAMPLES:

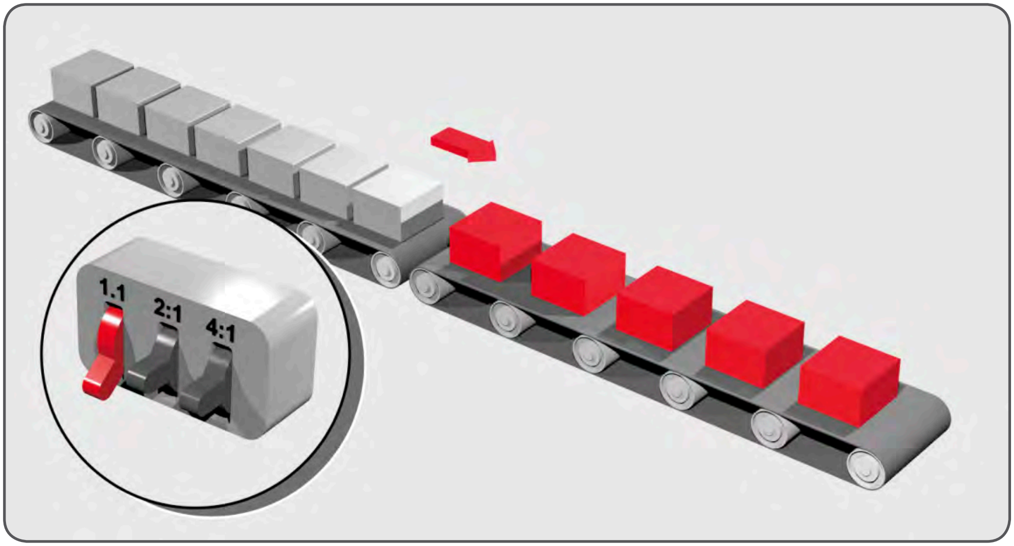
EXAMPLE 1:

In a press feed a roller is required to rotate at a speed one quarter of the measured rate from an encoder mounted on the incoming conveyor. The roller is wired to the master axis 0. The reference encoder is connected to axis 1.

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

EXAMPLE 2:

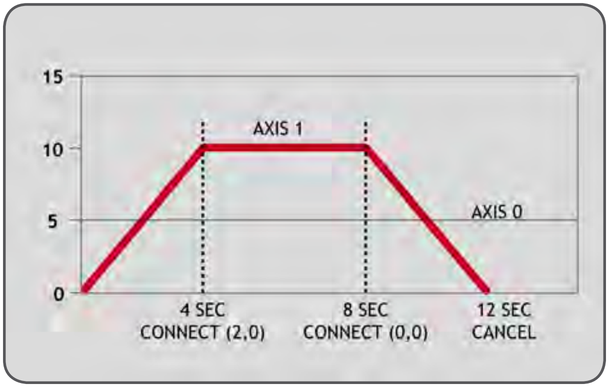
A machine has an automatic feed on axis 1 which must move at a set ratio to axis 0. This ratio is selected using inputs 0-2 to select a particular “gear”, this ratio can be updated every 100msec. Combinations of inputs will select intermediate gear ratios. For example 1 ON and 2 ON gives a ratio of 6:1.




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

EXAMPLE 3:

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




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

SEE ALSO:

AXIS_MODE, CLUTCH_RATE, ENCODER_RATIO

CONNPATH

TYPE:

Axis Command

SYNTAX:

CONNPATH(ratio , driving_axis)

DESCRIPTION:

Enables you to link to the path of an interpolated movement by linking the demand position of the base axis, to the interpolated path distance of the driving axis.

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

As **CONNPATH** uses encoder data it is not affected by **UNITS**, if you need to change the scale of your encoder feedback you should use **ENCODER_RATIO**

PARAMETERS:

ratio:	This is the ratio between the interpolated distance moved on the driving axis to the distance moved on the base axis.
driving_axis:	This parameter specifies the axis to link to.

EXAMPLES:

EXAMPLE 1:

A glue laying robot uses a screw feed for the adhesive, this needs to turn a quarter of a revolution for every unit of distance moved.

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

EXAMPLE 2:

It is required to move 156mm on axis 0 through an interpolated path distance of 100mm on axes 1,2 and 3. This is achieved by using virtual axis 4 as the path distance of the interpolated group and applying a **MOVELINK** from axis 0 to it. **SPEED** is initially set to zero so that the **MOVE** and **MOVELINK** start at the same time.

```
CONNPATH(1,1)AXIS(4)
a=100
b=100
c=100

BASE(1,2,3)
SPEED=0
MERGE=ON

MOVE(a,b,c)
WA(1)
MOVELINK(156,REMAIN AXIS(1),0,0,4)AXIS(0)
SPEED=10
```

SEE ALSO:

CLUTCH_RATE, ENCODER_RATIO

CONSTANT

TYPE:
System Command

SYNTAX:
`CONSTANT ["name"[, value]]`

DESCRIPTION:
Up to 1024 **CONSTANTS** can be declared in the controller, these are then available to all programs. They should be declared on startup and for fast startup the program declaring CONSTANTs should also be the **ONLY** process running at power-up.

Once a **CONSTANT** has been assigned it cannot be changed, even if you change the program that assigns it.

While developing you may wish to clear or change a **CONSTANT**. You can clear a single **CONSTANT** by using the first parameter alone. All **CONSTANT**s can be cleared by issuing **CONSTANT**. You can view all **CONSTANTS** using **LIST_GLOBAL**.

PARAMETERS:

name:	Any user-defined name containing lower case alpha, numerical or underscore (_) characters.
value:	The value assigned to the name.

EXAMPLES:

EXAMPLE 1:
Declare 2 CONSTANTs and use them within the program

```
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
```

EXAMPLE 2:
Use the command line to clear a defined constant

```
>>CONSTANT "NAK"
>>
```

EXAMPLE 3:
Use the command line to clear all defined constants

```
>>CONSTANT
>>
```

SEE ALSO:
GLOBAL, LIST_GLOBAL

CONTROL

TYPE:
System Parameter (Read Only)

DESCRIPTION:
The Control parameter returns the ID number of the *Motion Coordinator* in the system:

VALUE:

Value	Controller
400	MCSimulator
402	MC403Z
403	MC403
404	Euro404
405	MC405
408	Euro408
464	MC464

When the *Motion Coordinator* is **LOCKED**, 1000 is added to the above numbers. For example a locked MC464 will return 1464.

EXAMPLES:

EXAMPLE 1:
Checking the control value of a locked controller on the command line:

```
>>PRINT CONTROL
1464
>>
```

EXAMPLE 2:

Checking the controller type in a program, if it fails then stop the programs. :

```
IF CONTROL <> 464 THEN
  PRINT#terminal, "This program was designed to run a MC464"
  HALT
ENDIF
```

COORDINATOR_DATA

TYPE:

Reserved Keyword

COPY

TYPE:

System Command (command line only)

SYNTAX:

`COPY "program" "newprogram"`

DESCRIPTION:

Used to make a copy of an existing program in memory under a new name.

PARAMETERS:

program:	the name of the program to be copied
newprogram:	the name of the copy

EXAMPLE:

Make a backup of a program named motion

```
>>COPY "MOTION"  "MOTION_BACK"
Compiling MOTION
Linking MOTION
Pass=4
OK
>>
```

CORNER_MODE

TYPE:
Axis Parameter

DESCRIPTION:
Allows the program to control the cornering action.

Automatic corner speed control enables system to reduce the speed depending on **DECEL_ANGLE** and **STOP_ ANGLE**

The **CORNER_STATE** machine allows interaction with a TrioBASIC program and the loading of buffered moves depending on **RAISE_ANGLE**

Automatic radius speed control enables the system to reduce the speed depending on **FULL_SP_RADIUS**.

You can enable any combination of the speed control bits.

VALUE:
16bit value, each bit represents a different corner mode.

Bit	Description	Value
0	Reserved	1
1	Automatic corner speed control	2
2	Enable the **CORNER_STATE** machine	4
3	Automatic radius speed control	8

EXAMPLE:
Enable the corner state machine and automatic corner speed control.

```
CORNER_MODE= 2+4
```

SEE ALSO:
CORNER_STATE, DECEL_ANGLE, FULL_SP_RADIUS, RAISE_ANGLE, STOP_ANGLE

CORNER_STATE

TYPE:
Axis Parameter

DESCRIPTION:

Allows a **BASIC** program to interact with the move loading process.

This can be used to facilitate tool adjustment such as knife rotation at sharp corners.

This parameter is only active when **CORNER_STATE** bit 2 is set. It is also required to use bit 1 of **CORNER_STATE** with **STOP_ANGLE** set to less than or equal to **RAISE_ANGLE** to stop the motion.

VALUE:

0	Load move and ramp up speed
1	Ready to load move, stopped
3	Load move

EXAMPLE:

When a transition exceeds **RAISE_ANGLE** it is required to lift a cutting knife and rotate it to a new position. The following process is required:

1. System sets **CORNER_STATE** to 1 to indicate move ready to be loaded with large angle change.
2. BASIC program raises knife.
3. BASIC program sets **CORNER_STATE** to 3.
4. System will load following move but with speed overridden to zero. This allows the direction to be obtained from **TANG_DIRECTION**.
5. BASIC program orients knife possibly using **MOVETANG**.
6. BASIC program clears **CORNER_STATE** to 0.
7. System will ramp up speed to perform the next move.

```
MOVEABSSP(x,y)
IF CHANGE_DIR_LAST>RAISE_ANGLE THEN
  WAIT UNTIL CORNER_STATE>0
  ‘Raise Knife
  MOVE(100) AXIS(z)
  CORNER_STATE=3
  WA(10)
  WAIT UNTIL VP_SPEED AXIS(2)=0
  ‘Rotate Knife
  MOVETANG(0,x) AXIS(r)
  ‘Lower Knife
  MOVE(-100) AXIS(z)
  ‘Resume motion
  CORNER_STATE=0
```

```
    ENDIF
```

SEE ALSO:
CORNER_MODE, RAISE_ANGLE, STOP_ANGLE

COS

TYPE:
Mathematical Function

SYNTAX:
```
value = COS(expression)
```

DESCRIPTION:
Returns the **COSINE** of an expression. Input values are in radians.

PARAMETERS:

value:	The **COSINE** of the expression
expression:	Any valid TrioBASIC expression.

EXAMPLE:
Print the cosine of zero to the command line with 3 decimal places
```
>>PRINT COS(0)[3]
1.000
```

CPU_EXCEPTIONS

TYPE:
Reserved Keyword

CRC16

TYPE:
Mathematical Command

SYNTAX:

```
result = CRC16(mode,{parameters})
```

DESCRIPTION:

Calculates a 16 bit Cyclic Redundancy Check (CRC) of data stored in contiguous Table Memory or **VR** Memory locations.

PARAMETERS:

mode:	0	Initialise the polynomial
	1	Calculate the CRC

MODE = 0:

SYNTAX:

```
result = CRC16(0, poly)
```

DESCRIPTION:

Initialises the command with the Polynomial

PARAMETERS:

result:	Always returns -1
poly:	Polynomial used as seed for CRC check range 0-65535 (or 0-$**FFFF**)

MODE = 1:

SYNTAX:

```
result = CRC16(1, source, start, end, initial)
```

DESCRIPTION:

Calculates the CRC

PARAMETERS:

result:	Returns the result of the CRC calculation. Will be 0 if the calculation fails.

source:	Defines where the data is loaded	
	0	Table Memory
	1	**VR** Memory
start:	Start location of first byte	
end:	End Location of last byte	
initial:	Initial CRC value. Normally $0 - $**FFFF**	

EXAMPLES:

EXAMLPE 1:

Calculate the CRC using Table Memory:

```
poly = $8005
CRC16(0, poly) ‘Initialise internal CRC table memory

TABLE(0,1,2,3,4,5,6,7,8) *load data into TABLE memory location 0-7
reginit = 0
calc_crc = CRC16(1,0,0,7,reginit) ‘Source Data=TABLE(0..7)
```

EXAMPLE 2:

Calculate the CRC using **VR**s:

```
‘ generate CRC lookup table
poly=$8005
CRC16(0,poly)

‘ create test data as “hello”
VR(100)=104
VR(101)=101
VR(102)=108
VR(103)=108
VR(104)=111
VR(105)=0
VR(106)=0
PRINT VRSTRING(100)

‘ calculate the crc16
crc=0
crc=CRC16(1,1,100,104,crc)

‘ print the result
PRINT HEX(crc)
```

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.

VALUE:
Any positive value in user **UNITS**

EXAMPLE:
Set up the **CREEP** speeds on 2 axes and then perform a **DATUM** routine.

```
BASE(2)
CREEP=10
SPEED=500
DATUM(4)
CREEP AXIS(1)=10
SPEED AXIS(1)=500
DATUM(4) AXIS(1)
```

SEE ALSO:
DATUM

D_GAIN

TYPE:
Axis Parameter

DESCRIPTION:
Used as part of the closed loop control, adding derivative gain to a system is likely to produce a smoother response and allow the use of a higher proportional gain than could otherwise be used.

High values may lead to oscillation. For a derivative term K_d and a change in following error de the contribution to the output O_d signal is:

$$O_d = K_d \times \delta_e$$

VALUE:
The derivative gain is a constant which is multiplied by the change in following error. Default value = 0

EXAMPLE:
Setting the gain values as part of a **STARTUP** program

```
P_GAIN=1
I_GAIN=0
D_GAIN=0.25
OV_GAIN=0
…
```

D_ZONE_MAX

TYPE:
Axis Parameter

DESCRIPTION:
Working in conjunction with **D_ZONE_MIN**, **D_ZONE_MAX** defines a DAC dead band. This clamps the DAC output to zero when the demand movement is complete and the magnitude of the following error is less than the **D_ZONE_MIN** value. The servo loop will be reactivated when either the following error rises above the **D_ZONE_MAX** value, or a fresh movement is started.

This can be used to prevent oscillations at static positions in Piezo systems.

VALUE:
Above this value the servo loop is reactivated when clamped in the dead band.

EXAMPLE:

The DAC output will be clamped at zero when the movement is complete and the following error falls below 3. When a movement is restarted or if the following error rises above a value of 10, the servo loop will be reactivated

```
D_ZONE_MIN = 3
D_ZONE_MAX = 10
```

SEE ALSO:

D_ZONE_MIN

D_ZONE_MIN

TYPE:

Axis Parameter

DESCRIPTION:

Working in conjunction with **D_ZONE_MAX**, **D_ZONE_MIN** defines a DAC dead band. This clamps the DAC output to zero when the demand movement is complete and the magnitude of the following error is less than the **D_ZONE_MIN** value. The servo loop will be reactivated when either the following error rises above the **D_ZONE_MAX** value, or a fresh movement is started.

This can be used to prevent oscillations at static positions in Piezo systems.

VALUE:

When the axis is **IDLE** and the magnitude of the following error is less than this value the DAC is clamped to zero.

EXAMPLE:

The DAC output will be clamped at zero when the movement is complete and the following error falls below 3. When a movement is restarted or if the following error rises above a value of 10, the servo loop will be reactivated

```
D_ZONE_MIN = 3
D_ZONE_MAX = 10
```

SEE ALSO:

D_ZONE_MAX

DAC

TYPE:
Axis Parameter

DESCRIPTION:
Writing to this parameter when **SERVO** = OFF and **AXIS_ENABLE** = ON allows the user to force a demand value for that axis. On an analogue axis this will set a voltage on the output. On a digital axis this will be the demand value.

When using a FlexAxis as a stepper or encoder output or anytime with **SERVO** = **OFF** the voltage outputs are available for user control.

The **WDOG** and **AXIS_ENABLE** must be ON for the demand value to be set. When the **WDOG** or **AXIS_ENABLE** is OFF you can write a value to DAC but the actual output (**DAC_OUT**) will be at 0.

VALUE:
The demand value for the axis

For a 12 bit DAC on an analogue axis:

DAC	Voltage
-2048	10V
2047	-10V

For a 16 bit DAC on an analogue axis:

DAC	Voltage
32767	10V
-32768	-10V

For digital axes check the drive specification for suitable values.

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
```

SEE ALSO:
DAC_OUT, DAC_SCALE, SERVO

DAC_OUT

TYPE:
Axis Parameter (Read Only)

DESCRIPTION:
DAC_OUT reads the demand value for the axis.

In an analogue system this will be the value sent to the voltage output (the DAC). If **SERVO** = ON this is the output of the closed loop algorithm. If **SERVO** = OFF it is the value set by the user in DAC

In a digital system it returns the demand value for the axis which could be the actual position, speed or torque depending on the axis **ATYPE**.

VALUE:
Demand value for the axis

EXAMPLE:
To check that the controller has set the correct voltage for axis 8 on an analogue system read **DAC_OUT** in the command line.

```
>>PRINT DAC_OUT AXIS(8)
288.0000
>>
```

SEE ALSO:
DAC, DAC_SCALE, ATYPE

DAC_SCALE

TYPE:
Axis Parameter

DESCRIPTION:
DAC_SCALE is an integer that is multiplied to the output of the closed loop algorithm. You can use it to

reverse the polarity of the demand value or to scale it so to effectively reduce the resolution of the closed loop algorithm.

As it is applied to the output of the closed loop algorithm it is not applied to position based axis.

VALUE:

Can be a positive or negative integer. The default values are shown in the following table:

MC464 Ethercat	1
MC464 Sercos	1
MC464 FlexAxis	16
MC464 Panasonic	16
MC464 SLM	16
MC405	1
MC403	1

To obtain the highest possible resolution of your system **DAC_SCALE** should be set to 1 or -1.

To avoid problems with the multiply by 16, **DAC_SCALE** should be set to 1 for an **SLM** axis

EXAMPLE:

EXAMPLE 1:

The FlexAxis uses a 16bit DAC. To make it compatible with the gain settings used on older 12 bit DACs, **DAC_SCALE** is set to 16.

The max output from closed loop algorithm is 2048 (for a 12bit system)

The max output from a 16bit DAC is 32768 which is 2048 multiplied by 16

EXAMPLE 2:

Set up an axis to work in the reverse direction. For a servo axis, both the **DAC_SCALE** and the **ENCODER_RATIO** must be set to minus values.

```
BASE(2) ‘ set axis 2 to work in reverse direction
DAC_SCALE = -1
ENCODER_RATIO(-1,1)
```

SEE ALSO:

DAC, DAC_OUT, ENCODER_RATIO

DATE$

TYPE:
String Function

SYNTAX:
DATE$

DESCRIPTION:
DATE$ is used as part of a **PRINT** statement or a **STRING** variable to write the current date from the real time clock. The date is printed in the format DD/MMM/**YYYY**. The month is displayed in short text form.

The **DATE$** is set through the **DATE** command

PARAMETERS:
None.

EXAMPLES:

EXAMPLE 1:
This will print the date in format for example 20th October 2010 will print the value: 20/Oct/2010

```
PRINT #5,DATE$
```

EXAMPLE 2:
Create an error message to print later in the program

```
DIM string1 AS STRING(30)
string1 = "Error occurred on the " + DATE$
```

DATE

TYPE:
System Function

DESCRIPTION:
Returns or sets the current date held by the real time clock.

SETTING THE DATE:

SYNTAX:
DATE=dd:mm:yy

DESCRIPTION:
Sets the date using the two digit year format or the four digit year format.

PARAMETERS:

dd:	day in two digit numeric format
mm:	Month in two digit numeric format
yy:	last two digits of the year using the range 00-99 representing 2000-2099 OR the full four digits of the year using the range 2000-2099

Years outside the range 2000-2099 are invalid.

EXAMPLE:
Set the date to the 20th October 2012

```
>>DATE=20:10:12
```

or

```
>>DATE=20:10:2012
```

READING THE DATE:

SYNTAX:

```
Value = DATE({mode})
```

DESCRIPTION:
Read the date value from the real time clock as a number.

PARAMETERS:

mode	**value**
none	The number of days since 01/01/2000 (with 01/01/2000 = 0)
0	The day of the current month
1	The month of the current year
2	The current year

EXAMPLES:

EXAMPLE 1:

Print the number of days since 1st January 2000 (with the 1st being day 0)

```
>>PRINT DATE
4676
>>
```

EXAMPLE 2:

Set a date then print it out using the US format

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

DATUM

TYPE:

Axis Command

SYNTAX:

`DATUM(sequence)`

DESCRIPTION:

Performs one of 6 datuming sequences to locate an axis to an absolute position. The creep speed used in the sequences is set using **CREEP**. The programmed speed is set with the **SPEED** command.

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

PARAMETER:

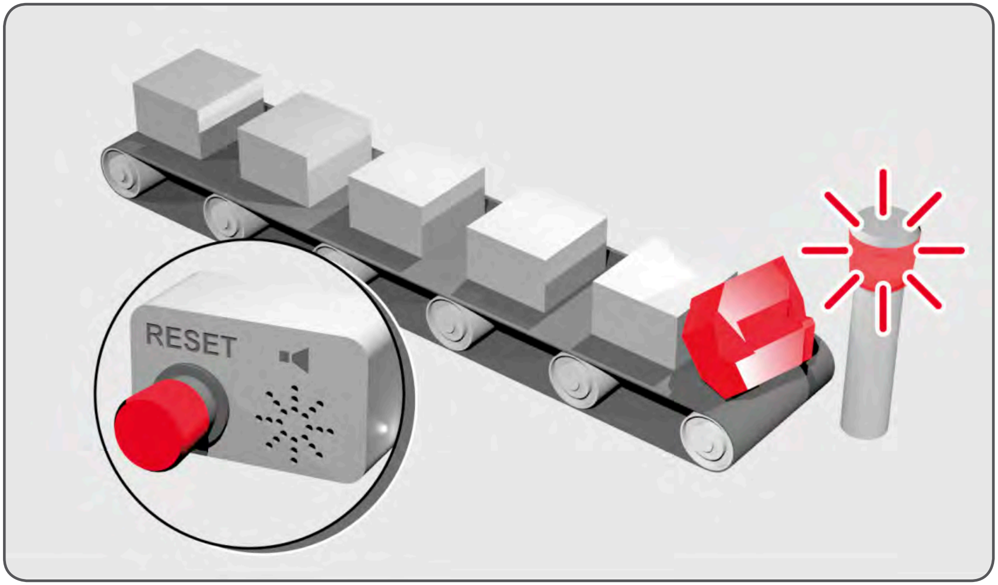
Sequence	Description
0	**DATUM**(0) clears the following error exceeded **FE_LIMIT** condition for ALL axes by setting these bits in **AXISSTATUS** to zero: **BIT 1** Following Error Warning **BIT 2** Remote Drive Comms Error **BIT 3** Remote Drive Error **BIT 8** Following Error Limit Exceeded **BIT 11** Cancelling Move
1	The axis moves at creep speed forward till the Z marker is encountered. The Measured position is then reset to zero and the Demand 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 Measured position is then reset to zero and the Demand 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 Measured position is then reset to zero and the Demand 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 Measured position is then reset to zero and the Demand position corrected so as to maintain the following error.
5	The axis moves at programmed speed forward until the datum switch is reached. The axis then reverses at creep speed until the datum switch is reset. It then continues in reverse at creep speed looking for the Z marker on the motor. The Measured position where the Z input was seen is then set to zero and the Demand position corrected so as to maintain the following error.
6	The axis moves at programmed speed reverse until the datum switch is reached. The axis then moves forward at creep speed until the datum switch is reset. It then continues forward at creep speed looking for the Z marker on the motor. The Measured position where the Z input was seen is then set to zero and the Demand position corrected so as to maintain the following error.
7	Clear **AXISSTATUS** error bits for the **BASE** axis only. Otherwise the action is the same as **DATUM**(0).

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

EXAMPLES:

EXAMPLE 1:

A production line is forced to stop if something jams the product belt, this causes a motion error. The obstacle has to be removed, then a reset switch is pressed to restart the line.

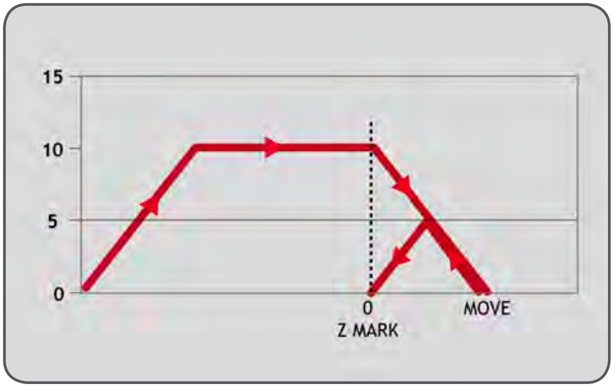



```
FORWARD                 'start production line
WHILE IN(2)=ON
  IF MOTION_ERROR=0 THEN
     OP(8,ON)           'green light on; line is in motion
        ELSE
     OP(8, OFF)
    GOSUB error_correct
  ENDIF
WEND
CANCEL
STOP

error_correct:
   REPEAT
    OP(10,ON)
    WA(250)
    OP(10,OFF)          'flash red light to show crash
    WA(250)
```

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

EXAMPLE 2:

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




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

EXAMPLE 3:

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


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

EXAMPLE 4:

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

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

SEE ALSO:
CREEP, DATUM_IN

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 used for **DATUM_IN** is active low.

VALUE:

-1	disable the input as **DATUM_IN** (default)
0-IO_Max	Input to use as datum input

Any type of input can be used, built in, Trio CAN I/O, CANopen, EtherCAT or virtual.

EXAMPLE:
Set input 28 as the **DATUM** input for axis 0 then perform a homing routine

```
DATUM_IN AXIS(0)=28
DATUM(3)
```

SEE ALSO:
DATUM

DAY$

TYPE:
String Function

SYNTAX:
DAY$

DESCRIPTION:
Used as part of a **PRINT** statement or a **STRING** variable to write the current day as a string.

The **DAY**$ is set through the **DATE** command

EXAMPLES:

EXAMPLE 1:
Print the day as part of a welcome message:

```
PRINT#5, “Welcome to Trio on “; DAY$
```

EXAMPLE 2:
Create a header to be used when writing a log to the SD card.

```
DIM header AS STRING(30)
header = DAY$ + “Start of production”
```

SEE ALSO:
DATE, DATE$, DAY, PRINT, STRING

DAY

TYPE:
System Function

SYNTAX:
value = DAY

DESCRIPTION:
Returns the current day as a number.

The **DAY** is set through the **DATE** command

RETURN VALUE:
0..6, Sunday is 0

EXAMPLE:
Print some text depending on the day
```
IF DAY=2 THEN
    PRINT#5, “Change filter”
ENDIF
```

SEE ALSO:
DATE, DAY$

DECEL

TYPE:
Axis Parameter

DESCRIPTION:
The **DECEL** axis parameter may be used to set or read back the deceleration rate of each axis fitted.

VALUE:
The deceleration rate in **UNITS**/sec/sec. Must be a positive value.

EXAMPLE:
Set the deceleration parameter and print it to the user.
```
DECEL=100’ Set deceleration rate
PRINT “ Decel is “;DECEL;” mm/sec/sec”
```

SEE ALSO:
ACCEL

DECEL_ANGLE

TYPE:
Axis Parameter

DESCRIPTION:
This parameter is used with **CORNER_MODE**, it defines the maximum change in direction of a 2 axis

interpolated move that will be merged at full speed. When the change in direction is greater than this angle the speed will be proportionally reduced so that:

VP_SPEED=**FORCE_SPEED** * (angle – **DECEL_ANGLE**) / (**STOP_ANGLE** – **DECEL_ANGLE**)

Where angle is the change in direction of the moves.

VALUE:

The angle to start to reduce the speed, in radians.

EXAMPLE:

Decelerate to a slower speed when the transition is between 15 and 45 degrees.

```
CORNER_MODE=2
DECEL_ANGLE = 15 * (PI/180)
STOP_ANGLE = 45 * (PI/180)
```

SEE ALSO:

CORNER_MODE, STOP_ANGLE

DEFPOS

TYPE:

Axis Command

SYNTAX:

DEFPOS(pos1 [,pos2[, pos3[, pos4...]]])

ALTERNATE FORMAT:

DP(pos1 [,pos2[, pos3[, pos4...]]])

DESCRIPTION:

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

PARAMETERS:

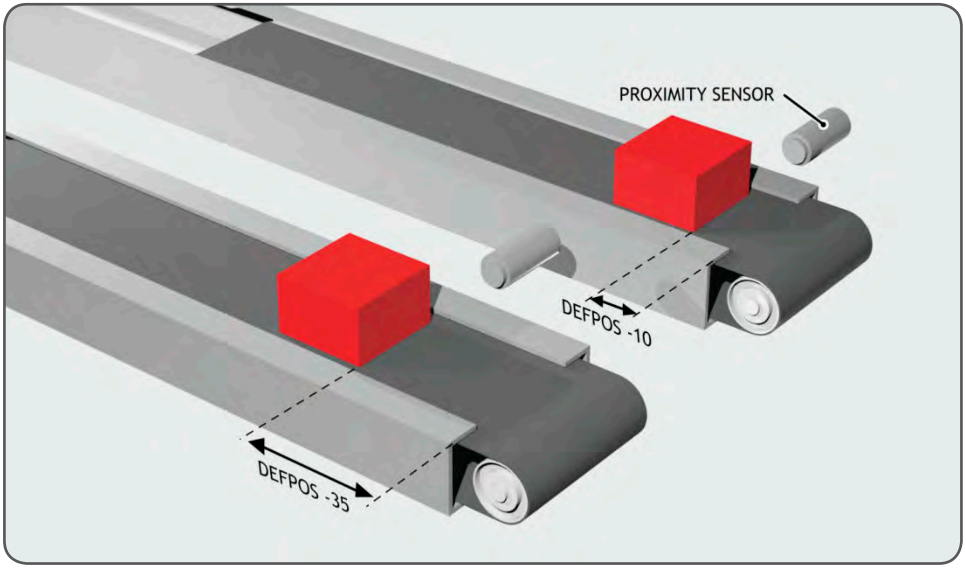
pos1:	Absolute position to set on current base axis in user units.
pos2:	Abs. position to set on the next axis in **BASE** array in user units.
pos3:	Abs. position to set on the next axis in **BASE** array in user units.

...	

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

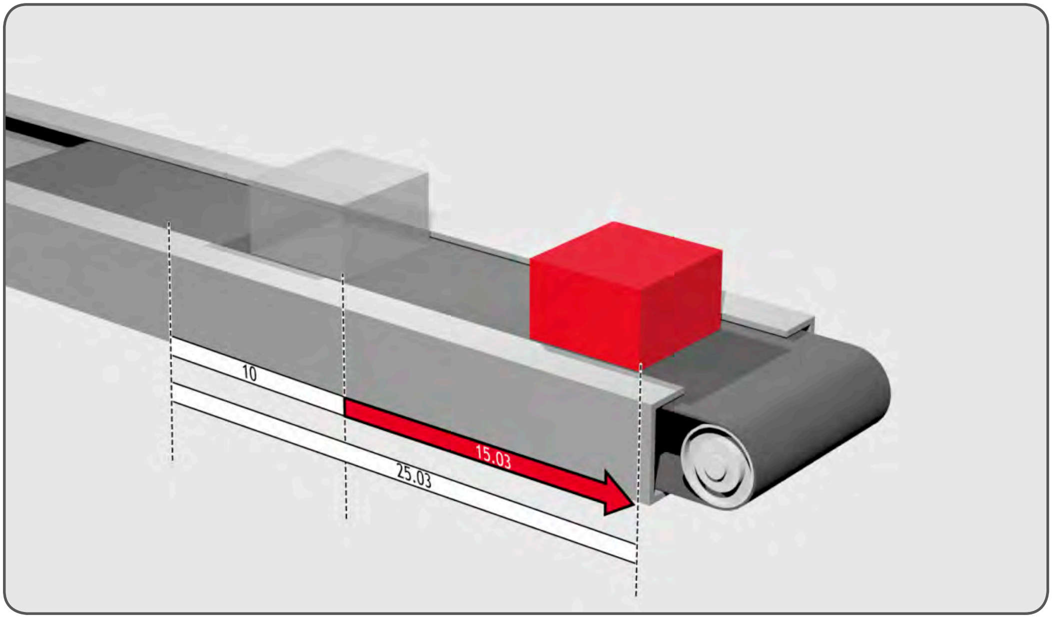
EXAMPLES:

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




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

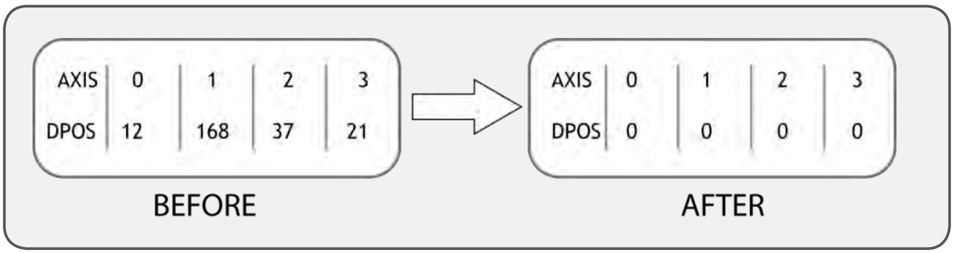
EXAMPLE 2:
Define the axis position to be 10, then start an absolute move, but make sure the axis has updated the position before loading the **MOVEABS**.




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

EXAMPLE 3:

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




```
>>BASE(0)
>>DEFPOS(0,0,0,0)
>>
```

SEE ALSO:

OFFPOS

DEL

TYPE:
System Command

SYNTAX:
DEL "program"

ALTERNATE FORMAT:
RM "program"

DESCRIPTION:
Used to delete a program form the controller memory.

This command should not be used from within *Motion* Perfect.

PARAMETERS:

program:	the name of the program to be deleted

EXAMPLE:
Delete an old program

```
>>DEL "oldprog"
OK
>>
```

DEMAND_EDGES

TYPE:
Axis Parameter (Read Only)

DESCRIPTION:
Allows the user to read back the current **DPOS** in encoder edges.

You can use **DEMAND_EDGES** to check that your **UNITS** or **ENCODER_RATIO** values are set correctly.

VALUE:
Demand position in encoder edges.

EXAMPLE:
Print the **DEMAND_EDGES** in the command line

```
>>PRINT DEMAND_EDGES AXIS(4)
523
>>
```

DEMAND_SPEED

TYPE:
Axis Parameter (Read Only)

DESCRIPTION:
Returns the speed output of the VPU, this is normally used for low level debug of the motion system.

VALUE:
VPU speed output in user **UNITS** per servo period.

EXAMPLE:
Check the VPU speed output using the command line

```
>>?DEMAND_SPEED
5.0000
>>
```

DEVICENET

TYPE:
System Command

SYNTAX:
DEVICENET(slot, function[,parameters…])

DESCRIPTION:
The command **DEVICENET** is used to start and stop the DeviceNet slave function which is built into the *Motion Coordinator*.

Polled IO data is transferred periodically:

From PLC to [**TABLE**(poll_base) -> **TABLE**(poll_base + poll_in)]
To PLC from [**TABLE**(poll_base + poll_in + 1) -> **TABLE**(poll_base + poll_in + poll_out)]

PARAMETERS:

slot:	Set -1 for built-in CAN port	
function:	0	Start the DeviceNet slave protocol on the given slot.
	1	Stop the DeviceNet protocol.
	2	Put startup baudrate into Flash EPROM

FUNCTION = 0:

SYNTAX:
`DEVICENET(slot, 0, baud, mac_id, poll_base, poll_in, poll_out)`

DESCRIPTION:
Start the DeviceNet protocol using the specified parameters

PARAMETERS:

baud:	Set to 125, 250 or 500 to specify the baud rate 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 transferred as poll data
poll_in:	Number of words to be received during poll. Range 0..4
poll_out:	Number of words to be sent during poll. Range 0..4

FUNCTION = 1:

SYNTAX:
`DEVICENET(slot, 1)`

DESCRIPTION:
Stop the DeviceNet protocol from running

FUNCTION = 2:

SYNTAX:

```
DEVICENET(slot, 2, baud)
```

DESCRIPTION:

Store the baud rate in flash EPROM for power up.

PARAMETERS:

baud:	Set to 125, 250 or 500 to specify the baud rate in kHz.

EXAMPLES:

EXAMPLE 1:

Start the DeviceNet protocol on the built-in CAN port

```
DEVICENET(-1,0,500,30,0,4,2)
```

EXAMPLE 2:

Stop the DeviceNet protocol on the CAN board in slot 2;

```
DEVICENET(2,1)
```

EXAMPLE 3:

Set the CAN board in slot 0 to have a baud rate of 125k bps on power-up;

```
DEVICENET(0,2,125)
```

DIM.. AS.. BOOLEAN/ FLOAT/ INTEGER/STRING

TYPE:

Declaration

SYNTAX:

```
DIM name AS type
DIM name AS FLOAT [(length)]
DIM name AS INTEGER [(length)]
DIM name AS STRING(length)
```

DESCRIPTION

By default local variables are type **FLOAT** and do not require declaration. It is possible to declare other types of values using the DIM declaration. **BOOLEAN**, **FLOAT**, **INTEGER** and **STRING** can be declared. It is also possible to make arrays of numerical types.

If **COMPILE_MODE** =1 then all local variables must be declared.

Local variables can be declared in an **INCLUDE** file.

TYPES:

BOOLEAN	1bit binary value (**TRUE** or **FALSE**)
FLOAT	64bit floating point number (default)
INTEGER	64bit signed integer value
STRING	**ASCII** text

TYPE = BOOLEAN:

SYNTAX:

DIM name AS BOOLEAN[(size [,size [,size]])]

DESCRIPTION:

Declare a variable as a **BOOLEAN** value. This can be used with **TRUE** and **FALSE**, any non-zero value written to a **BOOLEAN** variable will set its state to **TRUE**.

PARAMETERS:

name:	Any user-defined name containing lower case alpha, numerical or underscore (_) characters.
size:	The size of the array of **BOOLEAN**, up to 3 dimensions.

The size must be a number. You cannot use local variables, **VR** etc to set this value.

EXAMPLES:

Use a local variable as a flag to track the ok status of a machine.

```
DIM machine_ok AS BOOLEAN

machine_ok = TRUE

WHILE machine_ok = TRUE
  IF MOTION_ERROR <> 0 AND IN(0) = TRUE THEN
    machine_ok =FALSE
  ENDIF
WEND
```

TYPE = FLOAT:

SYNTAX:

```
DIM name AS FLOAT[(size [,size [,size]])]
```

DESCRIPTION:

Declare a variable as a floating point value.

PARAMETERS:

name:	Any user-defined name containing lower case alpha, numerical or underscore (_) characters.
size:	The size of the array of **FLOAT**, up to 3 dimensions.

The size must be a number. You cannot use local variables, **VR** etc to set this value.

EXAMPLES:

Use an array of positions to run a sequence of moves.

```
DIM position AS FLOAT(10)

position(0) = 0
position(1) = 10.3214
position(2) = 15.123
position(3) = 20.77569
position(4) = 25.2215
position(5) = 22.37895
position(6) = 21.7897
position(7) = 20.1457
position(8) = 15.4457
position(9) = 0

FOR x = 0 TO 9
 MOVEABS(position(x))
NEXT x
```

TYPE = INTEGER:

SYNTAX:

```
DIM name AS INTEGER[(size [,size [,size]])]
```

DESCRIPTION:

Declare a variable as an integer value. If a floating point number is assigned to an integer variable then the decimal part is truncated.

PARAMETERS:

name:	Any user-defined name containing lower case alpha, numerical or underscore (_) characters.
size:	The size of the array of **INTEGER**, up to 3 dimensions.

The size must be a number. You cannot use local variables, **VR** etc to set this value.

EXAMPLES:

Declare a local variable as an integer to use when reading in characters from the serial port.

```
DIM character AS INTEGER
DIM message AS STRING(200)

WHILE KEY#1
  GET#1, character
  message = message + CHR(character)
WEND
```

TYPE = STRING:

SYNTAX:

`DIM name AS STRING(length)`

DESCRIPTION:

Declare a variable as a string so that you can use it in **PRINT** statements, part of a logical condition or anywhere in the TrioBASIC that uses text. The variable can be assigned by any function or parameter that generates a string or manually.

You can use the **STR** function to change a numerical value to a string.

PARAMETERS:

name:	Any user-defined name containing lower case alpha, numerical or underscore (_) characters.
length:	Maximum number of characters that the variable can hold

The length must be a number. You cannot use local variables, **VR** etc to set this value.

EXAMPLES:

EXAMPLE 1:

Pre-define a set of error strings to use later:

```
DIM error1 AS STRING(20)
error1 = “Feed jammed”
DIM error2 AS STRING(20)
error2 = “Cutter jammed”
DIM error3 AS STRING(20)
error3 = “Out of material”

display_error:
IF error_number = 1 then
  PRINT error1
ELSEIF error_number = 2 then
  PRINT error2
ELSE
  PRINT error3
ENDIF
```

EXAMPLE 2:

Read in characters from a channel and append them to a string variable then finally printing them.

```
DIM captured_text AS STRING(50)
WHILE char<>13 OR count>50
  TICKS=10000 ‘5 second timeout on character
  WAIT UNTIL KEY#5 OR TICKS<0
  IF TICKS<0 THEN
    count=100 ‘exit loop
  ELSE
    GET#5,char
    captured_text = captured_text + CHR(char)
    count=count+1
  ENDIF
WEND
PRINT captured_text
```

EXAMPLE 3:

Using a string variable decide which motion routine to execute:

```
IF g_value = “G00” THEN ‘ rapid positioning
  SPEED = fast_speed
  MOVE(x,y,z)
  WAIT IDLE
  SPEED = standard_speed
ELSEIF g_value = “G01” THEN ‘ linear move
```

```
    MOVE(x,y,z)
  ELSEIF g_value = "G02" THEN ' anticlockwise circular move
    MOVECIRC(x,y,x+i_value,y+j_value,0)
  ELSEIF g_value = "G03" THEN ' clockwise circular move
    MOVECIRC(x,y,x+i_value,y+j_value,1)
  ELSE
    PRINT "Ignoring unsupported token: ";g_value
  ENDIF
```

SEE ALSO:

CHR, COMPILE_MODE, HEX, DATE$, DAY$, TIME$

DIR

TYPE:

System Command (command line only)

SYNTAX:

DIR [option]

ALTERNATE FORMAT:

LS [option]

DESCRIPTION:

Prints a list of all programs including their size and **RUNTYPE**.

PARAMETERS:

Parameter	Function
none	Directory listing of controller memory
d	Directory listing of SD card memory
s	Reserved function
x	Extended listing of controller memory (used by *Motion* Perfect).

DISABLE_GROUP

TYPE:
System Command

SYNTAX:
DISABLE_GROUP(parameter[,parameters…])

DESCRIPTION:
Used to create a group of axes which will be disabled if there is a motion error in one or more of the group. After the group is created, when an error occurs all the axes in the group will have their **AXIS_ENABLE** set to OFF and **SERVO** set to OFF.

Multiple groups can be made, although one axis cannot belong to more than one group.

Only axes that have individual enables should be used in a disable group. Such as Digital drives and Steppers.

DISABLE_GROUP(-1)

SYNTAX:
DISABLE_GROUP(-1)

DESCRIPTION:
Clears all groups

DISABLE_GROUP(AXIS1...)

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

DESCRIPTION:
Assigns the listed axis to a group

PARAMETERS:

axis1:	Axis number of first axis in group
axis2:	Axis number of second axis in group.
axisN:	Axis number of Nth axis in group.

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

EXAMPLES:

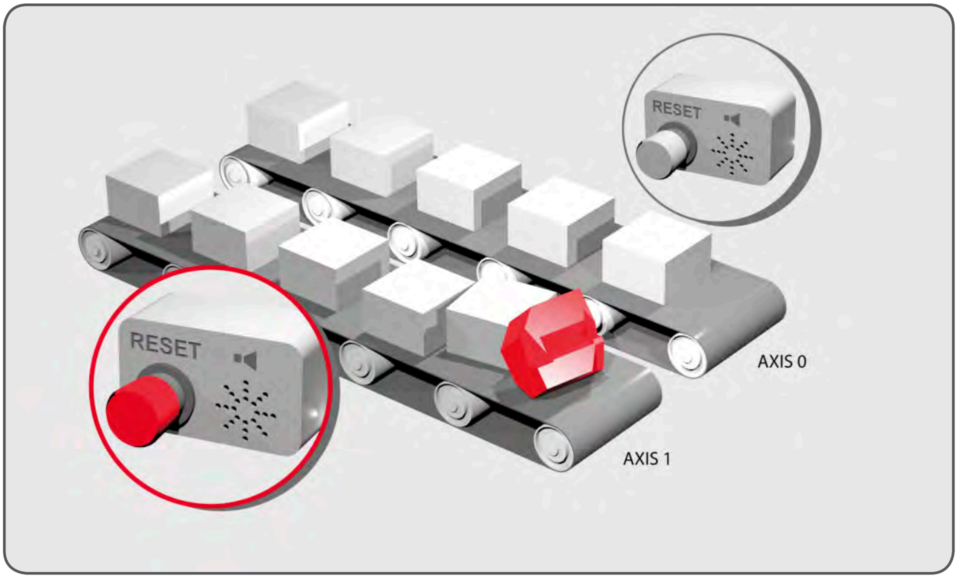
EXAMPLE 1:

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

```
DISABLE_GROUP(-1)       'remove any previous axis groupings
DISABLE_GROUP(0,1,2,6)  'group axes 0 to 2 and 6
DISABLE_GROUP(3,4,5,7)  'group axes 3 to 5 and 7
WDOG=ON 'turn on the enable relay and the remote drive enable
FOR ax=0 TO 7
  AXIS_ENABLE AXIS(ax)=ON  'enable the 8 axes
  SERVO AXIS(ax)=ON 'start position loop servo for each axis
NEXT ax
```

EXAMPLE 2:

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




```
DISABLE_GROUP(0) 'put axis 0 in its own group
DISABLE_GROUP(1) 'put axis 1 in another group
GOSUB group_enable0
```

```
GOSUB group_enable1
WDOG=ON
FORWARD AXIS(0)
FORWARD AXIS(1)

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

group_enable0:
  BASE(0)
  DATUM(7) ‘ clear motion error on axis 0
  WA(10)
  AXIS_ENABLE=ON
  SERVO=ON
RETURN
group_enable1:
  BASE(1)
  DATUM(7) ‘ clear motion error on axis 0
  WA(10)
  AXIS_ENABLE=ON
  SERVO=ON
RETURN
```

EXAMPLE 3:

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

```
DISABLE_GROUP(-1)      'remove any previous axis groupings
DISABLE_GROUP(0,1,2)   'group axes 0 to 2
GOSUB group_enable     'enable the axes and clear errors
WDOG=ON
SPEED=1000
FORWARD

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

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

SEE ALSO:

AXIS_ENABLE, SERVO

DISPLAY

TYPE:

System Parameter

DESCRIPTION:

Determines which group of the I/O channels are to be displayed on the LCD or LED bank.

VALUE:

Controller with an LCD use the following values in **DISPLAY**

Bits 16 - 31	Bits 0 - 15	Description
	0	Inputs 0-15 (default value)
	1	Inputs 16-31
	2	Outputs 0-15 (0-7 unused on existing controllers)
	3	Outputs 16-31
1		User control of the LCD segments *
	888	Reserved value

* MC405 only. When bit 16 is set, user control of the 3x7 segment characters is enabled. By default this is disabled.

Controller with an LED display use the following values in **DISPLAY**

Bits 0 - 15	Description
0	Inputs 0-7 (default value)
1	Inputs 7-15
2	Inputs 16-23
3	Inputs 24-31
4	Outputs 0-7 (0-7 unused on existing controllers)
5	Outputs 8-15
6	Outputs 16-23
7	Outputs 24-31

EXAMPLE 1:

Show outputs 16-31 on the MC464

```
>>DISPLAY=3
>>
```

EXAMPLE 2:

Enable user control of 3x7 segments on the MC405

```
>>DISPLAY.16 = 1
```

```
>>LCDSTR="123"
```

SEE ALSO:
`LCDSTR`

DISTRIBUTOR_KEY

TYPE:
Reserved Keyword

/ Divide

TYPE:
Mathematical operator

SYNTAX
`<expression1> / <expression2>`

DESCRIPTION:
Divides expression1 by expression2

PARAMETERS:

Expression1:	Any valid TrioBASIC expression
Expression2:	Any valid TrioBASIC expression

EXAMPLE:
Calculate a value for 'a' by dividing 10 by the sum of 2.1 and 9. The result is that a=0.9009

```
a=10/(2.1+9)
```

DLINK

TYPE:
System Command

SYNTAX:
`DLINK(function,…)`

DESCRIPTION:
This is a specialised command, to allow access to the SLM™ digital drive interface. The axis parameters have to be initialised by the **DLINK** function 2 command before the interface can be used for controlling an external drive.

The current **SLM** software dictates that the drive MUST be powered up after power is applied to the *Motion Coordinator*/ **SLM**.

PARAMETERS:

Function:	Specifies the required function.
0	Reserved function
1	Reserved function
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 communication channel associated with an axis
10	Read an EEPROM parameter

FUNCTION = 2:

SYNTAX:
`value = DLINK(2, slot, com)`

DESCRIPTION:
Check for presence SLM module on rear of motor.

PARAMETERS:

value:	Returns 1 if the SLM is answering, otherwise it returns 0.
slot:	The communications slot where the module is connected
com:	The communication channel where the axis is connected in the module

EXAMPLE

Check for a SLM module on slot 0, communication channel 0

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

FUNCTION = 3:

SYNTAX:

`value = DLINK(3, slot, com)`

DESCRIPTION:

Check for presence of SLM servo drive, such as MultiAx.

PARAMETERS:

value:	Returns 1 if the drive is answering, otherwise it returns 0.
slot:	The communications slot where the module is connected
com:	The communication channel where the axis is connected in the module

EXAMPLE:

Check for a SLM drive on slot 0, communication channel 0.

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

FUNCTION = 4:

SYNTAX:

`value = DLINK(4, slot, com, axis)`

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

value:	Returns **TRUE** if successful otherwise returns **FALSE**
slot:	The communications slot where the module is connected
com:	The communication channel where the axis is connected in the module
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:
Assign axis 0 to the drive connected to slot 0 and communication channel 0

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

FUNCTION = 5:

SYNTAX:

```
value =  DLINK(5, axis, parameter)
```

DESCRIPTION:
Read an SLM parameter

PARAMETERS:

value:	The value returned from SLM, returns -1 if the command fails
axis:	The axis number associated with the drive
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 the value of the SLM parameter 5 from axis 0.

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

FUNCTION = 6:

SYNTAX:

```
value = DLINK(6, axis, parameter, value)
```

DESCRIPTION:
Write an SLM parameter

PARAMETERS:

value:	Returns **TRUE** if successful otherwise returns **FALSE**
axis:	The axis number associated with the drive
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
value:	The value to write to the parameter

EXAMPLE:
Set SLM parameter 0 to the value 0 on axis 0.

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

FUNCTION = 7:

SYNTAX:
value = DLINK(7, axis, command)

DESCRIPTION:
Write an SLM command.

PARAMETERS:

value:	Returns **TRUE** if successful otherwise returns **FALSE**
axis:	The axis number associated with the drive Function 7
command:	The command number. (See drive documentation)

EXAMPLE:
Write SLM command 250 to axis 0

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

FUNCTION = 8:

SYNTAX:

```
value = DLINK(8, axis, parameter)
```

DESCRIPTION:

Read a drive parameter

PARAMETERS:

value:	The value returned from the drive, returns -1 if the command fails
axis:	The axis number associated with the drive
parameter:	The number of the drive parameter to be read. This is normally in the range 0...127. See the drive documentation for further information.

EXAMPLE:

Read drive parameter 53248 for axis 0

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

FUNCTION = 9:

SYNTAX:

```
value = DLINK(9, axis)
```

DESCRIPTION:

Return slot and communication channel associated with an axis

PARAMETERS:

value:	10 x slot number + communication channel, returns -1 if the command fails
axis:	The axis number associated with the drive.

EXAMPLE:

Read axis 2 SLM information

```
>>PRINT DLINK(9,2)
>>11.0000
```

This example is for slot 1, communication channel 1

FUNCTION = 10:

SYNTAX:

```
value = DLINK(10, axis, parameter)
```

DESCRIPTION:

Read an EEPROM parameter

PARAMETERS:

value:	The value from the EEPROM value, returns -1 if the command fails
axis:	The axis number associated with the drive.
parameter:	EEPROM parameter number. (See drive documentation)

EXAMPLE:

Return the EEPROM parameter 29, the Flux Angle from axis 0

```
>>PRINT DLINK(10,0,29)
>>62128.0000
```

$ Dollar

TYPE:

Special Character

SYNTAX

```
$number
```

DESCRIPTION:

The $ symbol is used to specify that the following signed 53bit number is in hexadecimal format.

EXAMPLES:

EXAMPLE 1:

Store the hexadecimal value of 38F3B into **VR** 10 and -A58 into **VR** 11

```
VR(10)=$38F3B
VR(11)=-$A58
```

EXAMPLE 2:

Turn on outputs 11,12,15,16

```
OP($CC00)
```

DPOS

TYPE:
Axis Parameter (Read Only)

DESCRIPTION:
The demand position **DPOS** is the demanded axis position generated by the motion commands.

DPOS is set to **MPOS** when **SERVO** or **WDOG** are OFF

DPOS can be adjusted without any motion by using **DEFPOS** or **OFFPOS**.

A step change in **DPOS** can be written using **ENDMOVE**

VALUE:
Demand position in user units. Default 0 on power up.

EXAMPLE:
Return the demand position for axis 10 in user units

```
>>? DPOS AXIS(10)
5432
>>
```

SEE ALSO:
DEFPOS, ENDMOVE, OFFPOS, AXIS_DPOS

DRIVE_CLEAR

TYPE:
Axis Function

SYNTAX:
`value = DRIVE_CLEAR(parameter)`

DESCRIPTION:
DRIVE_CLEAR allows the user to clear alarms in the drive. Currently this is only supports Panasonic A4N and A5N drives.

DRIVE_READ can be used to read the value of the alarm

PARAMETERS:

parameter:	0	Clear current alarm
	1	Clear all alarm history
	2	Clear all external alarms

SEE ALSO:
DRIVE_READ

DRIVE_CONTROL

TYPE:
Reserved Keyword

SEE ALSO:
DRIVE_READ, DRIVE_WRITE

DRIVE_CONTROLWORD

TYPE:
Axis Parameter

DESCRIPTION:
Sets the Control Word which is sent cyclically to a remote drive connected by a fieldbus. For example in CANopen over EtherCAT (CoE) the **DRIVE_CONTROLWORD** would set the value in object $6040 sub-index $00.

VALUE:
Example for a CANopen over EtherCAT (CoE) remote drive. See specific drive manuals for further details.

Bit	Description
0	Switch on
1	Enable voltage
2	Quick stop
3	Enable operation
4	Homing operation start

Bit	Description
5	Operation mode specific
6	Operation mode specific
7	Fault reset
8	Halt

EXAMPLE:

Write to the CoE control word sent cyclically to the drive connected as axis 6 on an EtherCAT network.

```
BASE(6)
DRIVE_CW_MODE=1 ' take manual control of the Control Word
DRIVE_CONTROLWORD = $2F ' set the bits to enable the drive
```

DRIVE_CW_MODE

TYPE:

Axis Parameter

DESCRIPTION:

The operation of the control word sent cyclically to a remote drive is, by default, controlled by the firmware. For example the control word will usually be under the control of the **WDOG** and **AXIS_ENABLE** parameters so that the drive can be enabled and disabled by software. Optionally, if **DRIVE_CW_MODE** is set to non-zero, the control word may be set by a user program.

VALUE:

The mode of operation for the drive control word.

0	System sets the value of the control word, depending on state of **WDOG** and **AXIS_ENABLE**. [default]
1	User program takes control of the control word via **DRIVE_CONTROLWORD**.
2	User program takes control of bits 11 to 15 via **DRIVE_CONTROLWORD**. Allows manufacturer specific bits to be changed while the enable bits are under control of **WDOG** and **AXIS_ENABLE**.

EXAMPLE:

EXAMPLE1

Take over the CoE control word sent cyclically to the drive connected as axis 0 on an EtherCAT network. Then toggle the reset bit.

```
BASE(0)
```

```
DRIVE_CW_MODE=1 ‘ take manual control of the Control Word
DRIVE_CONTROLWORD = $06 ‘ disable the drive
WA(10)
DRIVE_CONTROLWORD = $86 ‘ reset the drive
WA(10)
DRIVE_CONTROLWORD = $06
```

EXAMPLE2

Take over the CoE control word sent cyclically to the drive connected as axis 2 on an EtherCAT network. Then make a sequence to start homing.

```
  BASE(2)
  SERVO=OFF
  DRIVE_CW_MODE=1 ‘ set the control word to be user mode
  DRIVE_CONTROLWORD=$06 ‘ disable the drive
  ‘ Set the drive to DS402 homing mode
  CO_WRITE_AXIS(ax,$6060,$00,2,-1,6)
  ‘ wait for the homing mode to be accepted
  VR(100)=0
  REPEAT
    CO_READ_AXIS(ax,$6061,$00,2,100)
  UNTIL VR(100)=6

‘ set the homing method (1 for +ve direction, 2 for -ve)
  fwd=1
  rev=2
  CO_WRITE_AXIS(ax,$6098,$00,2,-1,fwd)

  DRIVE_CONTROLWORD=$1f ‘start homing
  WA(20)

  ‘ wait for Homing Done flag (bit 12)
  REPEAT
    WA(1)
  UNTIL DRIVE_STATUS.12=1
  WA(20)
  DEFPOS(ENCODER) ‘ set the axis position to drive’s value
  SERVO=ON
  WDOG=ON
  ‘ Set the drive to position mode
  CO_WRITE_AXIS(ax,$6060,$00,2,-1,8)
  ‘ Set control word to normal enabled state
  DRIVE_CONTROLWORD=$2f
  DRIVE_CW_MODE=0 ‘ set the control word back to wdog mode
```

DRIVE_FE

TYPE:
Axis Parameter

DESCRIPTION:
Returns the value of following error calculated by a remote drive in position mode. For this value to be active, the cyclic data transfer from the drive must be first configured to return the drive actual position error value. For a drive connected by CanOpen over EtherCAT (CoE) the value will be configured as part of the Process Data Object. (PDO)

VALUE:
The drive position error returned in drive units.

EXAMPLE:

EXAMPLE1
Display the drive's position error to *Motion* Perfect terminal 5.

```
PRINT #5,"Drive Position Error = ";DRIVE_FE AXIS(3)
```

EXAMPLE2
Wait for the drive's position error to go below a pre-defined threshold value.

```
BASE(2)
WAIT UNTIL ABS(DRIVE_FE) < 300
```

DRIVE_FE_LIMIT

TYPE:
Axis Parameter

ALTERNATE FORMAT:
None

DESCRIPTION:
This is the maximum allowable following error applied to the **DRIVE_FE** value. i.e. the actual following error in a remote drive which is received via a fieldbus such as EtherCAT. When exceeded the controller will generate an **AXISSTATUS** error, by default this will also generate a **MOTION_ERROR**. The **MOTION_ERROR** will disable the **WDOG** relay thus stopping further motor operation.

This limit may be used to guard against fault conditions such as mechanical lock-up, loss of encoder feedback, etc.

When either **DRIVE_FE_LIMIT** or **FE_LIMIT** are exceeded, bit 8 of **AXISSTATUS** is set.

VALUE:
The maximum allowable following error in user units. The default value is 20000 encoder edges.

EXAMPLE:
Initialise the axis as part of a **STARTUP** routine. **FE_LIMIT** is set larger than **DRIVE_FE_LIMIT** because the internal calculated FE is usually bigger than the following error calculated within the remote drive.

```
FOR x = 0 to 4
  BASE(x)
  UNITS = 100
  FE_LIMIT = 50
  DRIVE_FE_LIMIT = 10
  SPEED = 100
  ACCEL=1000
  DECEL=ACCEL
NEXT x
```

SEE ALSO:
FE, FE_LIMIT, DRIVE_FE

DRIVE_INDEX

TYPE:
Axis Parameter

SYNTAX:
DRIVE_INDEX AXIS(n) = value

DESCRIPTION:

DRIVE_INDEX is used to map additional **PDO** parameters in the EtherCAT servo drive into **VR** variables. The value given is the base **VR** address for the mapping. The non-standard **PDO** parameters are mapped one per **VR**, starting with the first **PDO** parameter following the standard objects.

This axis parameter can be added to the **MC_CONFIG**.

The EtherCAT drive must be configured with an application specific profile before this function can be used.

PARAMETERS:

value:	The **VR** index where incoming PDO data will be mapped

EXAMPLES:

EXAMPLE 1:

Transfer application data to and from the drive cyclically in the PDO telegram. The EtherCAT axis is pre-configured for special application software to run in the drive.

```
DRIVE_INDEX = 100
' Get incoming cyclic data
user_status_1 = VR(100)
user_status_2 = VR(101)
' Set outgoing data
VR(102) = user_control_word
VR(103) = winder_mode
VR(104) = ref_value_1
VR(105) = ref_value_2
VR(106) = correction_value
VR(107) = program_state
```

DRIVE_MODE

TYPE:

Axis Parameter (**MC_CONFIG**)

SYNTAX:

```
DRIVE_MODE AXIS(n) = value
```

DESCRIPTION:

DRIVE_MODE sets the mode of operation to be used by a remote drive over EtherCAT. This **MUST** be set in **MC_CONFIG** if the EtherCAT is to be initialised on power up in the required mode. **DRIVE_MODE** automatically sets the drive's mode of operation and the axis **ATYPE**.

This axis parameter can be added to the **MC_CONFIG**.

PARAMETERS:

value:	1 : Cyclic Synchronous Position mode (CSP) 2 : Cyclic Synchronous Velocity mode (CSV) 3: Cyclic Synchronous Torque mode (CST)

EXAMPLES:

EXAMPLE 1:

Four EtherCAT axes are to be set up, 2 axes in position mode, 1 axis in velocity mode and 1 axis in torque mode. Note that the *Motion Coordinator* can close the position loop when the drive is in CSV or CST mode, or the axis can be operated open-loop.

```
‘ setup 4 axes in MC_CONFIG
‘ Note: ATYPE is set automatically, do not set in MC_CONFIG
DRIVE_MODE AXIS(0)=1 ‘ position mode
DRIVE_MODE AXIS(1)=1 ‘ position mode
DRIVE_MODE AXIS(2)=2 ‘ velocity mode
DRIVE_MODE AXIS(3)=3 ‘ torque mode
```

SEE ALSO:

DRIVE_PROFILE

DRIVE_PARAMETER

TYPE:

Reserved Keyword

SEE ALSO:

DRIVE_READ, DRIVE_WRITE

DRIVE_PROFILE

TYPE:

Axis Parameter (**MC_CONFIG**)

SYNTAX:

DRIVE_PROFILE AXIS(n) = value

DESCRIPTION:

DRIVE_PROFILE allows the selection of different EtherCAT profiles from the internal database to be used with a remote drive over EtherCAT. This **MUST** be set in **MC_CONFIG** if the EtherCAT is to be initialised on power up with the required profile.

This axis parameter can be added to the **MC_CONFIG**.

The EtherCAT drive must have an application specific profile within the *Motion Coordinator*'s internal database before this function can be used.

PARAMETERS:

value:	0 :	Use the default “standard profile” with minimum objects passed between drive and *Motion Coordinator*.
	1 - n :	Use the application profile numbered.

EXAMPLES:

EXAMPLE 1:

Set up 4 axes to use application profiles for the cyclic PDO telegram. The EtherCAT axis profiles can be examined with the **ETHERCAT**($116, vendor_ID) command.

In the *Motion* Perfect terminal command line enter **ETHERCAT**($120) to see a list of **VENDOR** IDs.

```
ETHERCAT($120)
Kollmorgen (0x0000006A)
```

Next enter **ETHERCAT**($116, vendor_id)

```
ETHERCAT($116,$6a)
Kollmorgen (0x0000006A), AKD (0x00414B44), 65, (0)
Kollmorgen (0x0000006A), AKD (0x00414B44), 65, (1)
Kollmorgen (0x0000006A), AKD (0x00414B44), 65, (2)
etc.
```

The number in parentheses is the profile number. The profile PDO details will also be listed. 65 is the **ATYPE**, in this case EtherCAT velocity control.

In **MC_CONFIG**, put the required profile number for each axis.

```
DRIVE_PROFILE AXIS(0)=2
DRIVE_PROFILE AXIS(1)=2
DRIVE_PROFILE AXIS(2)=2
DRIVE_PROFILE AXIS(3)=1
```

SEE ALSO:

DRIVE_MODE

DRIVE_READ

TYPE:
Axis Function

SYNTAX:
`value = DRIVE_READ(parameter [,vr_index])`

DESCRIPTION:
DRIVE_READ allows the controller to read a parameter from a digital bus connected drive. Currently this is only supports Panasonic A4N and A5N drives.

The parameter index and details can be found in the *Motion* Perfect intelligent drives tool.

PARAMETERS:

	Value	Description
value:	1	**DRIVE_READ** was successful
	0	**DRIVE_READ** failed
	If vr_index is not used the return value is the parameter value	
parameter:	parameter_number	A4N parameter number to read
	(class * 256) + parameter_number or (class * $100) + parameter_number	A5N parameter number to read
	65536 + SSID_code or $10000 + SSID_code	Read a System ID into a **VRSTRING**
	131072 + (alarm_index * 4096) + alarm_function or $20000 + (alarm_index * $1000) + alarm_function	Read an Alarm code
	196608 + (index * 4096) + monitor_number or $30000 + (index * $1000) + monitor_number	Read a Monitor Value
vr_index:	VR in which to store the returned value	

System **ID**, Alarm codes and Monitor Commands apply to both A4N and A5N drives.

SYSTEM STRING ID CODES

SSID_code	Description
$010	Drive Vendor
$120	Drive Model No.
$130	Drive Serial No.
$140	Drive Firmware Version
$220	Motor Model No.
$230	Motor Serial No.
$310	External Scale Vendor
$320	External Scale Model No.

ALARM FUNCTIONS

Alarm Code	Description	Index
$000	Alarm Read	Index of alarm to be read
$001	Clear Current Alarm	0
$011	Clear All Alarms	0
$012	Clear External Alarm	0

DRIVE_CLEAR can be used to clear alarms

EXAMPLES:

EXAMPLE 1:

Read parameter 124, external scale direction from a A4N drive

```
success = DRIVE_READ(124,0)
IF success = 0 THEN
  PRINT “Error reading drive parameter”
ELSE
  PRINT “External scale direction = “; VR(0)[0]
ENDIF
```

EXAMPLE 2:

Read class 3 parameter 26, external scale direction from a A5N drive

```
success = DRIVE_READ(3 * 256 + 26,0)
IF success = 0 THEN
  PRINT “Error reading drive parameter”
ELSE
  PRINT “External scale direction = “; VR(0)[0]
ENDIF
```

EXAMPLE 3:

Read the system ID to find the Panasonic servo drive serial number into a **VRSTRING** starting at **VR**(0).

```
success = DRIVE_READ($10000 + $130,0)
IF success = 0 THEN
  PRINT “Error reading drive parameter”
ELSE
  PRINT “Driver Serial No. = “;VRSTRING(0)
ENDIF
```

EXAMPLE 4:

Read the alarm history from the Panasonic servo drive.

```
PRINT “Alarm Read AXIS(“;axis_no[0];”)”
FOR past_alarm = 0 TO 14
  DRIVE_READ($20000 + past_alarm * 4096 + 0 ,0)
  PRINT “Alarm history index “; past_alarm[0];” = “;VR(0)[0]
NEXT past_alarm
```

EXAMPLE 5:

Read monitor type code 102 to find the encoder resolution of a Panasonic servo drive.

```
success = DRIVE_READ($30000 + $102, 0)
IF success = FALSE THEN
  PRINT “Error reading drive parameter”
ELSE
  PRINT “Encoder resolution = “;VR(0)[0]
ENDIF
```

EXAMPLE 6:

The following routine can be used to home to the Z mark on the motor encoder using an A4N.This works by waiting for the Z mark o be seen on the drive then reading the mechanical angle.

```
pos = DRIVE_READ($30201)
oneturn=10000’ Distance for one turn depends on encoder type

IF pos <> -1 THEN
  PRINT “Mechanical offset:”;pos[0]
```

```
ELSE
  PRINT “Drive has not yet seen Z mark”
  MOVE(oneturn)
  WAIT UNTIL DRIVE_READ($30201)<>-1
  CANCEL
  WAIT IDLE
  pos = DRIVE_READ($30201)
  PRINT “Mechanical offset:”;pos[0]
ENDIF
DEFPOS(pos)
```

DRIVE_SET_VAL

TYPE:
Reserved Keyword

SEE ALSO:
DRIVE_READ, **DRIVE_WRITE**

DRIVE_STATUS

TYPE:
Axis Parameter

DESCRIPTION:
Returns the Status Word received cyclically from a remote drive connected by a fieldbus. For example in CANopen over EtherCAT (CoE) the **DRIVE_STATUS** would have the value from object $6041 sub-index $00.

VALUE:
Example for a CANopen over EtherCAT (CoE) remote drive. See specific drive manuals for further details.

Bit	Description
0	Ready to switch on
1	Switched on
2	Operation enabled
3	Fault

4	Voltage enabled
5	quick stop
6	switch on disabled
7	warning

EXAMPLE:

Read the CoE status from the drive connect as axis 4 on an EtherCAT network.

```
PRINT #5,HEX(DRIVE_STATUS AXIS(4))
```

DRIVE_TORQUE

TYPE:

Axis Parameter

DESCRIPTION:

Returns the actual torque value calculated by a remote drive. For this value to be active, the cyclic data transfer from the drive must be first configured to return the drive actual torque value. For a drive connected by CanOpen over EtherCAT (CoE) the value will be configured as part of the Process Data Object. (PDO)

VALUE:

The drive torque returned in drive units.

EXAMPLE:

EXAMPLE1

Display the drive's torque to *Motion* Perfect terminal 5.

```
PRINT #5,"Drive torque value = ";DRIVE_TORQUE AXIS(2)
```

EXAMPLE2

Wait for the drive's torque value to go below a pre-defined level.

```
BASE(16)
WAIT UNTIL DRIVE_TORQUE < 3000
```

DRIVE_VALUE

TYPE:
Reserved Keyword

SEE ALSO:
DRIVE_READ, **DRIVE_WRITE**

DRIVE_WRITE

TYPE:
Axis Function

SYNTAX:
`result = DRIVE_WRITE (parameter, value)`

DESCRIPTION:
DRIVE_WRITE allows the controller to write to a parameter from a digital bus connected drive. Currently this is only supports Panasonic A4N and A5N drives.

The parameter numbers and details can be found in the *Motion* Perfect intelligent drives tool.

PARAMETERS:

result:	1	**DRIVE_WRITE** was successful
	0	**DRIVE_WRITE** failed
parameter:	parameter_number	A4N parameter to write
	class * 256 + parameter_number	A5N parameter to write
	128	Stores all drive parameters into EPROM
	129	Resets all drive parameters to default values
value:		The value to be written to the parameter. (Use 0 for parameter numbers 128 & 129)

EXAMPLES:

EXAMPLE 1:

Write parameter 122, encoder scale on an A4N drive

```
success = DRIVE_WRITE(122, 10000)
If success = 0 THEN
  PRINT “Error writing drive parameter”
ELSE
  PRINT “Encoder scale set”
ENDIF
```

EXAMPLE 2:

Write class 0 parameter 8, encoder scale on an A5N drive

```
success = DRIVE_WRITE(0 * 256 + 8, 15000)
If success = 0 THEN
  PRINT “Error writing drive parameter”
ELSE
  PRINT “Encoder scale set”
ENDIF
```

EXAMPLE 3:

Store all drive parameters in EPROM

```
success = DRIVE_WRITE(128, 0)
IF success = 0 THEN
  PRINT “Error storing drive parameters to EPROM”
ELSE
  PRINT “Drive parameters stored in EPROM”
ENDIF
```

EXAMPLE 4:

Reset all drive parameters to default values

```
success = DRIVE_WRITE(129, 0)
IF success = 0 THEN
  PRINT “Error resetting drive parameters”
ELSE
  PRINT “Drive parameters reset to defaults”
ENDIF
```

DRIVEIO_BASE

TYPE:

System Parameter (**MC_CONFIG**)

DESCRIPTION:

This parameter sets the start address of any drive I/O channels. Together with **CANIO_BASE**, **MODULEIO_BASE** and **NODE_IO** the I/O allocation scheme can replace and expand the behaviour of **MODULE_IO_MODE**.

VALUE:

-1	Drive I/O disabled (default)
0	Drive I/O allocated automatically
>= 8	Drive I/O is located at this IO point address, truncated to the nearest multiple of 8

EXAMPLE:

A system with MC464, a Panasonic module (slot 0) and a **CANIO** Module will have the following I/O assignment:

DRIVEIO_BASE=0 + **MODULEIO_BASE**=0 + **CANIO_BASE**=0

0-7	Built in inputs
8-15	Built in bi-directional I/O
16-23	Panasonic module inputs
24-39	**CANIO** bi-directional I/O
40-47	Panasonic drive inputs
48-1023	Virtual I/O

DRIVEIO_BASE=-1 + **MODULEIO_BASE**=0 + **CANIO_BASE**=0

0-7	Built in inputs
8-15	Built in bi-directional I/O
16-23	Panasonic module inputs
24-39	**CANIO** bi-directional I/O
40-1023	Virtual I/O

DRIVEIO_BASE=200 + **MODULEIO_BASE**=80 + **CANIO_BASE**=400

0-7	Built in inputs
8-15	Built in bi-directional I/O
16-79	Virtual I/O
80-87	Panasonic module inputs
88-199	Virtual I/O
200-207	Panasonic drive inputs
208-399	Virtual I/O
400-415	**CANIO** bi-directional I/O
416-1023	Virtual I/O

SEE ALSO:
CANIO_BASE, MODULEIO_BASE, NODE_IO, MODULE_IO_MODE

DUMP

TYPE:
Reserved Keyword

EDPROG

TYPE:
System Command

SYNTAX:
`EDPROG [parameters,] function`

ALTERNATE FORMAT:
`& function[, parameters]`

DESCRIPTION:
This is a special command that may be used to manipulate the **SELECTed** programs on the controller.

It is not normally used except by *Motion* Perfect.

FUNCTIONS:

1	I	Insert string
2	S	Search for string
3	D	Delete line
4	L	Print lines
5	N	Print number of lines
6	A	Print label addresses
7	C	Prints the name of the currently selected program
8	R	Replace line
9	K	Print checksum
10	Z	Print checksum of specified program
11	X	Print object code checksum
12	Q	Checks if the controller directory is corrupt
13	V	Print variable list
14	M	Commit changes

FUNCTION = A:

SYNTAX:

EDPROG 6, to_line, from_line

ALTERNATE SYNTAX:

& from_line, to_line A

DESCRIPTION:

Prints all label names in the region defined in the **SELECTed** program.

PARAMETERS:

from_line:	The first line of the **SELECTed** program to search
to_line:	The last line of the **SELECTed** program to search

FUNCTION = C:

SYNTAX:

EDPROG C

ALTERNATE SYNTAX:

& C

DESCRIPTION:

Prints the name of the currently **SELECTed** program.

FUNCTION = D:

SYNTAX:

EDPROG 3, line_no

ALTERNATE SYNTAX:

& line_no D

DESCRIPTION:

Deletes the specified line

PARAMETER:

line_no:	Any valid line number form the **SELECTed** program

FUNCTION = I:

SYNTAX:
EDPROG string, 1, line_no

ALTERNATE SYNTAX:
& line_no I,string

DESCRIPTION:
Insert the text string in the currently selected program at the specified line.

You should **NOT** enclose the string in quotes unless they need to be inserted into the program.

PARAMETERS:

line_no:	The line to insert the string
string:	The text string to insert into the **SELECTed** program

FUNCTION = K:

SYNTAX:
EDPROG 10

ALTERNATE SYNTAX:
& K

DESCRIPTION:
Print the checksum of the system software

FUNCTION = L:

SYNTAX:
EDPROG 4, end, start

ALTERNATE SYNTAX:

`& start, end L`

DESCRIPTION:

Print the lines of the currently selected program between start and end

PARAMETERS:

start:	The first line to print from the **SELECTed** program
end:	The last line to print from the **SELECTed** program

FUNCTION = M:

SYNTAX:

`EDPROG 14`

ALTERNATE SYNTAX:

`& M`

DESCRIPTION:

Saves all program changes to flash.

FUNCTION N:

SYNTAX:

`EDPROG 5`

ALTERNATE SYNTAX:

`& N`

DESCRIPTION:

Print the number of lines in the currently **SELECTed** program

FUNCTION = Q:

SYNTAX:

`EDPROG 12`

ALTERNATE SYNTAX:
`& Q`

DESCRIPTION:
Returns the state of the controllers program memory.

RETURN VALUE:

0	Controller memory OK
1	Controller memory corrupted

FUNCTION = R:

SYNTAX:
`EDPROG string, 8, line`

ALTERNATE SYNTAX:
`& line R, string`

DESCRIPTION:
Replace the line <line> in the currently **SELECTed** program with the text <string>.

You should **NOT** enclose the string in quotes unless they need to be inserted into the program.

PARAMETERS:

line_no:	The line to replace
string:	The text string to replace the line in the **SELECTed** program

FUNCTION = S:

SYNTAX:
`EDPROG string, 2, to_line, from_line`

ALTERNATE SYNTAX:
`& from_line, to_line S string`

DESCRIPTION:
Prints the line number of the first occurrence of the string in the region defined in the **SELECTed** program.

PARAMETERS:

from_line:	The first line of the **SELECTed** program to search
to_line:	The last line of the **SELECTed** program to search
string	The string to search for

FUNCTION = V:

SYNTAX:
`EDPROG 13`

ALTERNATE SYNTAX:
`& V`

DESCRIPTION:
Print all variables defined in the **SELECTed** program.

FUNCTION = X:

SYNTAX:
`EDPROG 11`

ALTERNATE SYNTAX:
`& X`

DESCRIPTION:
Print the 16bit CRC checksum of the **SELECTed** program.

FUNCTION = Z:

SYNTAX:
`EDPROG progname, 10`

ALTERNATE SYNTAX:
`& Z, progname`

DESCRIPTION:
Print the CRC checksum of the specified program.

RETURN VALUE:
Returns the checksum using standard **CCITT** 16 bit generator polynomial.

SEE ALSO:
SELECT

EDPROG1

TYPE:
System Command

SYNTAX:
EDPROG1 prog_name,[parameters,] function

ALTERNATE FORMAT:
! prog_name, prog_name, function[, parameters]

DESCRIPTION:
This is a special command that may be used to manipulate the **SELECTed** programs on the controller.

It is not normally used except by *Motion* Perfect.

FUNCTIONS:

1	I	Insert string
2	S	Search for string
3	D	Delete line
4	L	Print lines
5	N	Print number of lines
6	A	Print label addresses
7	C	Prints the name of the currently selected program
8	R	Replace line
9	K	Print checksum
10	Z	Print checksum of specified program

11	X	Print object code checksum
12	Q	Checks if the controller directory is corrupt
13	V	Print variable list
14	M	Commit changes

FUNCTION = A:

SYNTAX:
`EDPROG16, to_line, from_line`

ALTERNATE SYNTAX:
`! prog_name, from_line, to_line A`

DESCRIPTION:
Prints all label names in the region defined in the **SELECTed** program.

PARAMETERS:

from_line:	The first line of the **SELECTed** program to search
to_line:	The last line of the **SELECTed** program to search

FUNCTION = C:

SYNTAX:
`EDPROG1C`

ALTERNATE SYNTAX:
`! prog_name, C`

DESCRIPTION:
Prints the name of the currently **SELECTed** program.

FUNCTION = D:

SYNTAX:
`EDPROG1 prog_name, 3, line_no`

ALTERNATE SYNTAX:
```
! prog_name, line_no D
```

DESCRIPTION:
Deletes the specified line

PARAMETER:

line_no:	Any valid line number form the **SELECTed** program

FUNCTION = I:

SYNTAX:
```
EDPROG1 prog_name, string, 1, line_no
```

ALTERNATE SYNTAX:
```
! prog_name, line_no I,string
```

DESCRIPTION:
Insert the text string in the currently selected program at the specified line.

You should **NOT** enclose the string in quotes unless they need to be inserted into the program.

PARAMETERS:

line_no:	The line to insert the string
string:	The text string to insert into the **SELECTed** program

FUNCTION = K:

SYNTAX:
```
EDPROG1 prog_name, 10
```

ALTERNATE SYNTAX:
```
! prog_name, K
```

DESCRIPTION:
Print the checksum of the system software

FUNCTION = L:

SYNTAX:

```
EDPROG1 prog_name, 4, end, start
```

ALTERNATE SYNTAX:

```
! prog_name, start, end L
```

DESCRIPTION:

Print the lines of the currently selected program between start and end

PARAMETERS:

start:	The first line to print from the **SELECTed** program
end:	The last line to print from the **SELECTed** program

FUNCTION = M:

SYNTAX:

```
EDPROG1 prog_name, 14
```

ALTERNATE SYNTAX:

```
! prog_name, M
```

DESCRIPTION:

Saves all program changes to flash.

FUNCTION N:

SYNTAX:

```
EDPROG1 prog_name, 5
```

ALTERNATE SYNTAX:

```
! prog_name, N
```

DESCRIPTION:

Print the number of lines in the currently **SELECTed** program

FUNCTION = Q:

SYNTAX:

`EDPROG1 prog_name, 12`

ALTERNATE SYNTAX:

`! prog_name, Q`

DESCRIPTION:

Returns the state of the controllers program memory.

RETURN VALUE:

0	Controller memory OK
1	Controller memory corrupted

FUNCTION = R:

SYNTAX:

`EDPROG1 prog_name, string, 8, line`

ALTERNATE SYNTAX:

`! prog_name, line R, string`

DESCRIPTION:

Replace the line <line> in the currently **SELECTed** program with the text <string>.

You should **NOT** enclose the string in quotes unless they need to be inserted into the program.

PARAMETERS:

line_no:	The line to replace
string:	The text string to replace the line in the **SELECTed** program

FUNCTION = S:

SYNTAX:

`EDPROG1 prog_name, string, 2, to_line, from_line`

ALTERNATE SYNTAX:

`! prog_name, from_line, to_line S string`

DESCRIPTION:
Prints the line number of the first occurrence of the string in the region defined in the **SELECTed** program.

PARAMETERS:

from_line:	The first line of the **SELECTed** program to search
to_line:	The last line of the **SELECTed** program to search
string	The string to search for

FUNCTION = V:

SYNTAX:
```
EDPROG1 prog_name, 13
```

ALTERNATE SYNTAX:
```
! prog_name, V
```

DESCRIPTION:
Print all variables defined in the **SELECTed** program.

FUNCTION = X:

SYNTAX:
```
EDPROG1 prog_name, 11
```

ALTERNATE SYNTAX:
```
! prog_name, X
```

DESCRIPTION:
Print the 16bit CRC checksum of the **SELECTed** program.

FUNCTION = Z:

SYNTAX:
```
EDPROG1 prog_name, progname, 10
```

ALTERNATE SYNTAX:
```
! prog_name, Z, progname
```

DESCRIPTION:
Print the CRC checksum of the specified program.

RETURN VALUE:
Returns the checksum using standard **CCITT** 16 bit generator polynomial.

SEE ALSO:
SELECT

ENCODER

TYPE:
Axis Parameter (Read Only)

DESCRIPTION:
The **ENCODER** axis parameter holds a raw copy of the positional feedback device.

The **MPOS** axis measured position is calculated from the **ENCODER** value automatically allowing for overflows and offsets.

VALUE:

Feedback device	Value
Incremental encoder:	The value latched in the encoder hardware register
Absolute Encoder:	The positional value using the number of bits set in **ENCODER_BITS**
Digital Axis:	Raw position feedback from the drive

EE ALSO:
ENCODER_BITS, MPOS

ENCODER_BITS

TYPE:
Axis Parameter (**MC_CONFIG**)

DESCRIPTION:
This parameter is only used with an absolute encoder axis. It is used to set the number of data bits to be clocked out of the encoder by the axis hardware. There are 2 types of absolute encoder supported by this

parameter; SSI and EnDat.

If the number of **ENCODER_BITS** is to be changed, the parameter must first be set to zero before entering the new value.

ENCODER_BITS must be set before the **ATYPE** is set

VALUE:

Encoder type	Bits	Value	Function
All:	0	0	No data is clocked out of the encoder (default)
SSI:	Bit 0-5	0-32	The number of bits to be clocked out of the encoder.
	Bit 6	64	Set for Binary, clear for Gray code (default)
	Bit 7	128	Reverses direction (inverts the data bits)
EnDat:	Bits 0..7	0-255	The total number of data bits returned
	Bits 8..13	256-8192	The number of multi-turn bits
	Bit 14	16384	This is set by the controller when a correct CRC is calculated from the encoder position data.

EXAMPLES:

EXAMPLE 1:

Set up 2 axes of SSI absolute encoder

```
ENCODER_BITS AXIS(3) = 12
ENCODER_BITS AXIS(7) = 21
```

EXAMPLE 2:

Re-initialise **MPOS** using absolute value from encoder

```
SERVO=OFF
ENCODER_BITS = 0
ENCODER_BITS = databits
```

EXAMPLE 3:

A 25 bit EnDat encoder has 12 multi-turn and 13 bits/turn resolution. (Total number of bits is 25)

```
ENCODER_BITS = 25 + (256 * 12)
ATYPE = 47
```

SEE ALSO:
ATYPE, ENCODER_CONTROL, ENCODER_READ, ENCODER_WRITE

ENCODER_CONTROL

TYPE:
Axis Parameter

DESCRIPTION:
Endat encoders can be set to either cyclically return their position, or they can be set to a parameter read/write mode.

Using the **ENCODER_READ** or **ENCODER_WRITE** functions will set the parameter to 1 automatically.

VALUE:

0	position return mode (default value)
1	sets parameter read/write mode

EXAMPLE:
Reset **ENCODER_CONTROL** after an **ENCODER_READ** so that the position is returned.

```
value = ENCODER_READ($A700)
ENCODER_CONTROL = 0
```

SEE ALSO:
ENCODER_READ, ENCODER_WRITE

ENCODER_FILTER

TYPE:
Axis Parameter

DESCRIPTION:
This parameter allows filtering to be applied to an encoder feedback to reduce the impact of jitter. The smaller the value the larger the time constant and so the less impact jitter will have on the system.

This parameter can be used to reduce jitter on a master axis which is linked to another axis.

VALUE:
Filter parameter range 0.001 to 1 (default 1).

EXAMPLE:
Apply a filter to a line encoder so that the connected axes are not affected by any jitter:

```
BASE(0)
ENCODER_FILTER= 0.95
BASE(1)
CONNECT(1,0)
```

ENCODER_ID

TYPE:
Axis Parameter

DESCRIPTION:
This parameter returns the Encoder Identification (**ENID**) parameter from a Tamagawa absolute encoder.

VALUE:
Only encoders returning 17 are currently supported

EXAMPLE:
Initialise a Tamagawa absolute encoder and check it is working by looking at **ENCODER_ID**.

```
ATYPE = 46
IF ENCODER_ID<>17 THEN
  PRINT#term, “Incorrect ENID”
ENDIF
```

ENCODER_RATIO

TYPE:
Axis Command

SYNTAX:
ENCODER_RATIO(mpos_count, input_count)

DESCRIPTION:
This command allows the incoming encoder count to be scaled by a non integer ratio:

MPOS = (mpos_count / input_count) x encoder_edges_input

When using the servo loop you will need to adjust the gains to maintain performance and stability.

Unlike the **UNITS** parameter, which only affects the scaling seen by the user programs, **ENCODER_RATIO** affects all motion commands.

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

PARAMETERS:

mpos_count:	An integer number which defines the numerator
input_count:	An integer number which defines the denominator

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

EXAMPLES:

EXAMPLE 1:

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

As 8192 cannot be exactly divided into 360 **ENCODER_RATIO** is used to adjust the encoder feedback.

The highest value that is less than 8192 yet divides into 360 should be chosen. This is 7200 (7200 / 20 = 360). This reduces the resolution from 0.044 to 0.055 degrees, but enables you to program easily in degrees.

```
ENCODER_RATIO(7200,8192)
UNITS = 20 ‘ axis calibrated in degrees
```

EXAMPLE 2:

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

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

EXAMPLE 3:

Set up an axis to work in the reverse direction. For a servo axis, both the **ENCODER_RATIO** and the **DAC_SCALE** must be set to minus values.

```
BASE(5) ‘ set axis 5 to work in reverse direction
```

```
DAC_SCALE = -1
ENCODER_RATIO(-1,1)
```

EXAMPLE 4:

Set up a digital position control axis, for example EtherCAT Position, to work in the reverse direction. For an axis where the servo-drive closes the position loop, both the **ENCODER_RATIO** and the **STEP_RATIO** must be set to minus values.

```
BASE(30) ‘ set axis 30 to work in reverse direction
ENCODER_RATIO(-1,1)
STEP_RATIO(-1,1)
```

SEE ALSO:

STEP_RATIO, DAC_SCALE

ENCODER_READ

TYPE:

Axis Function

SYNTAX:

value = ENCODER_READ (address)

DESCRIPTION:

Read an internal register from an EnDat absolute encoder.

PARAMETERS:

value:	Value returned from the specified register. Returns -1 if the encoder has not been initialised
address:	The address of the EnDat encoder register to be read

EXAMPLES:

EXAMPLE 1

Initialise and check an EnDat encoder

```
ENCODER_BITS=25+256*12
ATYPE=47
IF ENCODER_READ($A700)=-1 then
  PRINT “Failed to initialise EnDat Encoder
ENDIF
ENCODER_CONTROL=0
```

EXAMPLE 2

Read the number of encoder bits from an EnDat encoder. This can be done before **ENCODER_BITS** is set to find the correct value to use. This command will work with any EnDat 2.1 encoder.

```
>>BASE(1)
>>PRINT ENCODER_READ($A10d)AND $3F
25
>>
```

SEE ALSO:

ENCODER_CONTROL, ENCODER_WRITE

ENCODER_STATUS

TYPE:

Axis Parameter

DESCRIPTION:

This axis parameter returns both the status field SF and the **ALMC** encoder error field from a Tamagawa absolute encoder.

VALUE:

Bits 0..7	SF field
Bits 8..15	**ALMC** field

Value is 0 if the encoder has not been initialised

EXAMPLE:

Print the SF field and **ALMC** field in hex

```
PRINT “SF field = 0x”; HEX (ENCODER_STATUS AND $FF)
PRINT “ALMC field = 0x”; HEX ((ENCODER_STATUS AND $FF00)/$FF)
```

ENCODER_TURNS

TYPE:

Axis Parameter

DESCRIPTION:

Returns the number of multi-turn counts from EnDat or Tamagawa absolute encoders.

The multi-turn data is not automatically applied to the axis **MPOS** after initialisation of a Tamagawa absolute encoder. The application programmer must apply this from BASIC using **OFFPOS** or **DEFPOS** as required.

VALUE:

The number of multi-turn counts from the encoder.

EXAMPLE:

Initialise a Tamagawa encoder and apply the number of turns to **MPOS**. The encoder returns 17bits for the position and 16bits for the number of turns.

```
ATYPE=46
OFFPOS= ENCODER_TURNS*2^17
WAIT UNTIL OFFPOS = 0
```

ENCODER_WRITE

TYPE:

Axis Function

SYNTAX:

Value = ENCODER_WRITE (address, data)

DESCRIPTION:

Write an internal register to an Absolute Encoder on an EnDat absolute encoder.

PARAMETERS:

value:	Returns **TRUE** if the write was successful and **FALSE** if it fails
address:	The address of the EnDat encoder register to be written to
data:	Value to be written to the specified register.

EXAMPLE:

Write a value to the EnDat encoder and check it has been written, then set the encoder back to position mode

```
IF NOT ENCODER_WRITE (endat_address, setvalue) THEN
  PRINT “Fail to write to encoder”
ENDIF
ENCODER_CONTROL=0
```

SEE ALSO:
ENCODER_CONTROL, ENCODER_READ

END_DIR_LAST

TYPE:
Axis Parameter

DESCRIPTION:
Returns the direction of the end of the last loaded interpolated motion command. You can use the parameter to set an initial direction before loading a SP motion command. **END_DIR_LAST** will be the same as **START_DIR_LAST** except in the case of circular moves.

Write to **END_DIR_LAST** when initialising a system or after a sequence of moves which are not **SP** commands.

This parameter is only available when using **SP** motion commands such as **MOVESP**, **MOVEABSSP** etc.

VALUE:
End direction, in radians between -PI and PI. Value is always positive.

EXAMPLES:

EXAMPLE1:
Return the end direction of a move.

```
>>MOVESP(10000,-10000)
>>PRINT END_DIR_LAST
2.3562
>>
```

EXAMPLE 2:
Write to the end direction to set the direction of the **MOVE** before calculating the change.

```
MOVE(10000,-10000)
END_DIR_LAST = 2.3562
MOVESP(10000,1324)
VR(10)=CHANGE_DIR_LAST
```

SEE ALSO:
CHANGE_DIR_LAST, START_DIR_LAST

ENDMOVE

TYPE:
Axis Parameter

DESCRIPTION:
This parameter holds the absolute 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.

Writing to **DPOS** will make a step changes. This can easily lead to "Following error exceeds limit" errors unless the steps are small or the **FE_LIMIT** is high.

As it is an absolute value **ENDMOVE** is adjusted by **OFFPOS**/**DEFPOS**. The individual moves in the buffer are incremental and are not adjusted by **OFFPOS**.

VALUE:
The absolute position of the end of the current move in user **UNITS**.

EXAMPLE:
Check the value of **ENDMOVE** to confirm you calculated move is correct.

```
MOVE(distance*pitch)
IF ENDMOVE>200 THEN
  CANCEL
  PRINT#5, "Calculated distance to large"
ENDIF
```

ENDMOVE_BUFFER

TYPE:
Axis Parameter (Read only)

DESCRIPTION:
This holds the absolute position of end of the buffered sequence of moves.

As it is an absolute value **ENDMOVE_BUFFER** is adjusted by **OFFPOS**/**DEFPOS**. The individual moves in the buffer are incremental are not adjusted by **OFFPOS**.

VALUE:
Returns the length of all remaining moves for an axis.

EXAMPLE:
Add some moves to the buffer, then check the value of **ENDMOVE_BUFFER**

```
>>MOVE(100)
>>MOVE(150)
>>MOVE(25)
>>PRINT ENDMOVE_BUFFER
275.000
>>
```

ENDMOVE_SPEED

TYPE:
Axis Parameter

DESCRIPTION:
This parameter sets the end speed for a motion command that support the advanced speed control (commands ending in SP). The **VP_SPEED** will decelerate until **ENDMOVE_SPEED** is reached at the end of the profile.

The lowest value of **ENDMOVE_SPEED**, **FORCE_SPEED** or **STARTMOVE_SPEED** will take priority.

ENDMOVE_SPEED is loaded into the buffer at the same time as the move so you can set different speeds for subsequent moves. If there is no further motion commands in the buffer the current move will decelerate to a stop.

VALUE:
The speed at which the SP motion command will end, in user **UNITS**. (default 0)

EXAMPLES:

EXAMPLE 1:
In this example the controller will start ramping down the speed (at the specified rate of **DECEL**) so at the end of the **MOVESP**(20) the **VP_SPEED**=10. The next move continues with a **FORCE_SPEED** of 10. The final **ENDMOVE_SPEED** is overwritten to zero as there are no more buffered moves.

```
FORCE_SPEED=15
ENDMOVE_SPEED=10
MOVESP(20)
FORCE_SPEED=10
ENDMOVE_SPEED=5
```

```
MOVESP(5)
```

EXAMPLE 2:
A machine can merge interpolated moves however it must slow down to 50% of the speed for the transition.

```
FORCE_SPEED=1000
ENDMOVE_SPEED=500 '50% of FORCE_SPEED
MOVE(100,10)
MOVE(70,-10)
MOVE(120,15)
```

EPROM

TYPE:
Reserved Keyword

EPROM_STATUS

TYPE:
Reserved Keyword

= Equals

TYPE:
Mathematical operator
(Comparison or assignment operator).

COMPARISON OPERATOR:

SYNTAX:
`<expression1> = <expression2>`

DESCRIPTION:
Returns **TRUE** if expression1 is equal to expression2, otherwise returns **FALSE**.

PARAMETERS:

Expression1:	Any valid TrioBASIC expression
Expression2:	Any valid TrioBASIC expression

EXAMPLE:

```
IF IN(7)=ON THEN GOTO label
```

If input 7 is ON then program execution will continue at line starting “label:”

ASSIGNMENT OPERATOR:

SYNTAX:

```
Value = expression
```

DESCRIPTION:

Assigns a value from the result of the expression.

PARAMETERS:

value:	the variable in which to store the value
expression:	any valid TrioBASIC expression

EXAMPLE:

Set the sum of 10 and 9 into local variable ‘result’

```
result = 10 + 9
```

ERROR_AXIS

TYPE:

System Parameter (Read Only)

DESCRIPTION:

Returns the number of the axis that caused the **MOTION_ERROR**.

ERROR_AXIS should only be read when **MOTION_ERROR**<>0

VALUE:

Number of the axis that caused the **MOTION_ERROR**

This default value is 0 and is reset to 0 after **DATUM**(0)

EXAMPLE:

If there is a motion error print error information.

```
IF MOTION_ERROR THEN
  PRINT#5, "Axis to cause error = "; ERROR_AXIS
  PRINT#5, "AXISSTATUS of ERROR_AXIS = "; AXISSTATUS AXIS( ERROR_AXIS)
ENDIF
```

SEE ALSO:

AXISSTATUS, MOTION_ERROR, FE_LATCH

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.

VALUE:

The line number on the specified process that caused the error

EXAMPLE:

Display the **ERROR_LINE** as part of a sub routine called by 'ON **BASICERROR GOTO**'

```
error_routine:
  VR(100) = RUN_ERROR
  PRINT "The error ";RUN_ERROR[0];
  PRINT " occurred in line ";ERROR_LINE[0]
STOP
```

SEE ALSO:

BASICERROR, RUN_ERROR

ERRORMASK

TYPE:

Axis Parameter

DESCRIPTION:

The value held in this parameter is bitwise ANDed 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.

After a critical error has tripped the enable relay, the *Motion Coordinator* must either be reset, or a **DATUM**(0) command must be executed to reset the error flags.

VALUE:

The mask to be ANDed with the **AXISSTATUS**

For the MC464, the default value is 268 which will trap critical errors. This is **AXISSTATUS** bits 2, 3 and 8 which are digital drive communication errors and exceeding the following error limit.

EXAMPLE:

Configure the **ERRORMASK** so that the **WDOG** is turned off when there are communication failures (4), remote drive errors (8), the following error exceeds the limit (256) or the limit switches have been hit(16 + 32).

```
ERRORMASK= 4+8+16+32+256
```

SEE ALSO:

AXISSTATUS, DATUM(0)

ETHERCAT

TYPE:

System Command

SYNTAX:

ETHERCAT(function, slot [,parameters…])

DESCRIPTION:

The command **ETHERCAT** is used to perform advanced operations on the EtherCAT network. In normal use the EtherCAT network will start automatically without the need for any commands in a startup program. Some **ETHERCAT** command functions may be useful when debugging and setting up an EtherCAT system, so a small sub-set is described here.

The **ETHERCAT** command returns **TRUE**(-1) if successful and **FALSE** (0) if the command execution was in error. Functions which return a value must either put the value in a **VR** or print it to the current output terminal.

PARAMETERS:

function:	Function to be performed	
	$00	Start EtherCAT network
	$01	Stop EtherCAT network
	$21	Set EtherCAT State
	$22	Get EtherCAT State
	$64	Send reset sequence to a drive
	$87	Display network configuration
slot:	Set to the P876 EtherCAT module slot number	

FUNCTION = $00: START ETHERCAT NETWORK

SYNTAX:

```
ETHERCAT(0, slot, [,MAC_retries])
```

DESCRIPTION:

Initialise EtherCAT network, and put it onto operational mode.

PARAMETERS:

MAC_retries:	Sets the number of times the master attempts to restart the Ethernet auto-negotiation. Default = 2.

EXAMPLE:

Check for the EtherCAT state and if not in Operational State, restart the EtherCAT and set an output to indicate that a re-start is in progress.

```
'--Init EtherCAT if needed.
slt=0
ecs_vr=30 'use VR 30 for returned value
chk = ETHERCAT($06,slt,ecs_vr) 'test state

IF chk<>TRUE OR VR(ecs_vr)<>3 THEN
  OP(9,ON)
  WA(15000) 'wait 15sec for drive to power up
  ETHERCAT(0,slt) 'init EtherCAT
ENDIF
```

FUNCTION = $01: STOP ETHERCAT NETWORK

SYNTAX:

ETHERNET(1, slot)

DESCRIPTION:

Closedown the EtherCAT network.

PARAMETERS:

None.

EXAMPLE:

Stop the EtherCAT protocol from the terminal and then re-start it.

```
>>ETHERCAT(1, 0)
>>ETHERCAT(1, 0)
>>
```

FUNCTION = $21: SET ETHERCAT STATE

SYNTAX:

ETHERCAT($21, slot, state, display)

DESCRIPTION:

This function controls the EtherCAT State Machine. (ESM) It requests the master change to given EtherCAT 'state', and hence changes all slaves to the same state. When a change to a higher state is made, the EtherCAT network will progress to the new state through the in-between states to allow correct starting of the network.

PARAMETERS:

state:	EtherCAT state request	
	-1	Reserved
	0	Initial (EtherCAT ESC value 0x01)
	1	Pre-Operational (0x02)
	2	Safe-Operational (0x04)
	3	Operational (0x08)

display:	Function	
	1	Writes state change information to the standard output stream. (Default)
	0	Do not write out state change information.

EXAMPLE:

Change the EtherCAT to Safe-Operational and suppress the information that would be printed to the terminal.

```
ETHERCAT($21, 0, 2, 0)
```

FUNCTION = $22: GET ETHERCAT STATE

SYNTAX:

```
ETHERCAT($22, slot, vr_number)
```

DESCRIPTION:

Gets the present state of the EtherCAT running on the defined slot. The value returned shows the EtherCAT state as follows:

- 0 - Initial
- 1 - Pre-oprational
- 2 - Safe-Operational
- 3 - Operational

PARAMETERS:

vr_number:	The **VR** number where the returned value will be put. (-1 forces the value to be printed on the terminal)

EXAMPLE:

In the terminal, request the EtherCAT state value.

```
>>ETHERCAT($22, 0, -1)
3
>>
```

FUNCTION = $64: SEND RESET SEQUENCE TO A DRIVE

SYNTAX:

```
ETHERCAT($64, axis_number[, mode[, timeout]])
```

DESCRIPTION:

Reset a slave error. This function runs the error reset sequence on the drive control word. **DRIVE_CONTROLWORD** bit 8 is toggled high then low. This will instruct the drive to reset any errors in the drive where the cause of the error has been removed.

The response to a reset sequence will depend on the drive and how closely it follows the CoE DS402 specification.

PARAMETERS:

axis_number:	The axis number of the drive to be reset.	
mode:	0	The 'Fault Reset' (bit 7) of DS402 control word is set high and then set low again after a hard coded timeout. (default)
	1	Bit 7 is set high until the 'Fault Flag' (bit 3) of the status word goes low, or a timeout occurs.
timeout:	Optional timeout in msec used during mode 1 operation. Default is 100 msec. Range is 1 to 10000 msec.	

EXAMPLE:

EXAMPLE 1

Send control word reset sequence to drive at axis 8.

```
ETHERCAT($64, 8)
```

EXAMPLE 2

Send control word reset sequence to drive at axis 2. Use Mode 1 to force the reset bit to remain high until the status it 3 goes low or force the reset bit low again after 60 msec, even if the status bit is still high.

```
ETHERCAT($64, 2, 1, 60)
```

FUNCTION = $87: DISPLAY NETWORK CONFIGURATION

SYNTAX:

```
ETHERCAT($87, slot)
```

DESCRIPTION:

Displays the network configuration to the command line terminal in *Motion* Perfect.

PARAMETERS:

slot:	The slot number where the EtherCAT module is located

EXAMPLE:

In the terminal, request the EtherCAT network configuration.

```
>>ethercat($87,0)
EtherCAT Configuration (0):
    EK1100       :  0 : 0 : 2000
    EL2008       :  1 : 0 : 1000 (0:0/16:8)
    EL2008       :  2 : 0 : 1001 (0:0/24:8)
    EL2008       :  3 : 0 : 1002 (0:0/32:8)
    EL2008       :  4 : 0 : 1003 (0:0/40:8)
    EL2008       :  5 : 0 : 1004 (0:0/48:8)
    EK1110       :  6 : 0 : 2001
    RS2          :  7 : 0 : 1 (0)
    SGDV         :  8 : 0 : 2 (1)
>>
```

ETHERNET

TYPE:

System Command

SYNTAX:

`ETHERNET(rw, slot, function [,parameters…])`

DESCRIPTION:

The command **ETHERNET** is used to configure the operation of the Ethernet port.

Many of the **ETHERNET** functions are command line only; these are stored in flash EPROM and are then used on power up.

PARAMETERS:

rw:	Specifies the required action.	
	0	Read
	1	Write
slot:	Set to -1 for the built in Ethernet port	

function:	Function to be performed	
	0	IP Address
	1	Reserved function
	2	Subnet Mask
	3	MAC address
	4	Default Port Number
	5	Token Port Number
	6	PRP firmware version (read only)
	7	Modbus TCP mode
	8	Default Gateway
	9	Data configuration
	10	Modbus TCP port number
	11	ARP cache
	12	Reserved function
	13	Reserved function
	14	Configure endpoints for Modbus TCP or Ethernet IP

FUNCTION = 0: IP ADDRESS

SYNTAX:

```
ETHERNET(rw, slot, 0 [,byte1, byte2, byte3, byte4])
```

DESCRIPTION:

Prints or writes the Ethernet IP address. This is command line only.

You must power cycle the controller or perform **EX**(1) to apply the new **IP** address.

PARAMETERS:

byte1:	The first byte of the IP address
byte2:	The second byte of the IP address
byte3:	The third byte of the IP address

byte4:	The fourth byte of the IP address

The default address is 192.168.0.250

EXAMPLE:
Read the current IP address and then set a new IP address into the controller and perform an EX(1) to activate the address

Performing an **EX**(1) as in this example will close the communications and you will only be able to communicate again using the new **IP** address.

```
>>ETHERNET(0, -1, 0)
192.168.0.250
>>ETHERNET(1, -1, 0, 192, 168, 0, 201)
>>EX(1)
>>
```

FUNCTION = 2: SUBNET MASK

SYNTAX:
ETHERNET(rw, slot, 2 [,byte1, byte2, byte3, byte4])

DESCRIPTION:
Prints or writes the Subnet Mask. This is command line only.

You must power cycle the controller or perform **EX**(1) to apply the new **IP** address.

PARAMETERS:

byte1:	The first byte of the Subnet Mask
byte2:	The second byte of the Subnet Mask
byte3:	The third byte of the Subnet Mask
byte4:	The fourth byte of the Subnet Mask

The default Subnet Mask is 255.255.255.0

EXAMPLE:
Read the subnet mask and write a new value

```
>>ETHERNET(0, -1, 0)
255.255.255.0
>>ETHERNET(1, -1, 2, 255, 255, 128, 0)
>>
```

FUNCTION = 3: MAC ADDRESS

SYNTAX:

`ETHERNET(0, slot, 3)`

DESCRIPTION:

Prints the MAC address. This is command line only.

This function is read only.

PARAMETERS:

The MAC address is unique to your controller.

EXAMPLE:

Read the MAC address of a controller

```
>>ETHERNET(0, -1, 3)
00:06:70:00:00:FA
>>
```

FUNCTION = 4: DEFAULT PORT

SYNTAX:

`ETHERNET(rw, slot, 4 [, port])`

DESCRIPTION:

Prints or writes the default port number. This is command line only.

The default value is used by *Motion* Perfect and PCMotion and should not be changed unless absolutely necessary.

PARAMETERS:

port:	The port used for the main command line in the controller. (default 23)

FUNCTION = 5: TOKEN PORT

SYNTAX:

```
ETHERNET(rw, slot, 5 [, port])
```

DESCRIPTION:

Prints or writes the default port number for token channel which is used by the PCMotion ActiveX control. This is command line only.

The default value is used by the PCMotion ActiveX control and should not be changed unless absolutely necessary.

PARAMETERS:

port:	The port used for the token channel in the controller. (default 3240)

FUNCTION = 6: PRP FIRMWARE VERSION (READ ONLY)

SYNTAX:

```
Ethernet(0,slot,6)
```

DESCRIPTION:

Reads the communications processor s firmware version. This is command line only.

This function is read only

PARAMETERS:

Returns the flash application version and the bootloader version.

EXAMPLE:

Read the communications processor firmware with application version 61 and boot loader version 22.

```
>>ETHERNET(0, -1, 6)
61;22
>>
```

FUNCTION = 7: MODBUS TCP MODE

SYNTAX:

```
Ethernet(rw, slot, 7 [,mode])
```

DESCRIPTION:

Sets the Modbus TCP data type. This value is stored in RAM and so must be initialised every time the controller powers up. This can be done in a TrioBASIC program for example **STARTUP**

This must be configured before the Modbus master opens the port.

PARAMETERS:

mode:	0	16bit integer (default value)
	1	32bit single precision floating point without address halving
	2	32bit long word integers without address halving

If you want to use address halving please see **ETHERNET** Function 14

EXAMPLE:

Initialise the Modbus TCP port for floating point data.

```
ETHERNET(1,-1,7,1)
```

FUNCTION = 8: DEFAULT GATEWAY

SYNTAX:

```
ETHERNET(rw, slot, 8 [,byte1, byte2, byte3, byte4])
```

DESCRIPTION:

Prints or writes the Default Gateway. This is command line only.

You must power cycle the controller or perform **EX**(1) to apply the new Default Gateway.

PARAMETERS:

byte1:	The first byte of the Default Gateway
byte2:	The second byte of the Default Gateway

byte3:	The third byte of the Default Gateway
byte4:	The fourth byte of the Default Gateway

EXAMPLE:

Print then change the value of the default gateway.

```
>>ETHERNET(0, -1, 8)
192.168.0.225
>> ETHERNET(0,-1, 8, 192, 168, 0, 150)
>>
```

FUNCTION = 9: DATA CONFIGURATION

SYNTAX:

```
Ethernet(rw, slot, 9 [,mode])
```

DESCRIPTION:

Sets the Modbus TCP data source. This value is stored in RAM and so must be initialised every time the controller powers up. This can be done in a TrioBASIC program for example **STARTUP**

This must be configured before the Modbus master opens the port.

PARAMETERS:

mode:	0	VR (default value)
	1	Table

EXAMPLE:

Initialise the Modbus TCP port for table data.

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

FUNCTION = 10: MODBUS TCP PORT NUMBER

SYNTAX:

```
ETHERNET(rw, slot, 10 [, port])
```

DESCRIPTION:

Prints or writes the default port number for token channel which is used by Modbus TCP. This is command line only.

The default value is used by Modbus and should not be changed unless absolutely necessary.

PARAMETERS:

port:	The port used for the token channel in the controller. (default 502)

FUNCTION = 11: ARP CACHE

SYNTAX:

```
Ethernet(0, slot, 11)
```

DESCRIPTION:

Reads the ARP cache. This is command line only.

This function is read only

FUNCTION = 14: ENDPOINTS FOR MODBUS TCP OR ETHERNET IP

SYNTAX:

```
ETHERNET(1, slot, 14, endpoint_id, parameter_index, parameter_value )
```

DESCRIPTION:

This function allows the user to configure Ethernet IP and Modbus at a low level. The default values allow a master to connect without any configuration on the Controller side. These settings are stored in RAM and so must be initialised every time the controller powers up. This can be done in a TrioBASIC program for example **STARTUP**.

PARAMETERS:

endpoint_id:	This allows you to specify which end point you are reading or writing	
	0	Modbus TCP
	1	Ethernet IP Assembly Object, Instance 100 (input)
	2	Ethernet IP Assembly Object, Instance 101 (output)

parameter_index:	This parameter selects which of the endpoint variables you are reading or writing	
	0	Address
	1	Data location
	2	Data format
	3	Length
	4	Class
	5	Instance
	6	Operation Mode
parameter_value:	Dependent on Parameter index, see table below	

PARAMETER VALUES:

parameter_index	parameter_value	
0	The start position of the data location.	
1	The location of the data on the controller.	
	0	Register (reserved use)
	1	IO input
	2	IO output
	3	VR (default value)
	4	Table
	5	Digital IO Input
	6	Digital IO Output
	7	Analogue IO Input
	8	Analogue IO Input
2	The precision of the data.	
	0	Integer 16 bit (default value)
	1	Integer 32 bit
	2	Floating point 32 bit
	3	Floating point 64 bit

3	The number of the data locations returned.	
4	The class. This function is read only.	
	4	Ethernet IP
	68	Modbus
5	The instance of the endpoint. This function is read only.	
	0	Modbus
	100	Ethernet IP input
	101	Ethernet IP output
6	The Operation mode. Read/write.	
	0	Modbus TCP uses normal addressing
	1	Modbus TCP uses “address halving”

EXAMPLES:

EXAMPLE 1:

Configure Modbus using Function 14 to use Table and floating point 64bit

```
ETHERNET(1, -1, 14, 0, 1, 4)
ETHERNET(1, -1, 14, 0, 2, 3)
```

EXAMPLE 2:

Configure Ethernet IP for 50 **TABLE** inputs starting at 200 and 50 table outputs starting at 300 all at 32bit float

```
‘Inputs
ETHERNET(1, -1, 14, 1,0,200)
ETHERNET(1, -1, 14, 1, 1, 4)
ETHERNET(1, -1, 14, 1, 2, 2)
ETHERNET(1, -1, 14, 1, 3, 50)
‘Outputs
ETHERNET(1, -1, 14, 2,0,300)
ETHERNET(1, -1, 14, 2, 1, 4)
ETHERNET(1, -1, 14, 2, 2, 2)
ETHERNET(1, -1, 14, 2, 3, 50)
```

EXAMPLE 3:

Configure Modbus TCP floating point **TABLE** access, using address halving to match the addressing scheme used in the master.

```
ETHERNET(1, -1, 14, 0,2,2)
```

```
ETHERNET(1, -1, 14, 0, 1, 4)
ETHERNET(1, -1, 14, 0, 6, 1)
```

EX

TYPE:
System Command

SYNTAX:
`EX(processor)`

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

When performing an **EX** on the command line you will see the controller start up information that provides details of your controller configuration.

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 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.

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

PARAMETERS:

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

When you use **EX**(1) you will have to remake the Ethernet connection

EXECUTE

TYPE:
System Command

DESCRIPTION:
Used to implement the remote command execution via the Trio PCMotion ActiveX. For more details see the section on using the PCMotion

EXP

TYPE:
Mathematical Function

SYNTAX:
EXP(expression)

DESCRIPTION:
Returns the exponential value of the expression.

PARAMETERS:

expression:	Any valid TrioBASIC expression

EXAMPLE:
Print the expontential value of 1

```
>>PRINT EXP(1)
2.7183
```

>>

FALSE

TYPE:
Constant

DESCRIPTION:
The constant **FALSE** takes the numerical value of 0.

EXAMPLE:

```
test:
Use FALSE as part of a logical check
  res = IN(0) OR IN(2)
  IF res = FALSE THEN
    PRINT “Inputs are off”
  ENDIF
```

FAST_JOG

TYPE:
Axis Parameter

DESCRIPTION:
This parameter holds the input number to be used as the fast jog input. If the **FAST_JOG** is active then the jog inputs use the axis **SPEED** for the jog functions, otherwise the **JOGSPEED** will be used.

The input used for **FAST_JOG** is active low.

VALUE:

-1	disable the input as **FAST_JOG** (default)
0-63	Input to use as datum input

Any type of input can be used, built in, Trio CAN I/O, CANopen or virtual.

EXAMPLE:
Configure input 12 and 13 as jog inputs

```
FWD_JOG = 12
FAST_JOG = 13
JOGSPEED = 200
```

SEE ALSO:
FWD_JOG, JOGSPEED, REV_JOG

FASTDEC

TYPE:
Axis Parameter

DESCRIPTION:
The **FASTDEC** axis parameter may be used to set or read back the fast deceleration rate of each axis fitted. Fast deceleration is used when a **CANCEL** is issued, for example; from the user, a program, or from a software or hardware limit. If the motion finishes normally or **FASTDEC** = 0 then the **DECEL** value is used.

VALUE:
The deceleration rate in **UNITS**/sec/sec. Must be a positive value.

EXAMPLE:
```
    DECEL=100               'set normal deceleration rate
    FASTDEC=1000            'set fast deceleration rate
    MOVEABS(10000)          'start a move
    WAIT UNTIL MPOS= 5000   'wait until the move is half finished
    CANCEL                  'stop move at fast deceleration rate
```

SEE ALSO:
DECEL

FE

TYPE:
Axis Parameter (Read Only)

DESCRIPTION:
This parameter returns the position error, which is equal to the demand position (**DPOS**) - measured position (**MPOS**).

VALUE:
The following error returned in user **UNITS**.

EXAMPLE:

Wait for the position error to be below a value for 5 servo periods then pulse an output.

```
MOVEABS(200)
WAIT IDLE
FOR x=0 to 4
    WAIT UNTIL FE<5
NEXT x
OP(5,ON)
WA(2)
OP(5,OFF)
```

SEE ALSO:

FE_LATCH, FE_LIMIT, FE_RANGE

FE_LATCH

TYPE:

Axis Parameter (Read Only)

DESCRIPTION:

Contains the FE value which caused the axis to put the controller into **MOTION_ERROR**. This value is only set when the FE exceeds the **FE_LIMIT** and the **SERVO** = OFF.

VALUE:

Returns the FE value that caused a **MOTION_ERROR**

FE_LATCH is reset to 0 when the axis **SERVO** = **ON**.

EXAMPLE:

Read the **LE_LATCH** when there is a **MOTION_ERROR**

```
IF MOTION_ERROR THEN
  VR(10) = FE_LATCH AXIS (ERROR_AXIS)
ENDIF
```

SEE ALSO:

FE, FE_LIMIT

FE_LIMIT

TYPE:
Axis Parameter

ALTERNATE FORMAT:
FELIMIT

DESCRIPTION:
This is the maximum allowable following error. When exceeded the controller will generate an **AXISSTATUS** error, by default this will also generate a **MOTION_ERROR**. The **MOTION_ERROR** will disable the **WDOG** relay thus stopping further motor operation.

This limit may be used to guard against fault conditions such as mechanical lock-up, loss of encoder feedback, etc.

VALUE:
The maximum allowable following error in user units. The default value is 2000 encoder edges.

EXAMPLE:
Initialise the axis as part of a **STARTUP** routine

```
FOR x = 0 to 4
  BASE(x)
  UNITS = 100
  FE_LIMIT = 10
  SPEED = 100
  ACCEL=1000
  DECEL=ACCEL
NEXT x
```

SEE ALSO:
FE, FE_LATCH

FE_LIMIT_MODE

TYPE:
Axis Parameter

DESCRIPTION:
This parameter determines if an **AXISSTATUS** error is produced immediately when the FE exceeds the

FE_LIMIT or if it exceeds for 2 consecutive servo periods. This means that if **FE_LIMIT** is exceeded for one servo period only, it will be ignored.

This will increase the time to disable your drives in an error. You should only change from the default values under advice from Trio or your distributor.

VALUE:

0	**AXISSTATUS** error generated immediately (default)
1	**AXISSTATUS** error generated when **FE_LIMIT** is exceeded for 2 consecutive servo periods.

SEE ALSO:
FE, FE_LIMIT

FE_RANGE

TYPE:
Axis Parameter

DESCRIPTION:
Following error report range. When the FE exceeds this value the axis has bit 1 in the **AXISSTATUS** axis parameter set.

VALUE:
The value in user **UNITS** above which bit 1 is set in **AXISSTATUS**

EXAMPLE:
Using **FE_RANGE** to slow a machine down when the FE is too large.

```
'initialise the axis
FE_RANGE = 10
FE_LIMIT = 15
SPEED=100
…
'loop to check if FE_RANGE has been exceeded
WHILE NOT IDLE
VR(10) = AXISSTATUS
IF READBIT(1, 10) THEN
  'slow down by 1%
  SPEED = SPEED * 0.99
ENDIF
```

```
WEND
SPEED = 100
```

SEE ALSO:

FE, FE_LIMIT

FEATURE_ENABLE

TYPE:

System Command

SYNTAX:

FEATURE_ENABLE([feature _number [, “password”]])

DESCRIPTION:

*Motion Coordinator*s have the ability to unlock additional features by entering a “Feature Enable Code”. This function is used to enable protected features, such as additional remote axes on digital dive networks or other programming languages. This can only be run on the command line.

It is recommended to use *Motion* Perfect to enter and store the feature enable codes.

The password parameter is optional, if it is omitted then the command will prompt you to enter it.

You can purchase additional feature codes from the Trio Website or through your distributor, you will need the **SERIAL_NUMBER** of the controller.

If you enter the wrong password 3 times the controller will enter an attack state where it stops communicating. You can resume normal operation by power cycling the controller.

PARAMETERS:

feature_number:	None	Prints the security code and currently enabled features.
	0	1 remote axis
	1	2 remote axes
	2	4 remote axes
	3	8 remote axes
	4	16 remote axes
	5	32 remote axes
	6-11	Reserved use
	12	1 remote axis
	13	2 remote axes
	14	4 remote axes
	15	8 remote axes
	16	16 remote axes
	17	32 remote axes
	18-20	Reserved use
	21	IEC runtime
	22-31	Axis upgrade
	24-31	Reserved use
password:	The password for the required feature code	

When entering a feature a password is requested

When entering a password 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.

EXAMPLES:

EXAMPLE 1:

Check the enabled features on a controller

```
>>FEATURE_ENABLE
Security code=17980000000028
Enabled features: 0 1
```

Features 0 and 1 are enabled so an additional 3 axes on top of the built in axes included with the module.

EXAMPLE 2:

Enable an additional 4 axes (feature 2). For this controller and this feature, the password is 5P0APT.

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

SEE ALSO:

SERIAL_NUMBER

FHOLD_IN

TYPE:

Axis Parameter

ALTERNATE FORMAT:

FH_IN

DESCRIPTION:

This parameter holds the input number to be used as a feedhold input.

When the feedhold input is active motion on the specified axis has its speed overridden to the feedhold speed (**FHSPEED**) without canceling the move in progress. The change in speed uses **ACCEL** and **DECEL**. When the input is reset any move in progress when the input was set will go back to the programmed speed.

Set **FHSPEED** to zero to pause the motion on that axis

Moves which are not speed controlled e.g. **CONNECT**, **CAMBOX**, **MOVELINK** are not affected.

The input used for **FHOLD_IN** is active low.

VALUE:

-1	disable the input as feedhold (default)
0-63	Input to use as feedhold

Any type of input can be used, built in, Trio CAN I/O, CANopen or virtual.

EXAMPLE:
Configure inputs 21 as feedhold inputs for axis 2. The default **FHSPEED** = 0 so the motion can be paused using the feedhold input.

SEE ALSO:
FHSPEED

FHSPEED

TYPE:
Axis Parameter

DESCRIPTION:
When the feedhold input is active motion is ramped down to **FHSPEED**.

VALUE:
The speed in user units to use when the **FHOLD_IN** is active (default 0)

EXAMPLE:
Set **FHSPEED** to a value so that a slower speed is selected wen the **FHOLD_IN** is active

```
BASE(3)
SPEED=1000
FHSPEED=SPEED*0.1
```

SEE ALSO:
FHOLD_IN

FILE

TYPE:
System Command

SYNTAX:
```
value = FILE “function” [parameters]
```

DESCRIPTION:

This command enables the user to manage the data on the SD Card.

When the command prints to the selected channel, this channel can be selected using **OUTDEVICE**

PARAMETERS:

function:	CD	Change directory
	DEL	Delete file
	DETECT	Check for SD Card
	DIR	Print the current directory contents
	FIND_CLOSE	Ends the find session
	FIND_FIRST	Finds the first entry in the directory structure of the specified file type
	FIND_NEXT	Finds the next entry in the directory structure of the specified file type
	FIND_PREV	Finds the previous entry in the directory structure of the specified file type
	LOAD_PROGRAM	Loads the specified program to the controllers memory
	LOAD_PROJECT	Loads the specified project into the controllers memory
	LOAD_SYSTEM	Loads the specified firmware into the controller
	RD	Remove (delete) a directory
	MD	Make (create) a directory
	PWD	Prints the path of the directory
	SAVE_PROGRAM	Saves the specified program to the SD Card
	SAVE_PROJECT	Saves all programs from the controller to the SD Card.
	TYPE	Prints the selected file
parameters:	dependent on the function	
value:	returns **TRUE** if the function was successful otherwise returns **FALSE**	

FUNCTION = CD:

SYNTAX:
`value = FILE "CD" "directory"`

DESCRIPTION:
Change to the given directory. There is one active directory on the controller all SD Card commands are relative to this directory.

PARAMETERS:

directory:	string	The name of the child directory to move to
	\\	Move to the root directory
	..	Move up one level to the parent directory

EXAMPLES:

EXAMPLE 1
Use the command line to change to a new directory

```
>>file "CD" "new_directory"
OK \NEW_DIRECTORY
>>
```

EXAMPLE 2
Use the command line to change to a new directory 3 levels below

```
>>file "CD" " project1\\project2\\project3"
OK \PROJECT1\PROJECT2\PROJECT3
>>
```

EXAMPLE 3
Use the command line to move to the root directory

```
>>file "CD" "\\"
OK \
>>
```

FUNCTION = DEL:

SYNTAX:
value = **FILE** "DEL" "file"

DESCRIPTION:
Delete the given file inside the current directory.

PARAMETERS:

file:	The name of the file to be deleted, you must include the file extension

EXAMPLE:
Delete a **BASIC** program from the SD Card using the command line.

```
>>FILE “DEL” “STARTUP.bas”
OK
>>
```

FUNCTION = DETECT:

SYNTAX:
`value = FILE “DETECT”`

DESCRIPTION:
Checks if a SD Card is present in the slot

RETURN VALUE:
TRUE if an SD Card is detected correctly, otherwise **FALSE**.

EXAMPLE:
Check if an SD card is present before saving the table data.

```
IF FILE “DETECT” THEN
  STICK_WRITE(1501, 1000, 2000, 0)
ENDIF
```

FUNCTION = DIR:

SYNTAX:
`value = FILE “DIR”`

DESCRIPTION:
Print the contents of the current directory to the current output channel.

EXAMPLE:

Print the contents of the SD card on the command line.

```
>>FILE "DIR"
 Volume is NO NAME
 Volume Serial Number is 00C8-B79F
 Directory of \
07/Aug/2009 15:50    1169978 MC60CC~1.OUT MC464_20055__BOOT_013.out
20/Nov/2009 15:25 <DIR>      MC464_~1     MC464_Panasonic_Home
16/Feb/2009 13:16       1619 TRIOINIT.BAS TRIOINIT.BAS
20/Nov/2009 15:21 <DIR>      SHOW1        Show1
07/Jan/2000 04:54 <DIR>      NEW_DI~1     NEW_DIRECTORY
>>
```

FUNCTION = FIND_CLOSE:

SYNTAX:

value = FILE "FIND_CLOS"

DESCRIPTION:

Closes the internal **FIND** structure. Use when you have finished with **FIND_NEXT** and **FIND_PREVIOUS**.

FUNCTION = FIND_FIRST:

SYNTAX:

value = FILE "FIND_FIRST", type, vr_index

DESCRIPTION:

Initialises the internal **FIND** structures and locates the first directory entry of the given type. The found directory entries name is stored in a **VRSTRING**

PARAMETERS:

value:	**TRUE** if a directory entry is found otherwise **FALSE**	
type:	1	**FILE**
	2	**DIRECTORY**
vr_index:	The start position in **VR** memory where the **VRSTRING** is stored	

If there is an error initialising the internal **FIND** structures then the function returns **FALSE**.

FUNCTION = FIND_NEXT:

SYNTAX:

```
value = FILE "FIND_NEXT", vr_index
```

DESCRIPTION:

Finds the next directory entry of the type given in the corresponding **FIND_FIRST** command.

PARAMETERS:

value:	**TRUE** if a directory entry is found otherwise **FALSE**
vr_index:	The start position in **VR** memory where the **VRSTRING** is stored

If there is an error initialising the internal **FIND** structures then the function returns **FALSE**.

FUNCTION = FIND_PREV:

SYNTAX:

```
value = FILE  "FIND_PREV", vr_index
```

DESCRIPTION:

Finds the previous directory entry of the type given in the corresponding **FIND_FIRST** command.

PARAMETERS:

value:	**TRUE** if a directory entry is found otherwise **FALSE**
vr_index:	The start position in **VR** memory where the **VRSTRING** is stored

If there is an error initialising the internal **FIND** structures then the function returns *FALSE*.

FUNCTION = LOAD_PROGRAM:

SYNTAX:

```
value = FILE  "LOAD_PROGRAM" "file"
```

DESCRIPTION:
Load the given program into the *Motion Coordinator*. Only .BAS files are handled at present.

PARAMETERS:

file:	The name of the file that you wish to load.

FUNCTION = LOAD_PROJECT:

SYNTAX:
`value = FILE "LOAD_PROJECT" "name"`

DESCRIPTION:
Read the given *Motion* Perfect project file and load all the programs into the *Motion Coordinator*, once loaded any RUNTYPEs are automatically set.

PARAMETERS:

name:	The name of the project that you wish to load.

FUNCTION = LOAD_SYSTEM:

SYNTAX:
`value = FILE "LOAD_SYSTEM" "name"`

DESCRIPTION:
Loads system firmware onto the controller.

PARAMETERS:

name:	The name of the firmware file that you wish to load.

Loading incorrect firmware can prevent your controller from operating

FUNCTION = RD:

SYNTAX:
`value = FILE "RD" "name"`

DESCRIPTION:
Delete the given directory inside the current directory.

PARAMETERS:

name:	The name of the directory that you wish to delete.

FUNCTION = MD:

SYNTAX:
value = FILE "MD" "name"

DESCRIPTION:
Create the given directory inside the current directory.

PARAMETERS:

name:	The name of the directory that you wish to create.

EXAMPLE:
Using the command line create a new directory.

```
>>FILE "MD" "new_directory"
OK
>>
```

FUNCTION = PWD:

SYNTAX:
value = FILE "PWD"

DESCRIPTION:
Prints the path of the current directory to the current output channel.

FUNCTION = SAVE_PROGRAM:

SYNTAX:
value = FILE "SAVE_PROGRAM" "name" ["extension"]

DESCRIPTION:
Save the named file from the controllers memory to the SD card. If the extension is omitted then the default file extensions BAS, TXT or **TEMP** are used.

PARAMETERS:

name:	The name of the file that you wish to save to the SD Card.
extension:	Optional to define the file extension to be used

FUNCTION = SAVE_PROJECT:

SYNTAX:
value = FILE "SAVE_PROJECT" "name"

DESCRIPTION:
Create a *Motion* Perfect project with the given name inside the current directory. This implies creating the directory and the corresponding project and program files within this directory.

PARAMETERS:

name:	The name of the project that you are creating on the SD Card

FUNCTION = TYPE:

SYNTAX:
value = FILE "TYPE" "name"

DESCRIPTION:
Read the contents of the file inside the current directory and print it to the current output channel.

PARAMETERS:

name:	The name of the file that you wish to print

SEE ALSO
OUTDEVICE, STICK_READ, STICK_WRITE, STICK_READVR, STICK_WRITEVR

FILLET

TYPE:

Mathematical function

SYNTAX:

FILLET(data_in, data_out, options)

DESCRIPTION:

The **FILLET** function has 2 calculation functions:

The first function allows the dimensions of an arc that fillets or blends two 3-D vectors together to be easily calculated.

The second function allows the dimensions of two 2D arcs that blends 2 points with directions to be easily calculated.

PARAMETERS:

data_in:	Location of the input data in **TABLE** memory.	
data_out:	Location of the output data in **TABLE** memory.	
options:	0	Used to calculate the arc between 2 straight lines in 3D.
	1	Calculates a pair of arcs between 2 points with directions.

OPTION = 0

DESCRIPTION:

The function calculates the start, end, midpoint and centre of the 3D arc. The arc may easily be converted into motion using the **MSPHERICAL** command.

FILLET only works in system version 2.0220 and higher which outputs 19 data values including the fillet angle and fillet length.

PARAMETERS:

Input data: (7 data values required)

Table data	0	x vector A
	1	y vector A
	2	z vector A
	3	x vector B
	4	y vector B
	5	z vector B
	6	radius

Output data: (19 data values are output)

Table data	0	x A remain
	1	y A remain
	2	z A remain
	3	end x
	4	end y
	5	end z
	6	mid x
	7	mid y
	8	mid z
	9	centre x
	10	centre y
	11	centre z
	12	error
	13	output radius
	14	x B remain
	15	y B remain
	16	z B remain
	17	angle change
	18	fillet length

A remain: the xyz position of the start of arc relative to the start of the incoming vector.

Mid: the xyz position of a mid-point on the fillet arc relative to the start of arc.

Centre: the xyz position of the arc centre relative to the start of arc.

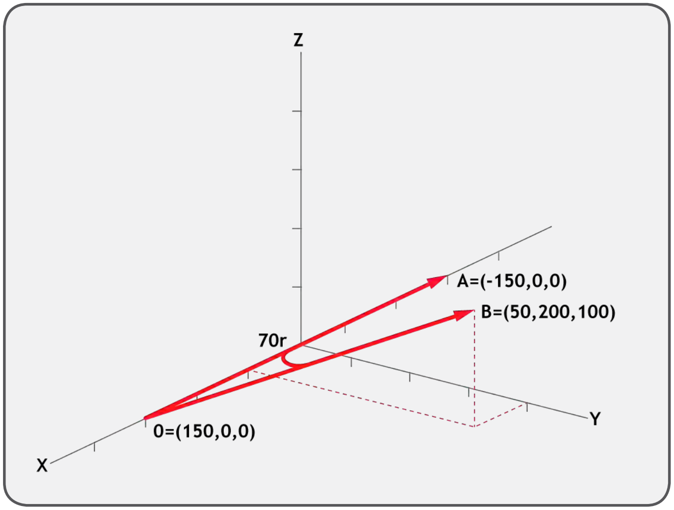
Error: set to 0 if no error, 1 = one or both vectors is zero length, 2 = vectors are co-linear.

Output radius: If the vectors are not long enough to allow the requested radius to be filleted (taking into account the options value) the output radius value will show the maximum possible otherwise will reflect the input radius.

B remain: the xyz position of the end of the outgoing vector relative to the end of the arc.

EXAMPLE:

Calculate the fillet of two 3D vectors and represent them by **MOVE** command for the vectors and **MSPHERICAL** for the fillet.




```
DEFPOS(150,0,0)

TRIGGER TABLE(100,-150,0,0)
TABLE(103,50,200,100,70)

FILLET(100,200,0)

xin=TABLE(200):yin=TABLE(201):zin=TABLE(202)

MOVE(xin,yin,zin)

xend=TABLE(203):yend=TABLE(204):zend=TABLE(205) xmid=TABLE(206):ymid=TA
```

```
BLE(207):zmid=TABLE(208)

MSPHERICAL(xend,yend,zend,xmid,ymid,zmid,0)

xout=TABLE(214):yout=TABLE(215):zout=TABLE(216)


MOVE(xout,yout,zout)

fillet_ang=TABLE(217):fillet_len=TABLE(218)

PRINT fillet_ang,fillet_len
```

OPTION = 1

DESCRIPTION:

The function calculates the start, end and centre of 2 arcs. The arc may be easily converted into motion using **MOVECIRC** or **MSPHERICAL** commands.

PARAMETERS:

Input data: (10 data values required).

Table data	0	X value point A
	1	Y value point A
	2	X direction point A
	3	Y direction point A
	4	X value point B
	5	Y value point B
	6	X direction point B
	7	Y direction point B
	8	Radius control (Set to 0 to allow **FILLET** to calculate the largest possible radius)
	9	Arc direction mode control: 0 - Use shortest route 1 - **LEFT TURN** – **LEFT TURN** arc forced 2 - **RIGHT TURN** – **RIGHT TURN** arc forced 3 - **LEFT TURN** then **RIGHT TURN** arc forced 4 - **RIGHT TURN** then **LEFT TURN** arc forced

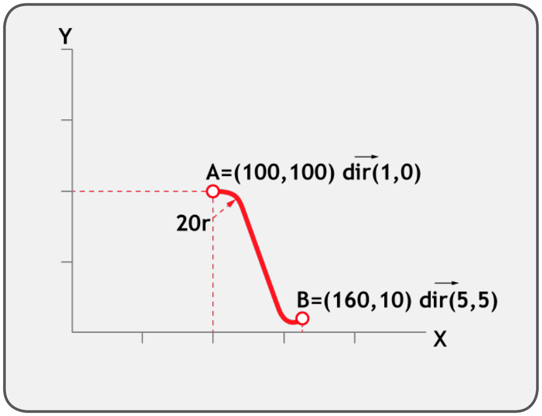
The direction at a point is specified using a pair of +/- incremental values. This need not be normalised to a length of 1 by the user. For example a direction along the X axis can be specified as (1, 0) a direction in the negative X direction would be (-1, 0). A direction along the Y axis would be (0, 1). Considering an angle to be the +/-PI angle from the Y axis. The direction is (sin(angle), cos(angle)).

Output data: (18 data values are output)

Table data	0	Bit 0 – Arc A Direction Bit 1 – Arc B Direction
	1	1 – **LEFT TURN** arc A then **LEFT TURN** arc B 2 – **RIGHT TURN** arc A then **RIGHT TURN** arc B 3 - **LEFT TURN** arc A then **RIGHT TURN** arc B 4 – **RIGHT TURN** arc A then **LEFT TURN** arc B
	2	X end position relative to start arc A
	3	Y end position relative to start arc A
	4	X centre position relative to start arc A
	5	Y centre position relative to start arc A
	6	X increment linking linear move (0 if radius unlimited)
	7	Y increment linking linear move (0 if radius unlimited)
	8	X end position relative to start arc B
	9	Y end position relative to start arc B
	10	X centre position relative to start arc B
	11	Y centre position relative to start arc B
	12	Error, 0 = no error
	13	Arc A Length
	14	Linking Move Length
	15	Arc B Length
	16	Total Length
	17	Radius calculated. If the radius is limited by the “radius control” input this value will be set to the limit radius.

EXAMPLE:

Calculate the dimensions of two arcs that blends two points with directions and represent them by **MCIRCLE** command.




```
max_r=20
dir_o=0

TABLE(3000,100,100,1,0,160,10,5,5,max_r,dir_o)

FILLET(3000,3200,1)

IF TABLE(3212) THEN
    PRINT "Error in data"
    STOP
ENDIF

direc1=TABLE(3200).0
direc2=TABLE(3200).1

end1x = TABLE(3202)
end1y = TABLE(3203)
cen1x = TABLE(3204)
cen1y = TABLE(3205)

px = TABLE(3206)
py = TABLE(3207)

end2x = TABLE(3208)
end2y = TABLE(3209)
cen2x = TABLE(3210)
cen2y = TABLE(3211)

arc1len = TABLE(3213)
midlen = TABLE(3214)
arc2len = TABLE(3215)

TRIGGER
```

```
IF arc1len>0 THEN MOVECIRC(end1x,end1y,cen1x,cen1y,direc1)
IF midlen >0 THEN MOVE(px,py)
IF arc2len>0 THEN MOVECIRC(end2x,end2y,cen2x,cen2y,direc2)

WAIT IDLE
```

FLAG

TYPE:

Logical and Bitwise Command

SYNTAX:

value = FLAG(flag_no [,state])

DESCRIPTION:

The **FLAG** command is used to set and read a bank of 24 flag bits.

The **FLAG** command is provided to aid compatibility with earlier controllers and is not recommended for new programs.

PARAMETERS:

value:	With one parameter it returns the state of the flag
	With 2 parameters it returns -1
flag_no:	The flag number is a value from 0..31.
state:	The state to set the given flag to. ON or OFF.

EXAMPLE:

Toggle a flag depending on a **VR** value

```
IF FLAG(21) and VR(100)=123 THEN
  FLAG(21,OFF)
ELSE IF NOT FLAG(21) and VR(100)<>123 THEN
  FLAG(21,ON)
ENDIF
```

FLAGS

TYPE:
Logical and Bitwise Command

SYNTAX:
value = FLAGS([state])

DESCRIPTION:
Read or Set the 32bit **FLAGS** as a block.

The **FLAGS** command is provided to aid compatibility with earlier controllers and is not recommended for new programs.

PARAMETERS:

value:	no parameters = returns the status of all flag bits
	with parameter = returns -1
state:	The decimal equivalent of the bit pattern to set the flags to

EXAMPLES:

EXAMPLE 1:
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
```

FLASH_DATA

TYPE:
Startup Parameter (**MC_CONFIG**)

DESCRIPTION:

FLASH_DATA controls whether **VR** or **TABLE** data is automatically backed up to flash memory.

The default setting (0) will use **VR** memory as the source for backup. However, by changing this parameter to 1 within **MC_CONFIG** will cause **TABLE** data as the source for backup. Please note that regardless of which data source is selected , only the first 4096 elements will be available for automatic backup.

VALUE:

0	VR memory selected for automatic backup (default)
1	**TABLE** memory selected for automatic backup

EXAMPLES:

EXAMPLE 1:

FLASH_DATA = 0 'Select **VR** memory for backup

EXAMPLE 2:

FLASH_DATA = 1 'Select **TABLE** memory for backup

FLASH_DUMP

TYPE:

Reserved Keyword

FLASHTABLE

TYPE:

System Function

SYNTAX:

FLASHTABLE(function,flashpage,tablepage)

DESCRIPTION:

Copies user data in RAM to and from the permanent **FLASH** memory.

If **FLASHTABLE** is being used then you cannot use **FLASHVR**(-1)

PARAMETERS:

function:	Specifies the required action.	
	1	Write a page of **TABLE** data into flash EPROM.
	2	Read a page of flash memory into **TABLE** data.
flashpage:	The index number (0 ... 31) of a 16000 values page of Flash EPROM where the table data is to be stored to or retrieved from.	
tablepage:	The index number (0 ... INT(**TSIZE**/16000)) of the page in table memory where the data is to be copied from or restored to.	

EXAMPLE:

Save the **TABLE** page 2 data in locations **TABLE**(32000) -**TABLE**(47999) to **FLASH** memory page 5.

```
FLASHTABLE(1,5,2)
```

SEE ALSO:

FLASHVR

FLASHVR

TYPE:

System Function

SYNTAX:

FLASHVR(function)

DESCRIPTION:

Copies user **VR** or **TABLE** data in RAM to and from the permanent **FLASH** memory.

If **FLASHVR**(-1) is being used then you cannot use **FLASHTABLE**

PARAMETERS:

function:	Specifies the required action.	
	-1	Stores the entire **TABLE** to the Flash EPROM and use it to replace the RAM table data on power-up.
	-2	Stop using the EPROM copy of table during power-up.
	-100	Force all changed **VR**'s to be committed to Flash EPROM (non battery backed controllers only)

After using function -1, any changed table data will be overwritten on the next power up or reset.

EXAMPLE:

Save the entire **TABLE** data to **FLASH** memory.

```
FLASHVR(-1)
```

SEE ALSO:

FLASHTABLE

FLEXLINK

TYPE:

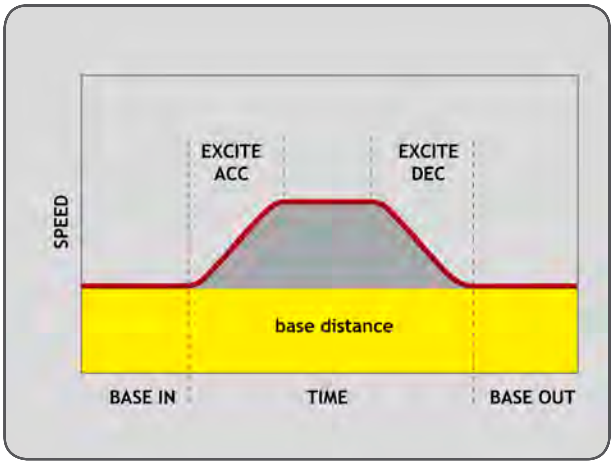
Axis Command

SYNTAX:

FLEXLINK(base_dist, excite_dist, link_dist, base_in, base_out, excite_acc, excite_dec, link_axis, options, start_pos)

DESCRIPTION

The **FLEXLINK** command is used to generate movement of an axis according to a defined profile. The motion is linked to the measured motion of another axis. The profile is made up of 2 parts, the base move and the excitation move both of which are specified in the parameters. The base move is a constant speed movement. The excitation movement uses sinusoidal profile and is applied on top of the base movement.



This command allows you to simplify a **CAMBOX** type movement through not having to use any table data.

PARAMETERS:

base_dist:	The distance the axis should move at a constant speed		
excite_dist:	The distance the axis should perform the profiled move		
link_dist:	The distance the link axis should move while the **FLEXLINK** profile executes		
base_in:	The percentage of the base move time that completes before the excitation move starts		
base_out:	The percentage of the base move time that completes after the excitation move completes.		
excite_acc:	The percentage of the excitation move time used for acceleration		
excite_dec:	The percentage of the excitation move time used for deceleration.		
link_axis:	The axis to link to.		
link_options:	Bit value options to customize how your **FLEXLINK** operates		
	Bit 0	1	link commences exactly when registration event **MARK** occurs on link axis
	Bit 1	2	link commences at an absolute position on link axis (see link_pos for start position)
	Bit 2	4	**FLEXLINK** 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)
	Bit 5	32	Link is only active during a positive move on the link axis
	Bit 8	256	link commences exactly when registration event **MARKB** occurs on link axis
	Bit 9	512	link commences exactly when registration event **R_MARK** occurs on link axis. (see link_pos for channel number)
link_pos:	link_option bit 1 - the absolute position on the link axis in user **UNITS** where the **FLEXLINK** is to start. link_option bit 9 – the registration channel to start the movement on		

The link_dist is in the user units of the link axis and should always be specified as a positive distance.

The link options for start (bits 1, 2, 8 and 9) may be combined with the link options for repeat (bits 4 and 8) and direction.

start_pos cannot be at or within one servo period's worth of movement of the **REP_DIST** position.

EXAMPLES:

EXAMPLE 1:

Suppose you want a smooth curve for 40% of a cycle and to remain stationary for the remainder:

```
FLEXLINK(0,10000,20000,60,0,50,50,1)
```

In this example the move length is 10000 and this is linked to 20000 distance on the link axis (1). The axis is stationary for 60% of the cycle and the move is 50% accel/50% decel.

EXAMPLE 2:

Suppose you want a 1:1 background link but to advance 500 using a smooth curve between 80% and 95% of a cycle:

```
FLEXLINK(10000,500,10000,80,5,50,50,1)
```

In this example the base move length is 10000 and this is linked to 10000 distance on the link axis (1). The excite distance is 500 and this starts after 80% of the cycle, with 5% at the end also clear of excitation. The "excite" move is 50% accel/50% decel.

FOR..TO.. STEP..NEXT

TYPE:

Program Structure

SYNTAX:

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

DESCRIPTION:

A FOR program structure is used to execute a block of code a number of times.

On entering this loop the variable is initialised 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**. If the value of the variable is less than or equal to the end parameter then the block of commands is repeatedly executed. Once the variable is greater than the end value the program drops out of the FOR.. **NEXT LOOP**.

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

PARAMETERS:

commands:	Trio **BASIC** statements that you wish to execute
variable:	A valid Trio **BASIC** variable. Either a global **VR** variable, or a local variable may be used.

start:	The initial value for the variable
end:	The final value for the variable
increment:	The value that the variable is incremented by , this may be positive or negative

The **STEP** increment is optional, if this is omitted then the **FOR NEXT** will increment by 1

The variable can be adjusted or used within the structure.

EXAMPLES:

EXAMPLE 1:

Turn ON outputs 10 to 18, using the variable to change the output.

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

EXAMPLE 2:

Index an axis from 5 to -5 using a negative **STEP**.

```
FOR dist=5 TO -5 STEP -0.25
  MOVEABS(dist)
  WAIT IDLE
  GOSUB pick_up
NEXT dist
```

EXAMPLE 3:

Using a FOR structure to move through a set of x,y positions. If there is a **MOTION_ERROR** then the variables are set to a large values so the loop no longer repeats

```
FOR x=1 TO 8
  FOR y=1 TO 6
    MOVEABS(x*100,y*100)
    WAIT IDLE
    GOSUB operation
    IF MOTIONERROR THEN
      x=10
      y = 10
    ENDIF
  NEXT y
NEXT x
```

FORCE_SPEED

TYPE:
Axis Parameter

DESCRIPTION:
This parameter sets the main speed for a motion command that supports the advanced speed control (commands ending in SP). The **VP_SPEED** will accelerate or decelerate so that the profile is completed at **FORCE_SPEED**

The lowest value of **SPEED**, **ENDMOVE_SPEED**, **FORCE_SPEED** or **STARTMOVE_SPEED** will take priority.

FORCE_SPEED is loaded into the buffer at the same time as the move so you can set different speeds for subsequent moves.

VALUE:
The speed at which the SP motion command will execute, in user **UNITS**. (default 0)

EXAMPLES:

EXAMPLE 1:
In this example the controller will ramp the speed down to a speed of 10 at the end of the **MOVE**. Then for the duration of the **MOVESP**(20) the speed will be 10, after which it will ramp back to a speed of 15.

```
SPEED = 15
MOVE(100)
FORCE_SPEED = 10
MOVESP(20)
MOVE(100)
```

EXAMPLE 2:
Use **FORCE_SPEED** to slow the profile speed down during a corner move

```
FORCE_SPEED=100
MOVESP(100,0)
FORCE_SPEED=50
MOVECIRCSP(100,100,100,0,1)
FORCE_SPEED=100
MOVESP(0,100)
```

SEE ALSO:
ENDMOVE_SPEED, STARTMOVE_SPEED

FORWARD

TYPE:
Axis Command

SYNTAX:
FORWARD

ALTERNATE FORMAT:
FO

DESCRIPTION:
Sets continuous forward movement. The axis accelerates at the programmed **ACCEL** rate and continues moving at the **SPEED** value until either a **CANCEL** or **RAPIDSTOP** command are encountered. It then decelerates to a stop at the programmed **DECEL** rate.

If the axis reaches either the forward limit switch or forward soft limit, the **FORWARD** will be cancelled and the axis will decelerate to a stop.

EXAMPLES:

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

FPGA_PROGRAM

TYPE:
System Function

SYNTAX:
value = FPGA_PROGRAM(program)

DESCRIPTION:
This function allows you to select between the different **FPGA** programs that are available on controllers that support **FPGA** re-programming.

Rather than using this command we recommend using the tool in *Motion* Perfect to select the **FPGA** variant.

PARAMETERS:

variant:	-1	Displays **FPGA** images stored in local controller flash memory
	>=0	The program number to load, see table below or check **FPGA_PROGRAM**(-1) to see available options.
value:	**TRUE**	**FPGA** programmed successfully

MC403:

FPGA_PROGRAM	FEATURES	NOTES
0	Servo, Stepper, **HW_PSWITCH**, SSI	Default program
1	Servo, Stepper, **HW_PSWITCH**, Tamagawa	
2	Servo, Stepper, **HW_PSWITCH**, EnDAT	**HW_PSWITCH** only available on first 2 axes

MC405:

FPGA_PROGRAM	FEATURES	NOTES
0	Servo, Stepper, **HW_PSWITCH**, SSI, Tamagawa	Default program
1	Servo, Stepper, **HW_PSWITCH**, SSI, EnDAT	
2	Reserved	

EXAMPLE:

Check the available **FPGA** programs then load program 1 so that an EnDAT encoder can be used. Do not forget to power cycle.

```
>>FPGA_PROGRAM(-1)
0 : (00C) Servo,Stepper,PSwitch,SSI,Tamagawa
1 : (00C) Servo,Stepper,PSwitch,SSI,EnDAT
>>FPGA_PROGRAM(1)
>>
```

SEE ALSO:

FPGA_VERSION

FPGA_VERSION

TYPE:

Slot Parameter

DESCRIPTION:

Using the **SLOT** modifier on the MC464 enables checking of the **FPGA** version number in the main controller and any of the expansion modules.

On controllers that support **FPGA** re-programming, the version number is split to display the main version number and program loaded.

VALUE:

On the MC464 it displays the **FPGA** version of the specified **SLOT**

On controllers that support **FPGA** variants the **FPGA** returns the following:

Bit	Description	Function
0 - 7	**FPGA** version number	Unique version number for this **FPGA** program
8 - 14	**FPGA** program	The currently installed **FPGA_PROGRAM**

Bits 8-14 return a number that is one higher than the one you use in **FPGA_PROGRAM**

EXAMPLE:

Check the currently installed **FPGA** program and its version number on the command line. The result shows that **FPGA** program 1 is installed and the version is 0C.

```
>>PRINT HEX(FPGA_VERSION)
10C
>>
```

SEE ALSO:

FPGA_PROGRAM, SLOT

FPU_EXCEPTIONS

TYPE:

Reserved Keyword

FRAC

TYPE:

Mathematical Function

SYNTAX:

```
value = FRAC(expression)
```

DESCRIPTION:

Returns the fractional part of the expression.

PARAMETERS:

value:	The fractional part of the expression
expression:	Any valid TrioBASIC expression

EXAMPLE:

Print the fractional part of 1.234 on the command line

```
>>PRINT FRAC(1.234)
0.2340
>>
```

FRAME

TYPE:

Axis Parameter

DESCRIPTION:

A **FRAME** is a transformation which enables the user to program in one coordinate system when the machine or robot does not have a direct or one-to-one mechanical connection to this coordinate system.

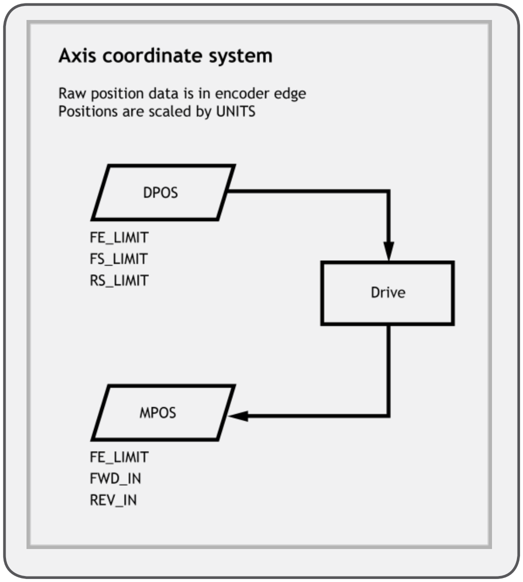
The **FRAME** command selects which transformation to use on axes in a **FRAME_GROUP**. Applying a **FRAME** to an axis in a **FRAME_GROUP** will apply that frame to all the axes in the group. To make this compatible with older firmware, if no FRAME_GROUPs have been configured then a default group is generated using the lowest axes, regardless of what axis the **FRAME** parameter was issued on.

Most transformations require configuration data to specify the lengths of mechanical links or operating modes. This is stored in the table with offsets detailed below in the parameters list. These table positions are offset by the ‘table_offset’ parameter in **FRAME_GROUP**. For a default **FRAME_GROUP** table_offset is 0.

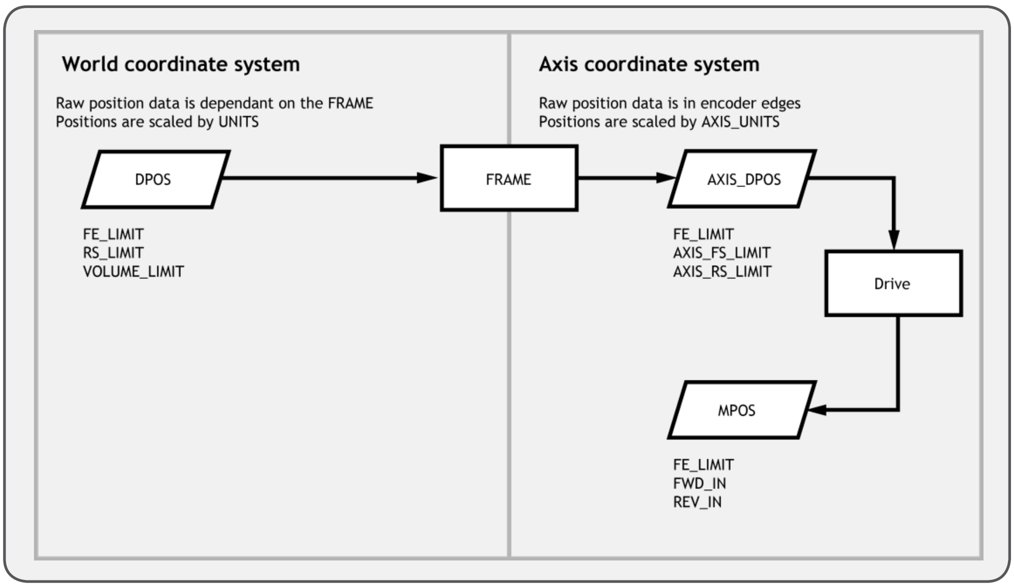
Do not change the **FRAME TABLE** parameters with the **FRAME** enabled. This can result in unpredictable movement which could cause damage or harm.

The kinematic runtime feature enable code is required to run **FRAME** 14 and higher

SYSTEM WITH FRAME=0



SYSTEM WITH FRAME<>0



AXIS SCALING

When a **FRAME** is enabled **UNITS** applies the scaling to the world coordinate system and **AXIS_UNITS** applies scaling to the axis coordinate system.

When **FRAME** is enabled **MPOS** is scaled by **AXIS_UNITS**, when frame is disabled **MPOS** is scaled by **UNITS**.

POSITION AND FOLLOWING ERRORS

When a **FRAME** is active **MPOS** is the motor position and **DPOS** is in the world coordinate system. **AXIS_DPOS** can be read to find the demand position in the motor coordinate system.

The following error is calculated between **MPOS** and **AXIS_DPOS** and so is the following error of the motor.

When using multiple frames or if you wish to group your axis you can use **DISABLE_GROUP** so that a **MOTION_ERROR** on one axis does not affect all.

HARDWARE AND SOFTWARE LIMITS

As **FS_LIMIT** and **RS_LIMIT** use **DPOS** they are both active in the world coordinate system. **VOLUME_LIMIT** also uses **DPOS** so is also in the world coordinate system. **FWD_IN** and **REV_IN**, **AXIS_FS_LIMIT** and **AXIS_RS_LIMIT** use **AXIS_DPOS** as so act on the forward and reverse limit of the motor.

When moving off **FWD_IN** and **AXIS_FS_LIMIT** the motor must move in a reverse direction. Due to the **FRAME** transformation this may not be a reverse movement in the world coordinate system. When moving off a **REV_IN** and **AXIS_RS_LIMIT** the motor must move in a forward direction. Due to the **FRAME** transformation this may not be a forward movement in the world coordinate system.

OFFSETTING POSITIONS

When a **FRAME** is enabled **OFFPOS** and **DEFPOS** must not be used as they cause a jump in both **DPOS** and **MPOS**. As the transformation separates **DPOS** and **MPOS** using these commands will cause an undesirable jump in motor position.

REP_DIST also causes a jump in **DPOS** and **MPOS** so when using a **FRAME** the position must never reach **REP_DIST**. **REP_OPTION** must be set to 0 and **REP_DIST** must be at least twice the size of the biggest possible move on the system.

When **DATUM** is complete it also causes a jump in **DPOS** and **MPOS**, so **DATUM** must never be used when **FRAME** <> 0

You can use **USER_FRAME** to define a different origin to program from.

POWER ON SEQUENCE AND HOMING

Some **FRAME** transformations require the machine to be homed and/ or moved to a position before the **FRAME** is enabled. This can be done using the **DATUM** function. If you home position is not the zero position of the **FRAME** then you can use **DEFPOS**/ **OFFPOS** to set the correct offset before enabling the **FRAME**.

When a **FRAME** is enabled **DPOS** is adjusted to the world coordinates which are calculated from the current **AXIS_DPOS**.

You should not perform a **DATUM** homing routine when the **FRAME** is enabled as this will change the **DPOS** which may result in undesirable motion. If you need to perform homing when the **FRAME** is enabled you can move to a registration position and then use **USER_FRAME** to apply the offset.

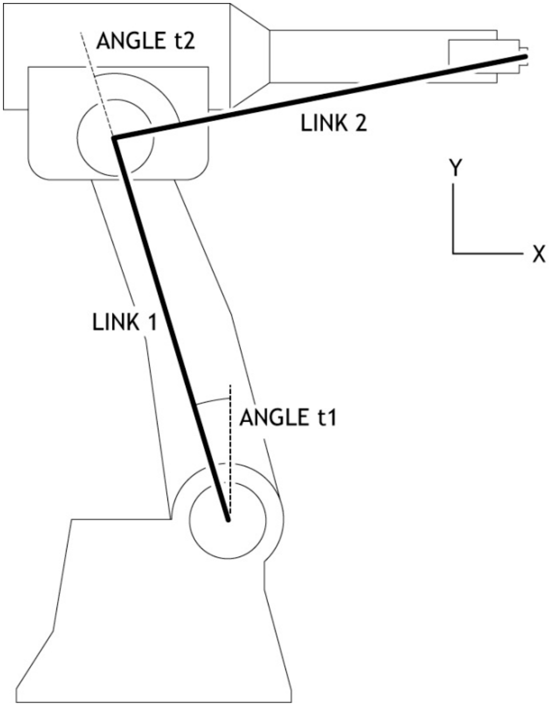
VALUE:

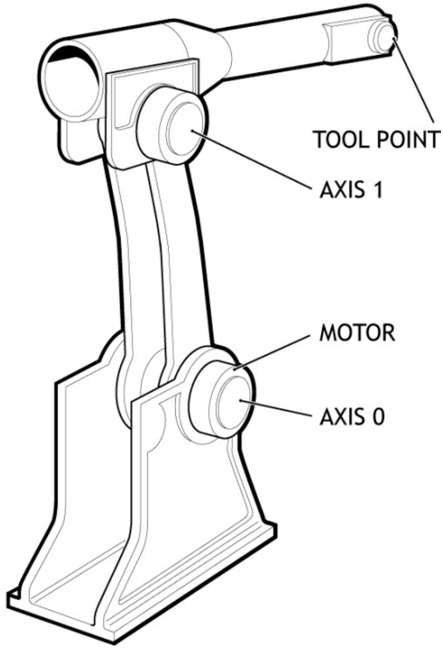
0	No transform
1	2 axis **SCARA** robot
2	XY single belt
5	2 axes rotation
6	Polar to Cartesian transformation
10	Cartesian to polar transformation
13	Dual arm robot transformation
14	3 arm delta robot.
15	4 axis **SCARA**
16	3 Axis Robot with 2 Axis Wrist
17	Wire guided camera
18	6 axis articulated arm
114	3 arm delta robot.
115	3 to 5 axis **SCARA**
116	3 Axis Robot with 2 Axis Wrist
119	3 to 5 axis cylindrical robot with 2 Axis Wrist

FRAME=1, 2 AXIS SCARA

DESCRIPTION:

Frame=1 allows the user to program in X, Y, Cartesian coordinates for a 2 axis **SCARA** arm like the example below. The frame allows for 2 configurations of a **SCARA** depending if the second axis motor is in the joint or at the base. The difference is that in angle t2 is referenced from link 1, or t2 is referenced from the base. A linkage or belt is typically used to keep t2 referenced to the base.





Second motor is carried on the end of Link 1, t2 is relative to link 1

Second motor in base with link arm to move upper part, t2 is relative to the base

Once the frame is enabled **DPOS** is measured in Micrometres, **UNITS** can then be set to a convenient scale.

HOMING

Is it required that the 2 motors' absolute positions are homed relative to the "straight up" position before the **FRAME** is enabled. In other words, the zero angle on each axis is with the arms in line and vertical. Of course it is not necessary for the motors to actually go to this position as you can offset the position using **DEFPOS** or **OFFPOS**.

JOINT CONFIGURATION

The joint configuration is determined by the position of the **SCARA** arm when you enable **FRAME** = 1

The joint is defined as Right Handed if:

(t2<t1) -both motors in base

(t2<0) -motors in the joint

Otherwise the robot is Left handed

PARAMETERS:

Table data	0	Length of arm 1 in micrometres
	1	Length of arm 2 in micrometres
	2	Edges per radian for joint 1
	3	Edges per radian for joint 2
	4	Internal value. Set to 0 to force frame re-calculation
	5	Axis configuration:
		0 – Both motors fixed in base
		1 – Motors at the joint
	6	Joint configuration (read only):
		0 – Left handed **SCARA**
		1 – Right handed **SCARA**
	7	used internally
	8	used internally

EXAMPLES:

EXAMPLE 1:

Set up the **SCARA** arm which is configured with the motors in the joints. Both motors return 16000 counts per revolution. The robot can be homed to switches which are at -80 degrees and +150degrees for the two joints. After setting **FRAME**=1 the tip of the second arm will be set with X, Y as (0,42426). This effectively makes the (0,0) XY position to be the bottom joint of the lower arm.

All the normal move types can then be run within the **FRAME**=1 setting until it is reset by setting **FRAME**=0. As the **FRAME** 1 makes the resolution of axes 0 and 1 micrometres, the **UNITS** can be set so you can program in mm.

```
FRAME=0

‘Enter Configuration Parameters:
TABLE(0, 300000) ‘    Length of arm 1 in mm * 1000
TABLE(1, 445000) ‘    Length of arm 2 in mm * 1000
TABLE(2, 16000/(2*PI)) ‘ edges per radian for joint 1
TABLE(3, 16000/(2*PI)) ‘ edges per radian for joint 2
TABLE(4, 0) ‘ Internal value. Set to 0 to force frame re-calculation
TABLE(5, 1) ‘ set to 1 for second joint fixed to arm 1

‘Home the robot to its mechanical limit switches
DATUM(3) AXIS(0) ‘ find home switch for lower part of arm
WAIT IDLE
```

```
DATUM(3) AXIS(1) ‘ find upper arm home position
WAIT IDLE

‘The mechanical layout may make it impossible to home at (0,0)
‘Define the home position values as their true angle (in edges)
DEFPOS(-3555,6667) ‘ say home position is -80 deg and +150 deg
WAIT UNTIL OFFPOS=0

‘Move both arms to start position PI/4 radians (45 degrees)
MOVEABS(-TABLE(2)*0.7854,TABLE(3)*0.7854*2)
WAIT IDLE

FRAME=1

UNITS AXIS(0)=1000
UNITS AXIS(1)=1000
```

EXAMPLE 2:

Set up the table for **SCARA** arm which is configured with both motors in the base. Once the table is configured the rest of the initialisation is the same as the above example.

```
‘ Enter Configuration Parameters:
TABLE(0,400000) ‘         Link 1 in mm * 1000
TABLE(1,250000) ‘         Link 2 in mm * 1000
TABLE(2, 4096*5/(2*PI)) ‘ t1 in edges per radian
TABLE(3, 4096*3/(2*PI)) ‘ t2 in edges per radian
TABLE(4,0) ‘ Internal value. Set to 0 to force frame re-calculation
TABLE(5,0) ‘ set to 0 for second joint fixed to base
```

FRAME=2, XY SINGLE BELT

DESCRIPTION:

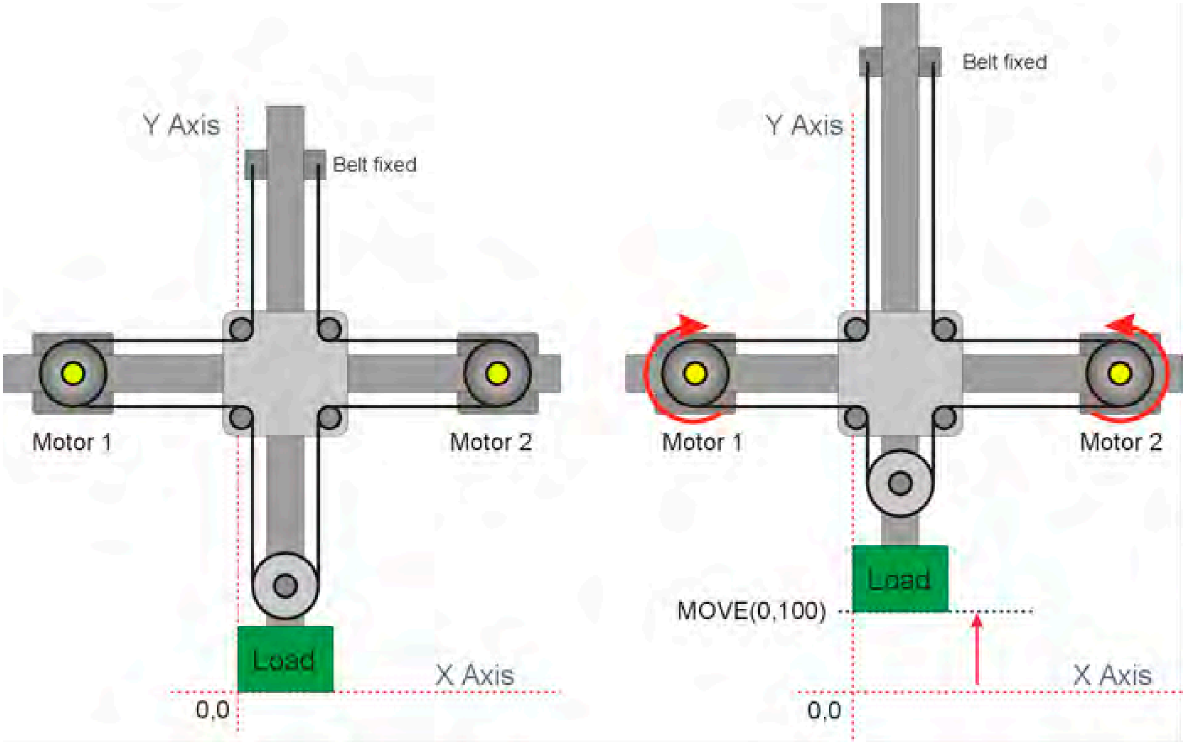
Switching to **FRAME**=2 will allow X-Y motion using a single-belt configuration. In this mode, an interpolated move of **MOVE**(0,100) produces motion on both motor 1 and motor 2 to raise the load vertically, based on the transformed position. Note that the two motors are located on the X-axis. The mass of the Y-axis can be minimized in this configuration. The equations for the transformed position of the X and Y axes are as follows:

Xtransformed = (**MPOS AXIS**(0)+ **MPOS AXIS**(1))*0.5

Ytransformed = (**MPOS AXIS**(0)- **MPOS AXIS**(1))*0.5

The transformed X-Y coordinates are derived from the measured encoder position (**MPOS**) of **AXIS**(0) and **AXIS**(1). This conversion is automatically accomplished by the *Motion Coordinator* when **FRAME**=2.

Once the frame is enabled **DPOS** is measured in encoder counts, **UNITS** can be set to enable a more convenient scale.



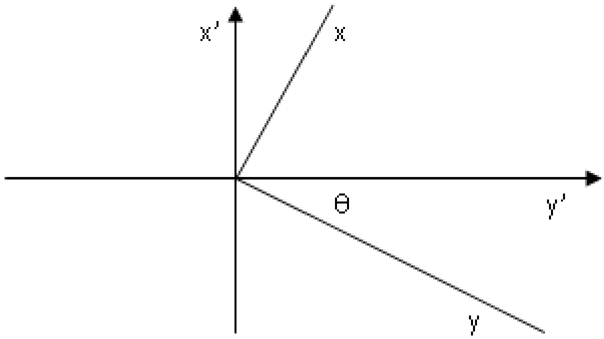
EXAMPLE:

```
ATYPE=0 'disable built in axes for MC464

FRAME=0

'Define a start position
DEFPOS(150,50)
FRAME=2
```

FRAME=5, 2 AXES ROTATION

DESCRIPTION:

This frame is designed to allow two orthogonal axes to be “turned” through an angle so that command inputs to x, y (along the required plane) are transformed to the fixed axes x’ and y’.



The transform is done by way of a 2 x 2 matrix, the coefficients of which can be easily derived from the required rotation angle of the operating plane.

CALCULATING THE MATRIX COEFFICIENTS:

For the frame to work, 2 sets of matrix coefficients must be entered, one for the forward transform and the second for the inverse. The transform calculates x and y according to the following:

$$(x', y') = (x, y) * \begin{pmatrix} \mathrm{TABLE}(0), & \mathrm{TABLE}(1) \\ \mathrm{TABLE}(2), & \mathrm{TABLE}(3) \end{pmatrix}$$

The inverse transform is calculated thus:

$$(x, y) = (x', y') * \begin{pmatrix} \mathrm{TABLE}(4), & \mathrm{TABLE}(5) \\ \mathrm{TABLE}(6), & \mathrm{TABLE}(7) \end{pmatrix}$$

HOMING:

The axes should be datumed in **FRAME**=0. Once this is done, then the frame can be set to 5 and move commands directed at either axis or at both axes together in the usual way. However the actual movement of x' and y' (the real axes) will be according to the transform.

If the axes need to be re-positioned according to the real axes, the frame can be turned off simply by setting **FRAME**=0. When this is done, the **DPOS** values will change to be the same as the **MPOS** positions, i.e. they become the positions in the x' / y' plane. The axes can then be moved to a new starting position and the frame set back to 5, perhaps with a new angle set.

PARAMETERS:

Table data	0	COS(theta)
	1	-SIN(theta)
	2	SIN(theta)
	3	COS(theta)
	4	**TABLE**(3) / det
	5	-**TABLE**(1) / det
	6	-**TABLE**(2) / det
	7	Table(0) / det

theta, the angle of rotation is in radians.

det = (**TABLE**(0) * **TABLE**(3)) - (**TABLE**(2) * **TABLE**(1))

EXAMPLE:

Configure a rotation of 45 degrees and run a move on the new X Y axes.

```
x_axis = 0
y_axis = 1

theta_degrees = 45 ‘Rotation angle in degrees
theta = theta_degrees * (2*PI/360) ‘Convert to radians
GOSUB calc_matrix
FRAME = 5

BASE(x_axis)
MOVE(xdist, ydist)
WAIT IDLE

STOP

‘=================================================
‘ Calculate the matrix parameters for FRAME 5
‘ Transform (x, y) * (TABLE(0), TABLE(1) )
‘                    (TABLE(2), TABLE(3) )
‘
‘ Inverse Transform:
‘          (x’, y’) * (TABLE(4), TABLE(5) )
‘                    (TABLE(6), TABLE(7) )
```

```
'=================================================
calc_matrix:
'Forward transform
TABLE(0, COS(theta))
TABLE(1, -SIN(theta))
TABLE(2, SIN(theta))
TABLE(3, COS(theta))

'Inverse transform
det = (TABLE(0) * TABLE(3)) - (TABLE(2) * TABLE(1))
TABLE(4, TABLE(3) / det)
TABLE(5, -TABLE(1) / det)
TABLE(6, -TABLE(2) / det)
TABLE(7, TABLE(0) / det)
RETURN
```

FRAME=6, POLAR TO CARTESIAN TRANSFORMATION

DESCRIPTION:

This transformation allows the user to program in polar (radius, angle) coordinates and the actual axis to move in a Cartesian (X, Y) coordinate system.

The first axis in the frame group is the Radius, the second is the angle. .

Once the frame is enabled the raw position data (**UNITS**=1) is measured in encoder counts for the radius axis and radians*scale for the angle, **UNITS** can then be set to a convenient scale. The origin for the robot is the zero position for the Cartesian system. The zero angle position is along Axis 0.

PARAMETERS:

Table data	0	Scale (counts per radian) for the rotary axis

EXAMPLES:

EXAMPLE 1:

A gantry robot has 2 axis configured in an X, Y configuration. For ease of programming the user would like to program in Polar coordinates. Both axes return 4000 counts per revolution. The **AXIS_UNITS** are set so that the axis coordinate system is in mm, the **UNITS** are set so that the World coordinate system is in mm and degrees.

```
scale = 1000000
UNITS AXIS(0) = 4000 'To program in mm
AXIS_UNITS AXIS(0) = 4000
UNITS AXIS(1) = scale*2*PI/360 'to program in degrees
AXIS_UNITS AXIS(1) = 4000
TABLE(0, scale) 'Set resolution for the angle axis
FRAME = 6
```

EXAMPLE 2:

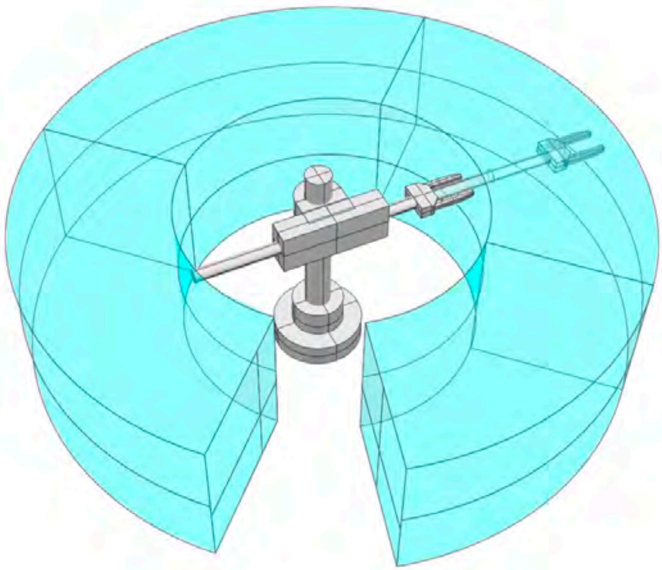
Using the robot configured in example 1 move the tool to 150mm along the X axis, then move the tool in a circle around the Polar coordinate system origin.

```
MOVEABS(150,0)
MOVE(0,360)
```

FRAME=10, CARTESIAN TO POLAR TRANSFORMATION

DESCRIPTION:

This **FRAME** transformation allows the user to program in Cartesian (X,Y) coordinates on a system that moves in a Polar (radius, angle) coordinate system. This is typically used on cylindrical robots where you need to program the arm extension (radius) and angle. The vertical Z axis can be simply added to make a 3 degree of freedom system.



Once the frame is enabled the raw position data (**UNITS**=1) is scaled the same for the X and Y axes, the resolution is set from the radius axis. **UNITS** can then be set to a convenient scale. The origin is the centre of the Polar system. .

The first axis in the group controls the radius axis and the second controls the rotary axis.

HOMING

Before enabling **FRAME**=10 the axes must be homed so that they are at a known position. When the **FRAME** is enabled the X and Y positions are calculated from the current Polar position.

Take care when executing moves that go close to the origin. Moves that travel through the origin will require infinite speed and acceleration. This is usually not possible to achieve and the axes will trip out due to excessive following error.

PARAMETERS:

Table data	0	Encoder edges/radian
	1	Number of revolutions, set by firmware
	2	Previous servo cycle's angle, set by firmware

EXAMPLE:

A cylindrical robot has 3 axis which extend the arm (radius), rotate the arm (angle) and move the up and down (Z). The radius and Z axes have 4000 counts per mm, this is used for the scale of the Cartesian axes in the **FRAME**. The rotate axis has 4000 counts per revolution, this should be divided by 2*PI to give the counts per revolution which is set in the table. The **UNITS** are set so that the Cartesian system can be programmed in mm, the **AXIS_UNITS** is set so that the axis are programmed in mm or degrees. Once the polar system has been homed the following code can be executed so that any further motion is programmed in Cartesian coordinates.

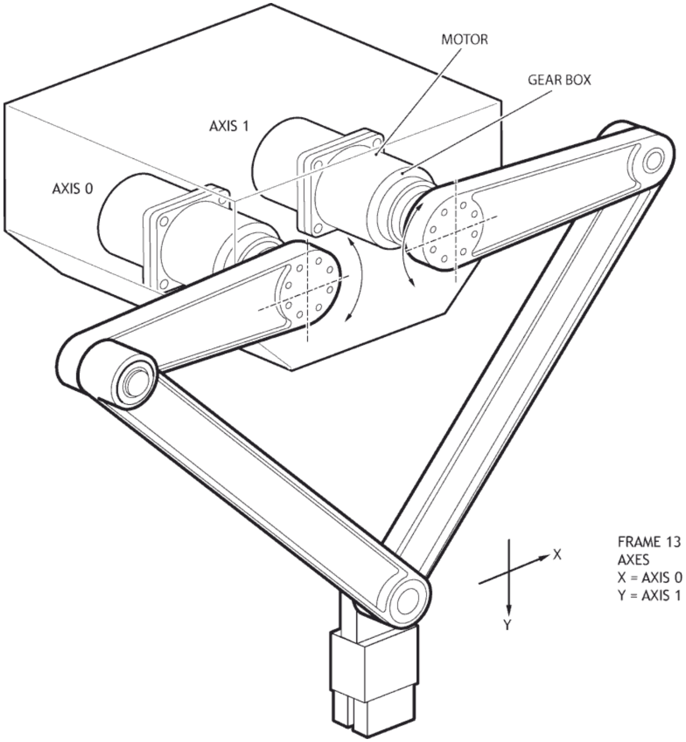
```
UNITS AXIS(0) = 4000 'To use in mm
AXIS_UNITS AXIS(0) = 4000 'To use in mm
edges_per_radian = 4000/(2*PI) 'Edges per radian for the rotary axis
UNITS AXIS(1) = 4000'To use in mm
AXIS_UNITS AXIS(1) = 4000 / 360 'To use in mm
TABLE(0,edges_per_radian)
UNITS AXIS(2) = 4000 'To use in mm
FRAME = 10
```

FRAME=13, DUAL ARM PARALLEL ROBOT

DESCRIPTION:

Frame 13 enables the transformation for a 2 arm parallel robot as shown. It is then possible to program in X Y Cartesian coordinates.

If the lower link is not directly connected as per the image but is separated, this is compensated for by decreasing the centre distance of the top link by the same amount.

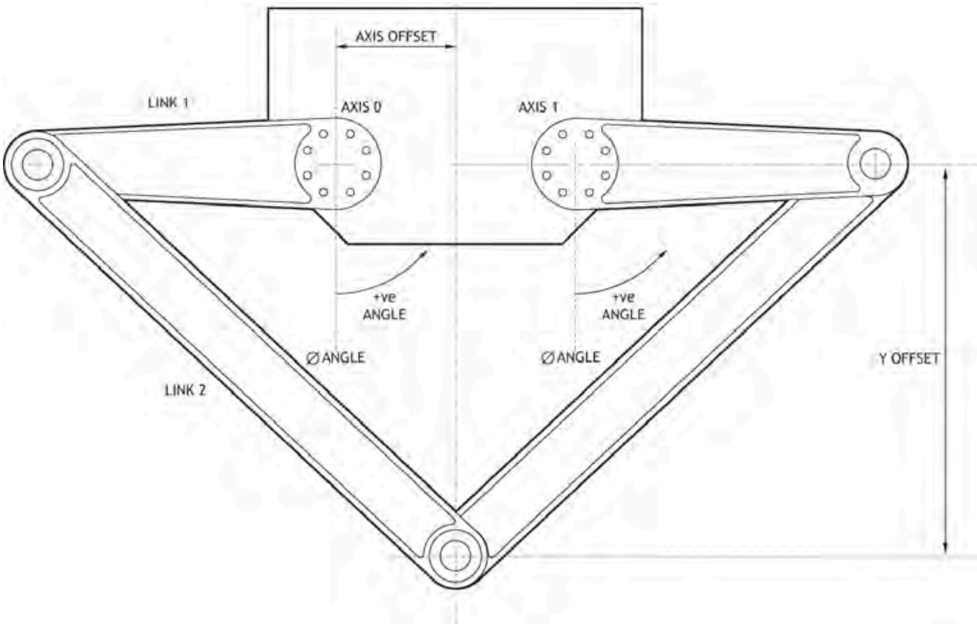


Once the frame is enabled the raw position data (**UNITS**=1) is measured in Micrometres, **UNITS** can then be set to a convenient scale.

HOMING

The 2 arm delta robot should be homed so that the two link 1's are vertical down. You do not need to enable the frame in this position, just ensure that it has been defined.

A vertical offset for the tool can be defined within the **FRAME** table data. This means that you can set the zero position vertically



PARAMETERS:

Table data	0	Link length 1 in microns
	1	Link length 2 in microns
	2	Encoder edges/radian axis 0
	3	Encoder edges/radian axis 1
	4	Horizontal offset axes from x datum
	5	Set Vertical datum with arms straight out
	6	calculated values
	7	calculated values
	8	calculated values
	12	first axis frame calculated value

EXAMPLE

The following is a typical startup program for **FRAME** 13.

```
FRAME=0
WA(10)
'------------------------------------------------
TABLE(0,220000)'Arm
TABLE(1,600000)'Forearm
TABLE(2,(2048*4*70)/2/PI)'pulse/radian
TABLE(3,(2048*4*70)/2/PI)'pulse/radian
TABLE(4,15000)'X-offset
TABLE(5,450000)'Y-offset = 450 mm below axis 0 centre
'------------------------------------------------

' set home position for arms at +/-90 degrees
DATUM(4) AXIS(0) 'find home switch for left arm
DATUM(3) AXIS(1) 'find home switch for right arm
WAIT IDLE AXIS(0)
WAIT IDLE AXIS(1)
home_0 = -TABLE(2)*PI/2
home_1 = TABLE(3)*PI/2
BASE(0,1)
DEFPOS(home_0,home_1)

WA(10)
FRAME=13
```

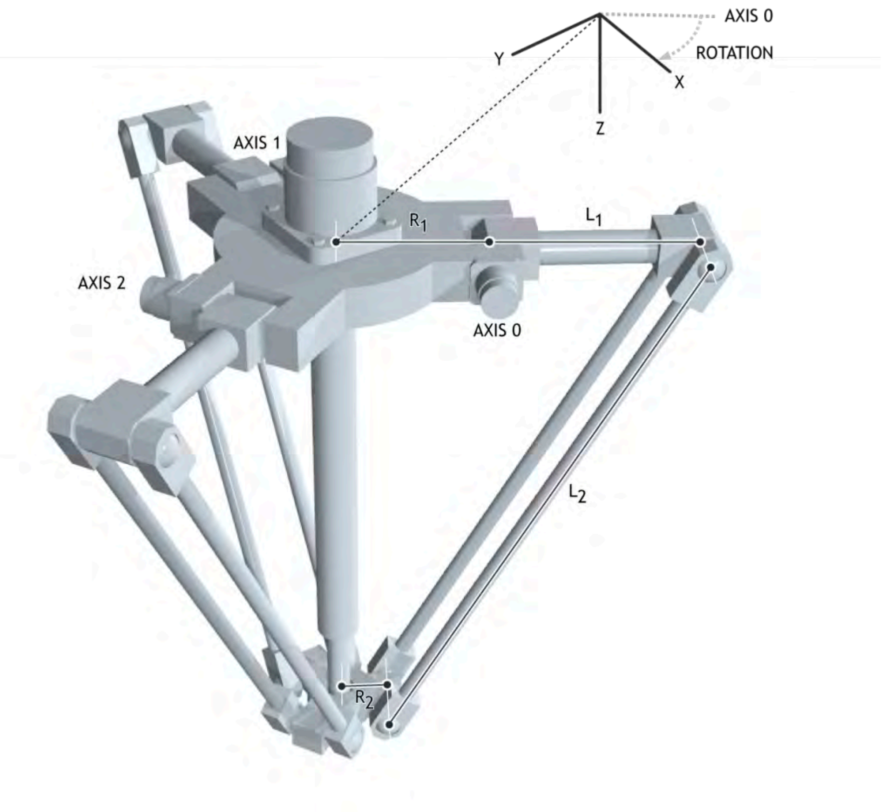
FRAME=14, DELTA ROBOT

DESCRIPTION:

FRAME=14 enables the transformation for a 3 arm 'delta' or 'parallel' robot. It transforms 3 axes from the mechanical configuration to Cartesian coordinates using the right hand rule.

For new projects **FRAME** 114 is recommended

FRAME=14 requires the kinematic runtime **FEC**



Once the frame is enabled the raw position data (**UNITS**=1) is measured in Micrometres, **UNITS** can then be set to a convenient scale. The origin for the robot is the centre of the top plate with the X direction following the first axis. This can be adjusted using the rotation parameter.

HOMING:

Before enabling **FRAME**=14 the position must be defined so that when the upper arms are horizontal the axis position is 0. You do not need to enable the frame in this position, just ensure that it has been defined.

PARAMETERS:

Table data	0	Top radius to joint in Micrometres (R1)
	1	Wrist radius to joint in Micrometres (R2)
	2	Upper arm length in Micrometres (L1)
	3	Lower arm length in Micrometres (L2)
	4	Edges per radian
	5	Angle of rotation in radians (Rotation)

EXAMPLE:

Start-up sequence for a 3 arm delta robot using the default **FRAME_GROUP**. Homing is completed using a sensor that detects when the upper arms are level.

```
‘ Define Link Lengths for 3 arm delta:
  TABLE(0,200000)’ Top radius to joint
  TABLE(1,50000)’  Wrist radius to joint
  TABLE(2,320000)’ Upper arm length
  TABLE(3,850000)’ Lower arm length

‘ Define encoder edges/radian
  ‘18bit encoder and 31:1 ratio gearbox
  resolution = 262144 * 31 / (2 * PI)
  TABLE(4,resolution)

‘ Define rotation of robot relative to global frame
  rotation = 30 ‘degrees
  TABLE(5, (rotation*2*PI )/360)

‘ Configure axis
  FOR axis_number=0 TO 2
    BASE(axis_number)
    ‘World coordinate system to operate in mm
    UNITS=1000
    SERVO=ON
  NEXT axis_number

  WDOG=ON
  BASE(0)

‘ Home and initialise frame
  ‘Arms MUST be horizontal in home position
  ‘ before frame is initialised.
```

```
FOR axis_number=0 TO 2
  DATUM(4)
  WAIT IDLE
NEXT axis_number

'Enable Frame
FRAME=14
```

FRAME=15, 4 AXIS SCARA

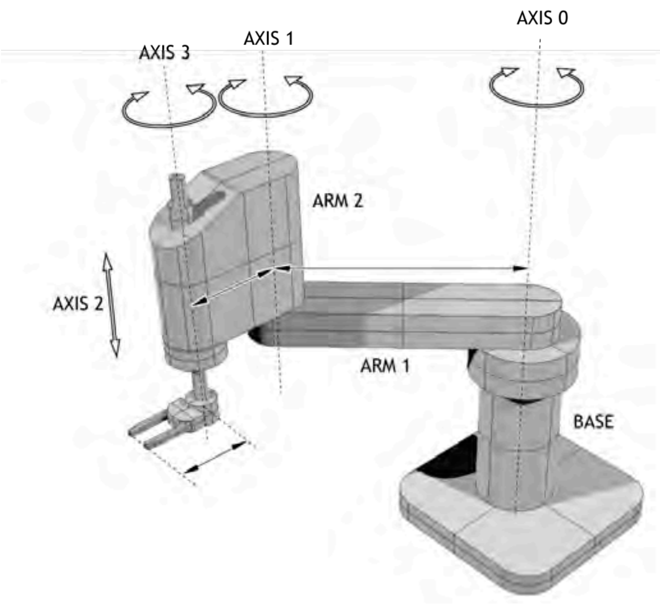
DESCRIPTION:

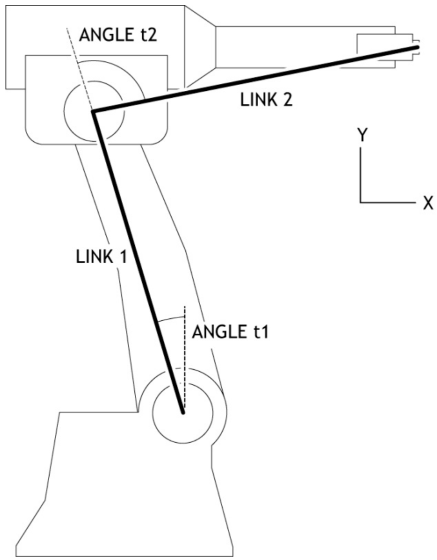
FRAME=15 enables the transformation for a 4 axis **SCARA** robot. This allows you to define the end position of the wrist in X.Y.Z and wrist angle (relative to the Y axis). The frame allows for 2 configurations of a **SCARA** depending if the second axis motor is in the joint or at the base. The difference is that the angle t2 is referenced from link 1, or the angle t2 is referenced from the base. A linkage or belt is typically used to keep t2 referenced to the base.

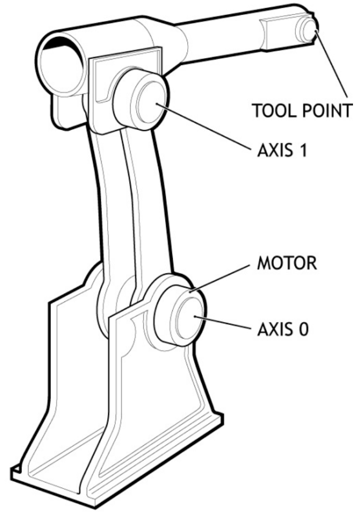
Some mechanical configurations have parasitic motion from the Z axis to the wrist angle. This can be included in the 'ratio' parameter. This is the change in encoder edges on the vertical for a change in wrist angle in encoder edges. Set this value to 0 if there is no parasitic motion.

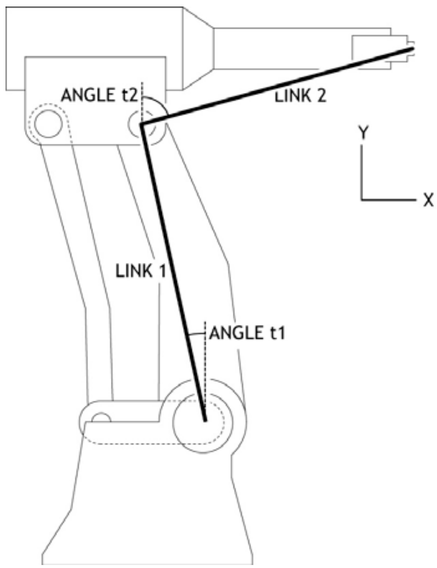
For new projects **FRAME** 115 is recommended

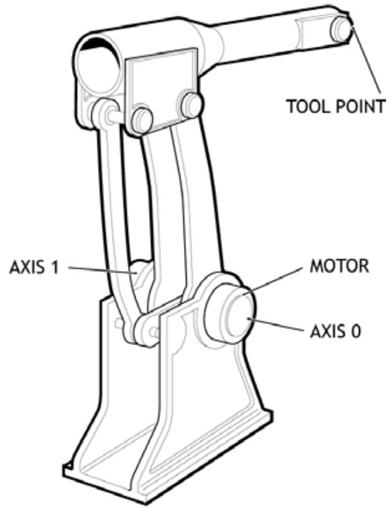
FRAME=15 requires the kinematic runtime **FEC**











Once the frame is enabled **DPOS** on the X,Y and Z axis are measured in Micrometres. The wrist axis is set to use Nanoradians. You can of course set **UNITS** for all axis to any suitable scale.

HOMING

Is it required that the X, Y and wrist absolute positions are homed relative to the “straight up” position before the **FRAME** is enabled. In other words, the zero angle on each axis is with the arms in line and vertical along the Y axis with Z=0. Of course it is not necessary for the motors to actually go to this position as you can offset the position using **DEFPOS** or **OFFPOS**.

JOINT CONFIGURATION

The joint configuration is determined by the position of the **SCARA** arm when you enable **FRAME** = 1

The joint is defined as Right Handed if:

(t2<t1) -both motors in base

(t2<0) -motors in the joint

Otherwise the robot is Left handed

PARAMETERS:

The table data values 0-8 are identical to **FRAME** 1, **SCARA**. This means you can easily switch between the 2 and 4 axis **SCARA**.

Table data	0	link1
	1	link2
	2	Encoder edges/radian axis 0
	3	Encoder edges/radian axis 1
	4	Internal value. Set to 0 to force frame re-calculation
	5	Mechanical configuration
		0 – Both motors fixed in base
		1 – Motors at the joint
	6	Joint configuration (read only)
		0 – Left handed **SCARA**
		1 – Right handed **SCARA**
	7	used internally
	8	used internally
	9	Encoder edges/radian axis 3
	10	link3
	11	Ratio of encoder edges moved on axis 2/ edge axis3
	12	Encoder edges/mm axis 2

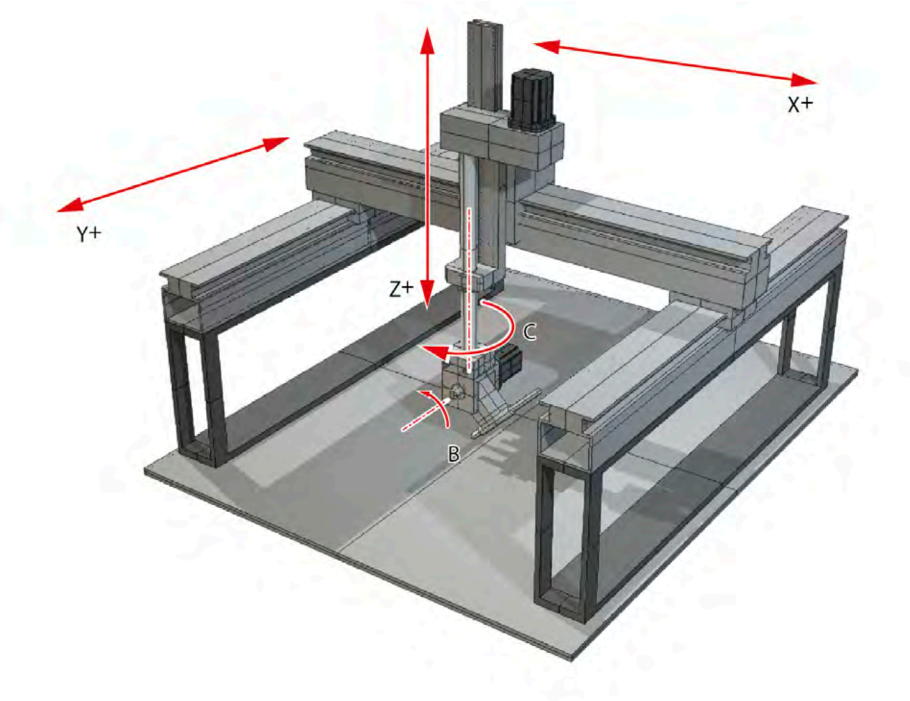
FRAME = 16, 3 AXIS ROBOT WITH 2 AXIS WRIST

DESCRIPTION:

The **FRAME** 16 transformation allows an XYZ Robot with 2 axis wrist to be easily programmed. The transformation function provides compensation in XYZ when the 2 wrist axes are rotated.

For new projects **FRAME** 116 is recommended

FRAME=16 requires the kinematic runtime **FEC**

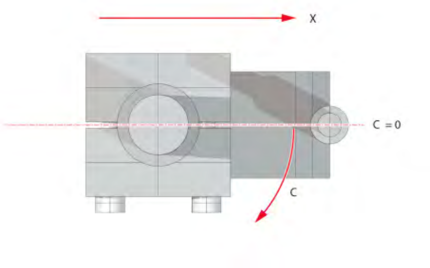


Once the frame is enabled **DPOS** on the X, Y and Z axis are measured in axis counts. The wrist axis is set to use Nanoradians. You can of course set **UNITS** for all axis to any suitable scale.

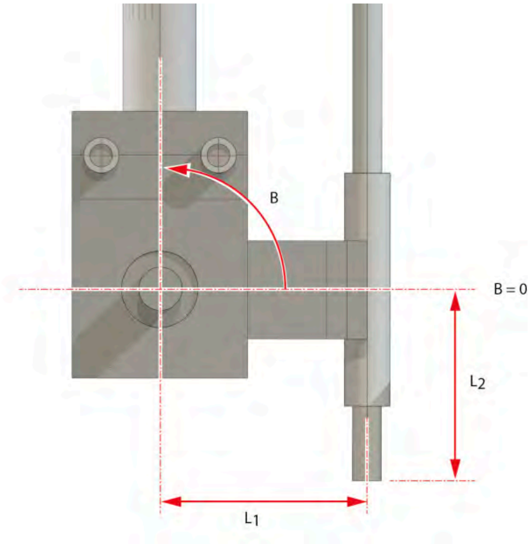
HOMING

Both wrist axes **MUST** be datumed to the correct zero position for the **FRAME** 16 transformation to operate. The zero position of the XYZ axes is not used by the transformation.

The zero position on the C axis (rotation about Z) is when the offset arm is in line with the X axis. The diagram below is drawn from above looking down on to the X-Y plane.



The zero position on the B axis(rotation about Y) is when the offset arm is the “straight down” position shown in the diagram.



The direction of motion on all 5 axes **MUST** match the diagram for the **FRAME** 16 transformation to operate.

If an axis direction of motion is inverted it can be reversed either:

Using the facility of the servo/stepper driver to invert the motion direction

On pulse direction axes using **STEP_RATIO** function inside the *Motion Coordinator*

On closed loop servo axes using **ENCODER_RATIO** / **DAC_SCALE** functions inside the *Motion Coordinator*

PARAMETERS:

Table data	0	Wrist joint to control point X offset (mm) (L1)
	1	Wrist joint to control point Z offset (mm) (L2)
	2	Wrist C axis encoder edges / radian
	3	Wrist B axis encoder edges / radian
	4	X axis encoder edges / mm
	5	Y axis encoder edges / mm
	6	Z axis encoder edges / mm

EXAMPLE:

Configure the table data for a XYZ Cartesian system with a spherical wrist.

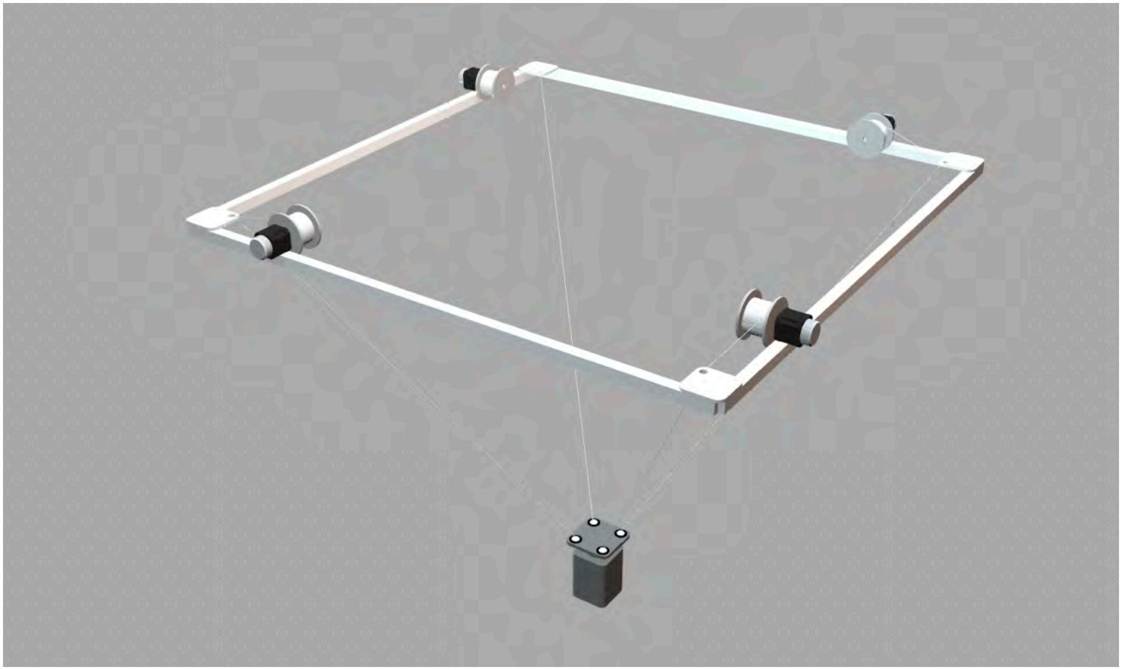
```
‘ Example:
‘ Wrist offsets: 60mm in X and 90 mm in Z
‘ XYZ pulses/mm 1600,1600,2560
‘ C and B axes pulses radian = 3200 * 16 / (2 * PI)

TABLE(100,60,90,3200 * 8 / PI, 3200 * 8 / PI,1600,1600,2560)

‘ Set FRAME_GROUP zero using axes 0,1,2,3,4

FRAME_GROUP(0,100,0,1,2,3,4)

FRAME=16

… program moves in XYZBC with tool angle compensation

FRAME=0

… program axes
```

FRAME=17, MULTI-WIRE CAMERA POSITIONING

DESCRIPTION:

The **FRAME** 17 transformation allows a wire mounted stadium camera to be easily programmed. The transformation function calculates the initial XYZ position of the camera using trilateration from 3 wire mounting points. During running the **FRAME** 17 calculations will calculate the wire lengths for up to 6 support wires with reels mounted in any XYZ positions.

FRAME=114 requires the kinematic runtime **FEC**



HOMING:

The length of wire related to each motor position must be known for the **FRAME** 17 transformation to operate. This requires that the wire winding drums are fitted with absolute encoders or that the system can start from a known position effectively datuming the axes.

PARAMETERS:

0	X axis position of payout position 1	User choice units
1	Y axis position of payout position 1	User choice units
2	Z axis position of payout position 1	User choice units
3	X axis position of payout position 2	User choice units
4	Y axis position of payout position 2	User choice units

5	Z axis position of payout position 2	User choice units
6	X axis position of payout position 3	User choice units
7	Y axis position of payout position 3	User choice units
8	Z axis position of payout position 3	User choice units
9	X axis position of payout position 4 (optional)	User choice units
10	Y axis position of payout position 4 (optional)	User choice units
11	Z axis position of payout position 4 (optional)	User choice units
12	X axis position of payout position 5 (optional)	User choice units
13	Y axis position of payout position 5 (optional)	User choice units
14	Z axis position of payout position 5 (optional)	User choice units
15	X axis position of payout position 6 (optional)	User choice units
16	Y axis position of payout position 6 (optional)	User choice units
17	Z axis position of payout position 6 (optional)	User choice units
18	Edges per user unit payout reel 1	Ratio (E.G. edges/mm)
19	Edges per user unit payout reel 2	Ratio (E.G. edges/mm)
20	Edges per user unit payout reel 3	Ratio (E.G. edges/mm)
21	Edges per user unit payout reel 4 (optional)	Ratio (E.G. edges/mm)
22	Edges per user unit payout reel 5 (optional)	Ratio (E.G. edges/mm)
23	Edges per user unit payout reel 6 (optional)	Ratio (E.G. edges/mm)
24	Option	0 or 1
25	Axes	3..6
26	Scale	Scale User units (see below)
27	Calculation Error	Output 0 (Error) 1 (Solution)

Payout positions: The positions (X,Y,Z) of between 3 and 6 payout positions must be specified to the calculation. These can be in the users choice of units. For example mm

Edges per user unit payout reel: These factors specify the number of encoder edges/user unit for each of the wire payout reels. The user units must be consistent with the payout positions so if the payout positions are specified in metres the edges number specified here must be edges/metre.

Option: The calculation for the camera position from 3 given lengths has 2 potential solutions. (The alternative solution normally requires negative gravity !) The Option parameter should be set to zero or 1 to give the correct solution.

Axes: A minimum of 3 wires are required. The **FRAME** 17 function will calculate the required wire lengths for between 3 and 6 payout drums. Note that the first 3 payouts only are used for calculating the starting position in **XYZ** from the 3 lengths. Where 4 or more wires are used the first 3 specified should be the most critical for the camera position.

Scale: When the **FRAME** 17 is running it calculates **INTEGER** positions in the **XYZ** space for the motion generator program inside the MC4XX. Since the user units (for example metres) are quite large distances a scale factor is required to ensure the integer positions are of fine resolution. The value should give fine resolution but the exact value is not critical. For example if the user units are metres the scale factor should be 100,000 or higher.

Calculation Error: In certain conditions (for example if the length of 1 or more wires is too short) the **FRAME** 17 calculation cannot be performed during the initial trilateration. In this case **TABLE** offset (27) is set to 0. 1 indicates a solution can be calculated.

EXAMPLE:

Test program using the **FRAME_TRANS** function to check correct operation:

```
ATYPE AXIS(0)=0
ATYPE AXIS(1)=0
ATYPE AXIS(2)=0
ATYPE AXIS(3)=0

FRAME_GROUP(1,100,0,1,2,3)

‘ These positions are in user units (mm for example)

TABLE(100,0,0,0)
TABLE(103,70,0,0)
TABLE(106,70,-40,0)

‘ 4th axis is not used to calculate starting position

TABLE(109,0,0,0)
TABLE(112,0,0,0)
TABLE(115,0,0,0)

‘ ratios:

ratio1=1000
ratio2=1000
ratio3=1000
ratio4=1000
```

```
TABLE(118,ratio1,ratio2,ratio3)
TABLE(121,ratio4,ratio5,ratio6)

' option:

scale = 1000

TABLE(124,1)'    solution option (1 or 0)
TABLE(125,4)'    axes 3..6
TABLE(126,1000)' scale factor

' These distances simulate axis positions so should be in edges:
TABLE(200,92.195*ratio1,60*ratio2,72.111*ratio3)

FRAME_TRANS(17,200,300,1,100)' convert wire lengths to XYZ

PRINT TABLE(300),TABLE(301),TABLE(302)

FRAME_TRANS(17,300,400,0,100)' convert XYZ to wire lengths

PRINT TABLE(400)/ratio1,TABLE(401)/ratio2,TABLE(402)/ratio3,TABLE(403)/
ratio4
```

FRAME=18, 6 AXIS ARTICULATED ARM

DESCRIPTION:

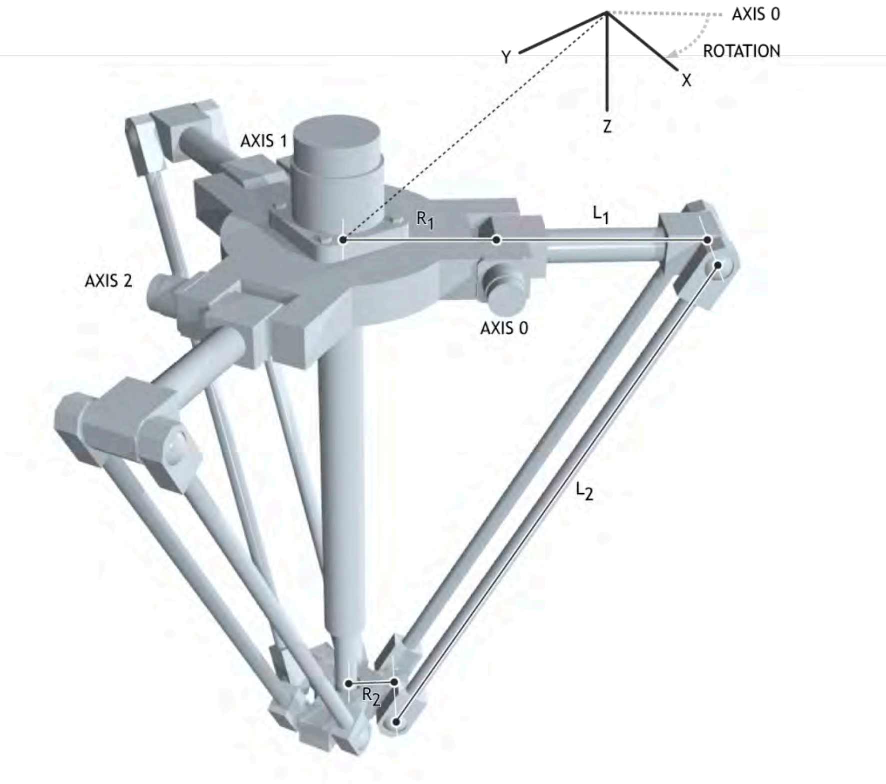
Please contact Trio for details.

FRAME=114, DELTA ROBOT

DESCRIPTION:

FRAME=114 enables the high accuracy transformation for a 3 arm ‘delta’ or ‘parallel’ robot. It transforms 3 axes from the mechanical configuration to Cartesian coordinates using the right hand rule.

FRAME=114 requires the kinematic runtime **FEC**



Once the **FRAME** is enabled set the **UNITS** to **FRAME_ANGLE_SCALE** so that the Cartesian movements use the same scale as that used in the table data. So if the **TABLE** data is programmed in mm then when **UNITS** is set to **FRAME_ANGLE_SCALE** then the robot can be programmed in mm.

The origin for the robot is the centre of the top plate with the X direction following the first axis. This can be adjusted using the rotation parameter.

HOMING:

Before enabling **FRAME**=114 the position must be defined so that when the upper arms are horizontal the axis position is 0. You do not need to enable the frame in this position or even move to it, just ensure that it has been defined.

Limits:

-70 to 90 degree

PARAMETERS:

Table data	0	Top radius to joint (R1)
	1	Wrist radius to joint (R2)
	2	Upper arm length (L1)
	3	Lower arm length (L2)
	4	Edges per radian
	5	Angle of rotation in radians (Rotation)
	6	Linkx (optional with 4 or 5 axis)
	7	Linky (optional with 4 or 5 axis)
	8	Linkz (optional with 4 or 5 axis)
	9	Encoder edges/radian (optional Z rotation)
	10	Encoder edges/radian (optional Y rotation)

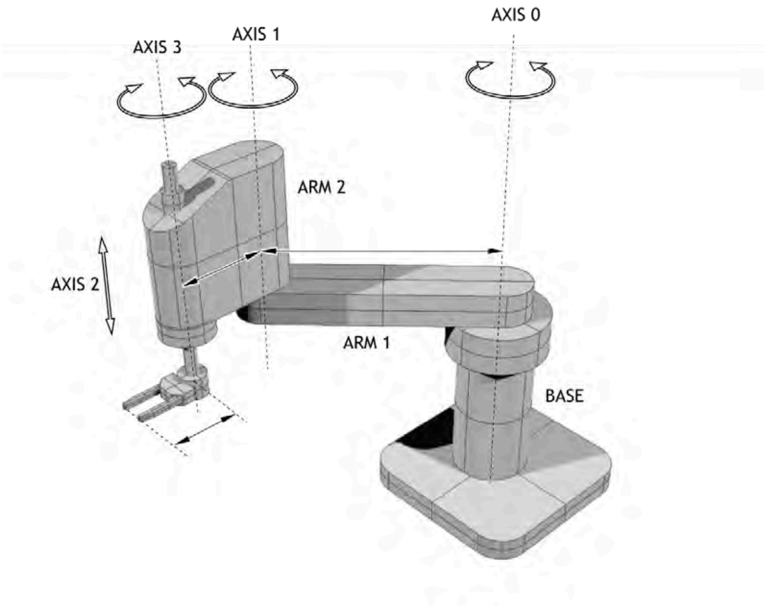
FRAME=115, 3 TO 5 AXIS SCARA

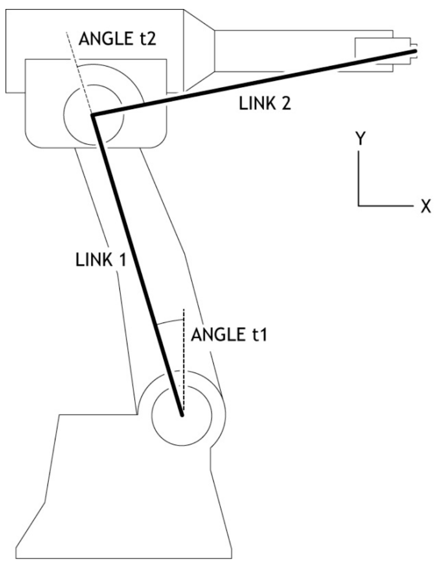
DESCRIPTION:

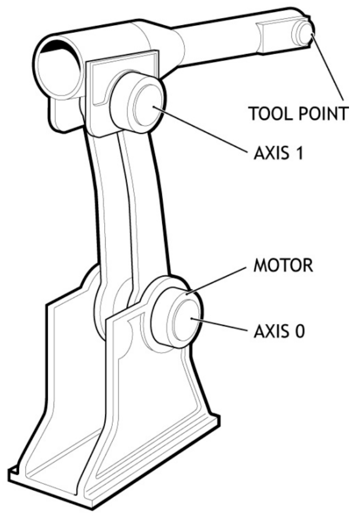
FRAME=115 enables the transformation for a 4 axis **SCARA** robot. This allows you to define the end position of the wrist in X,Y,Z and wrist angle (relative to the Y axis). The frame allows for 2 configurations of a **SCARA** depending if the second axis motor is in the joint or at the base. The difference is that the angle t2 is referenced from link 1, or the angle t2 is referenced from the base. A linkage or belt is typically used to keep t2 referenced to the base.

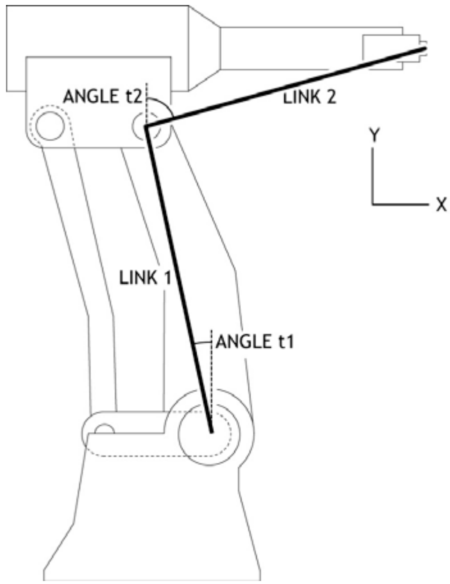
Some mechanical configurations have parasitic motion from the Z axis to the wrist angle. This can be included in the ‘ratio’ parameter. This is the change in encoder edges on the vertical for a change in wrist angle in encoder edges. Set this value to 0 if there is no parasitic motion.

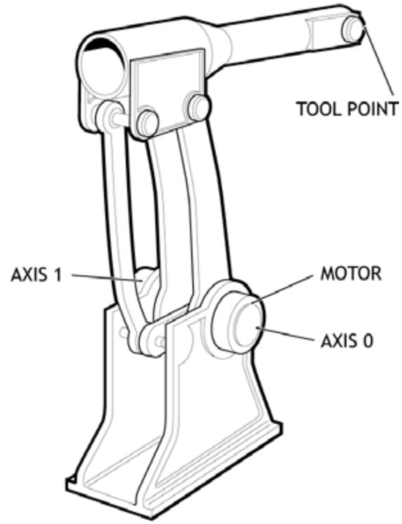
FRAME=115 requires the kinematic runtime **FEC**


AXIS 0
AXIS 3
AXIS 1
ARM 2
AXIS 2
ARM 1
BASE


ANGLE t2
LINK 2
Y
X
LINK 1
ANGLE t1


TOOL POINT
AXIS 1
MOTOR
AXIS 0





Once the **FRAME** is enabled set the **UNITS** to **FRAME_ANGLE_SCALE** so that the Cartesian movements use the same scale as that used in the table data. So if the **TABLE** data is programmed in mm then when **UNITS** is set to **FRAME_ANGLE_SCALE** then the robot can be programmed in mm.

Set the **UNITS** on the rotational (wrist) axes to **FRAME_ANGLE_SCALE** so that they are programmed in radians. You can of course set **UNITS** for all axis to any suitable scale.

HOMING

Is it required that the X, Y and wrist absolute positions are homed relative to the “straight up” position before the **FRAME** is enabled. In other words, the zero angle on each axis is with the arms in line and vertical along the Y axis with Z=0. Of course it is not necessary for the motors to actually go to this position as you can offset the position using **DEFPOS** or **OFFPOS**.

JOINT CONFIGURATION

The joint configuration is determined by the position of the **SCARA** arm when you enable **FRAME** = 1

The joint is defined as Right Handed if:

(t2<t1) –both motors in base

(t2<0) –motors in the joint

Otherwise the robot is Left handed

PARAMETERS:

The table data values 0-8 are identical to **FRAME** 1, **SCARA**. This means you can easily switch between the 2 and 5 axis **SCARA**.

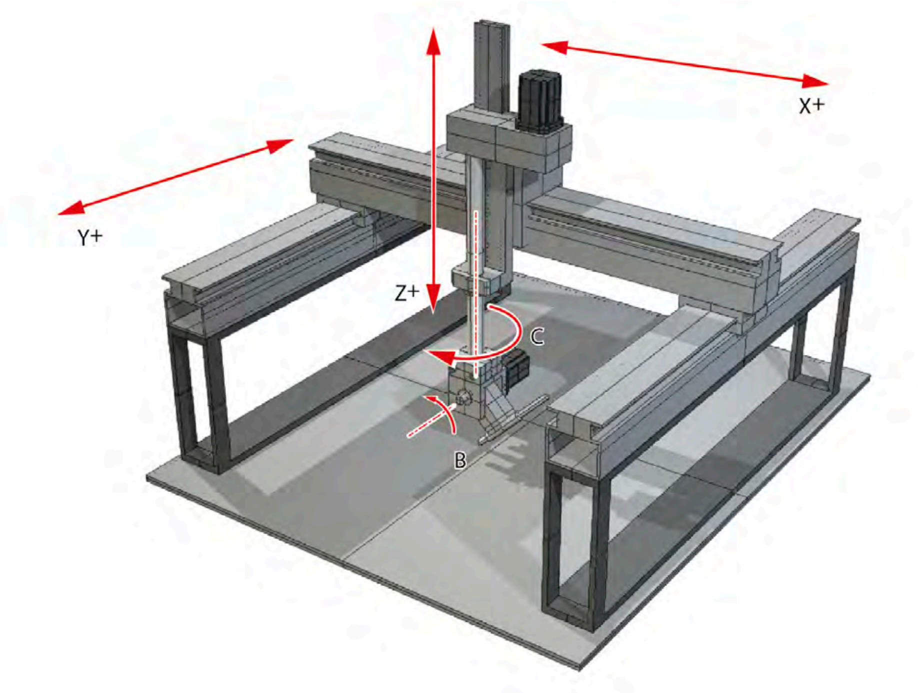
Table data	0	link1
	1	link2
	2	Encoder edges/radian axis 0
	3	Encoder edges/radian axis 1
	4	Mechanical configuration
		0 – Both motors fixed in base
		1 – Motors at the joint
	5	Joint configuration (read only)
		0 – Left handed **SCARA**
		1 – Right handed **SCARA**
	6	Encoder edges/mm axis 2
	7	Ratio of encoder edges moved on axis 2/ edge axis3
	8	Linkx (optional with 4 or 5 axis)
	9	Linky (optional with 4 or 5 axis)
	10	Linkz (optional with 4 or 5 axis)
	11	Encoder edges/radian (optional Z rotation)
	12	Encoder edges/radian (optional Y rotation)

FRAME = 116, 3 AXIS ROBOT WITH 2 AXIS WRIST

DESCRIPTION:

The **FRAME** 116 transformation allows an XYZ Robot with 2 axis wrist to be easily programmed. The transformation function provides compensation in XYZ when the 2 wrist axes are rotated.

FRAME=116 requires the kinematic runtime **FEC**

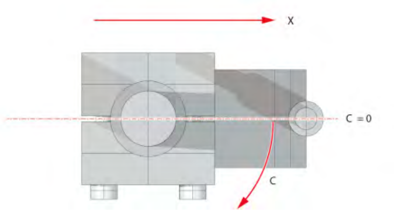


Once the **FRAME** is enabled set the **UNITS** to **FRAME_ANGLE_SCALE** so that the Cartesian movements use the same scale as that used in the table data. So if the **TABLE** data is programmed in mm then when **UNITS** is set to **FRAME_ANGLE_SCALE** then the robot can be programmed in mm.

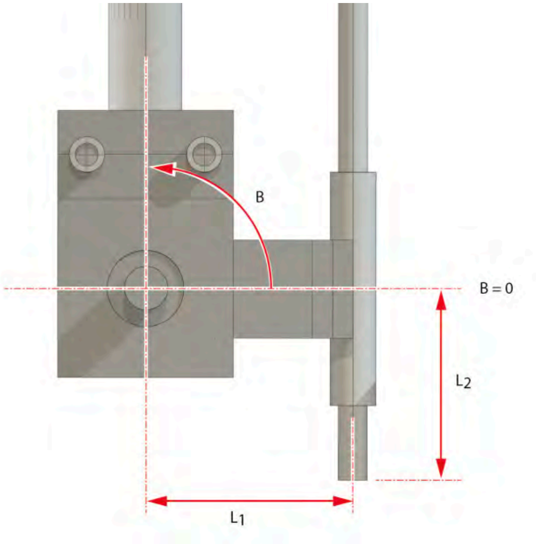
Set the **UNITS** on the rotational (wrist) axes to **FRAME_ANGLE_SCALE** so that they are programmed in radians. You can of course set **UNITS** for all axis to any suitable scale. Homing

Both wrist axes **MUST** be datumed to the correct zero position for the **FRAME** 116 transformation to operate. The zero position of the XYZ axes is not used by the transformation.

The zero position on the C axis (rotation about Z) is when the offset arm is in line with the X axis. The diagram below is drawn from above looking down on to the X-Y plane.



The zero position on the B axis(rotation about Y) is when the offset arm is the “straight down” position shown in the diagram.



The direction of motion on all 5 axes **MUST** match the diagram for the **FRAME** 116 transformation to operate.

- If an axis direction of motion is inverted it can be reversed either:
- Using the facility of the servo/stepper driver to invert the motion direction
- On pulse direction axes using **STEP_RATIO** function inside the *Motion Coordinator*
- On closed loop servo axes using **ENCODER_RATIO** / **D**AC**_SCALE** functions inside the *Motion Coordinator*

PARAMETERS:

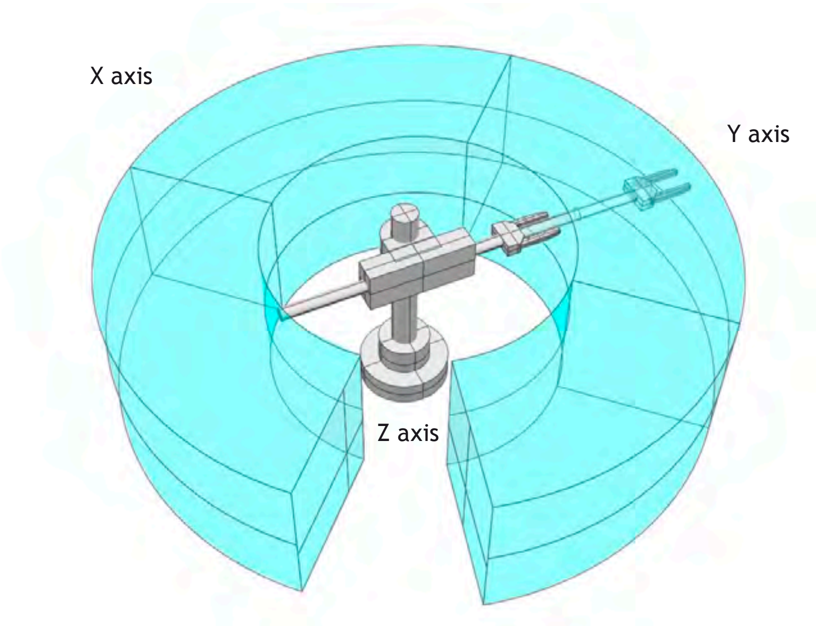
Table data	0	X axis encoder edges / mm
	1	Y axis encoder edges / mm
	2	Z axis encoder edges / mm
	3	Linkx (optional with 4 or 5 axis)
	4	Linky (optional with 4 or 5 axis)
	5	Linkz (optional with 4 or 5 axis)
	6	Encoder edges/radian (optional Z rotation)
	7	Encoder edges/radian (optional Y rotation)

FRAME 119

DESCRIPTION:

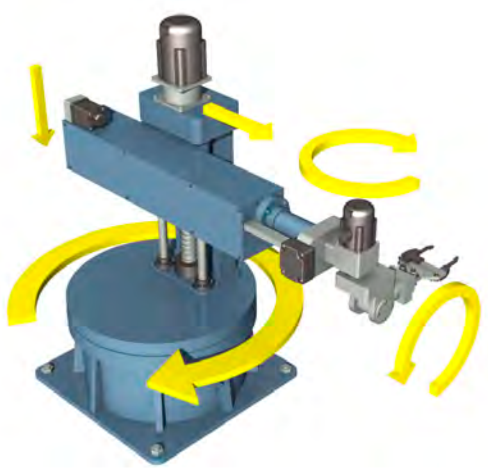
FRAME=119 enables the high accuracy transformation for a 3 axis cylindrical robot with a 2 axis wrist. It has optionally 3 to 5 axes which can be set by **FRAME_GROUP**.

FRAME=119 requires the kinematic runtime **FEC**



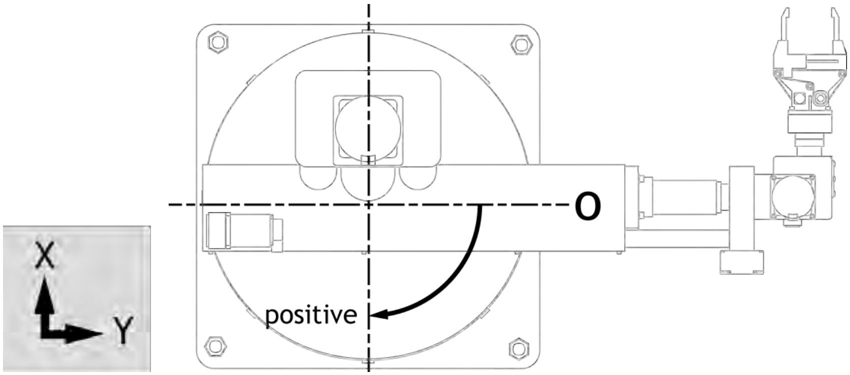
Once the **FRAME** is enabled set the **UNITS** to **FRAME_ANGLE_SCALE** so that the Cartesian movements use the same scale as that used in the table data. So if the **TABLE** data is programmed in mm then when **UNITS** is set to **FRAME_ANGLE_SCALE** then the robot can be programmed in mm.

The origin for the robot is the centre of the rotation axes.



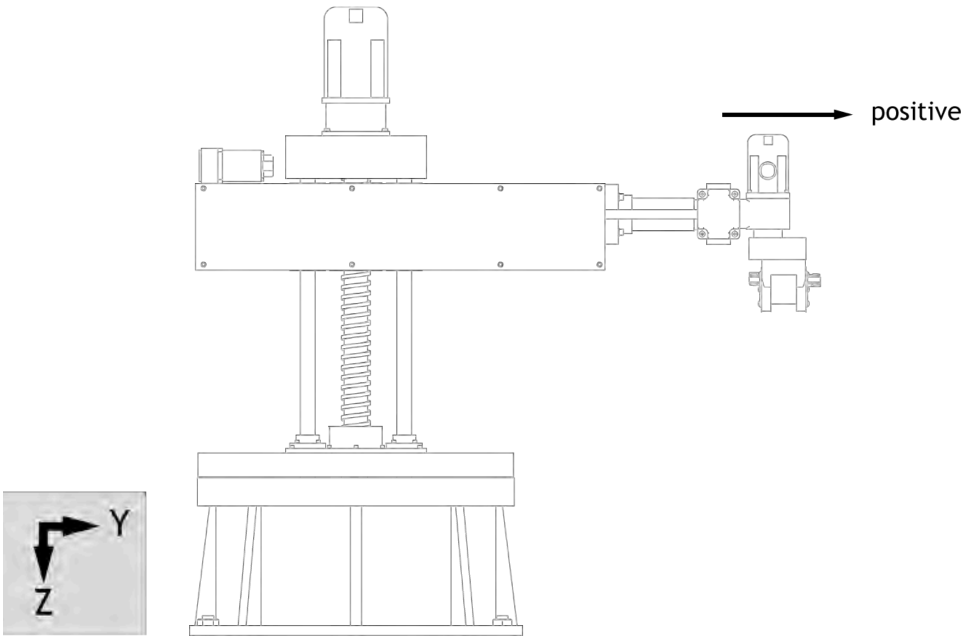
HOMING:

AXIS(0) – BASE ROTATION



Home so that the zero position is along the y axis
Positive direction is clockwise looking from above

AXIS(1) – ARM EXTENSION



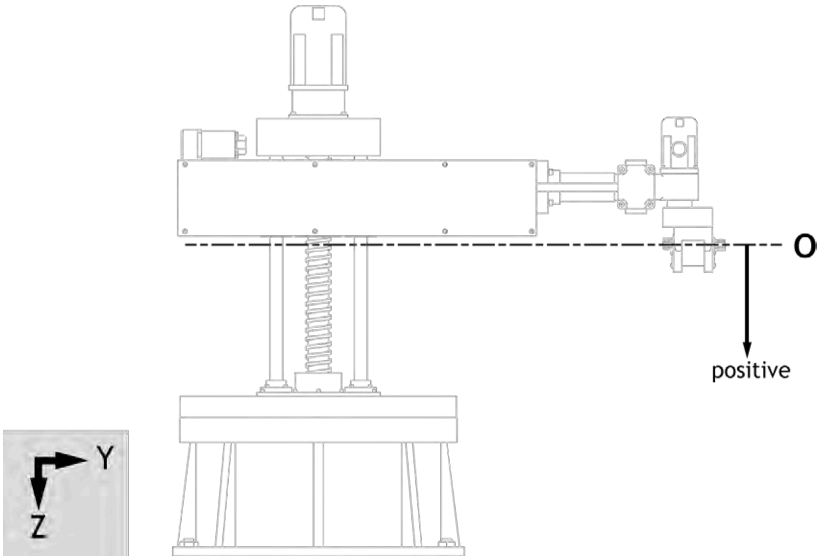
Home with arm at shortest position. Use **DEFPOS** to define the offset from the centre of rotation to the wrist

Positive direction is moving away from centre

Range: greater than zero

The arm extension must never be allowed to become zero or negative as this will result in a jump in motion. You can set your **RS_LIMIT** to prevent this situation.

AXIS(2) – VERTICAL AXIS

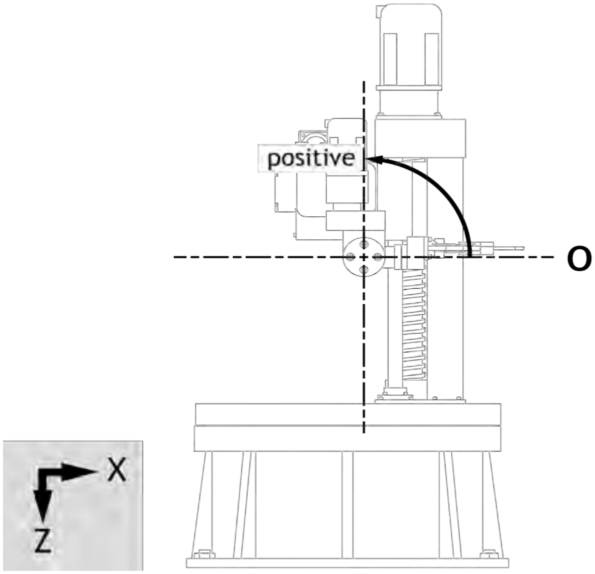


Home with zero at highest position

Positive direction is moving down

Range: 0 to infinite

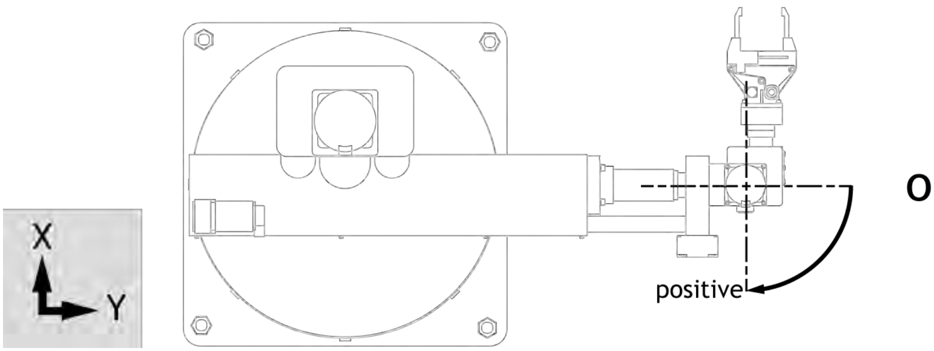
AXIS(3) – WRIST ROTATE ABOUT Y



Home so that the wrist is horizontal

Range: - infinite to infinite

AXIS(4) – WRIST ROTATE ABOUT Z



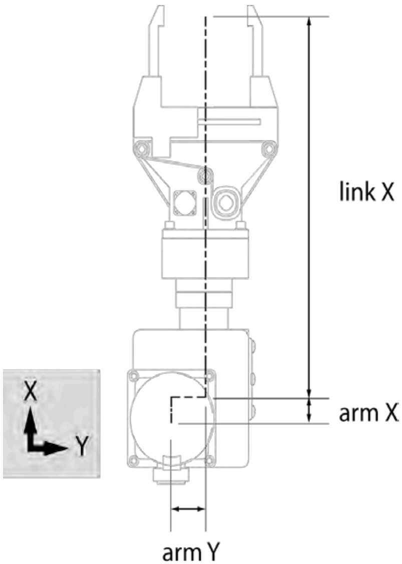
Home so that the zero position is along the y axis

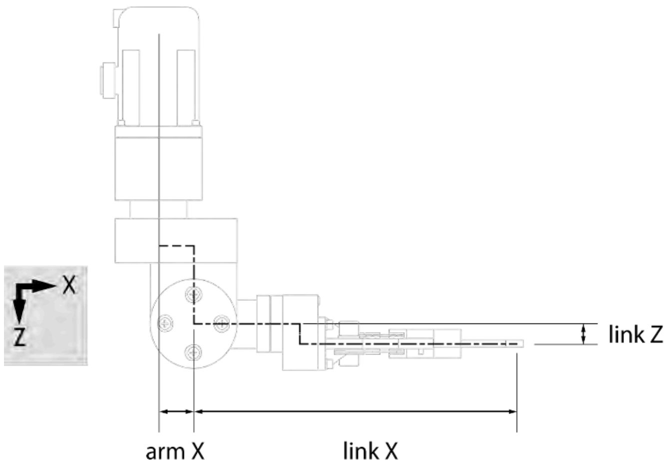
Positive direction is clockwise looking from above

Range: - infinite to infinite

PARAMETERS:

Table data	0	Edges per radian (base rotation)
	1	Edges per mm (arm extension)
	2	Edges per mm (vertical axis)
	3	Revolutions – set to 0
	4	Previous position – set to 0
	5	Linkx (optional with 4 or 5 axis)
	6	Linky (optional with 4 or 5 axis)
	7	Linkz (optional with 4 or 5 axis)
	8	Encoder edges/radian (optional Z rotation)
	9	Encoder edges/radian (optional Y rotation)





EXAMPLES:

EXAMPLE 1:

This example sets up a 5 axis system

```
    linkx = 50'mm
    linky = 50'mm
    linkz = 50'mm

    t1_encoder = 4*17000 'Encoder counts per revolution
    t1_gearbox = 50
    t1_edges_per_radian = t1_encoder * t1_gearbox / (2 * PI)
    t1_edges_per_degree = t1_encoder * t1_gearbox / (360)

    t2_encoder = 4*250 'Encoder counts per revolution
    t2_gearbox = 1
    t2_mm_per_rev = 1
    t2_edges_per_mm = t2_encoder * t2_gearbox / t2_mm_per_rev

    t3_encoder = 4*250 'Encoder counts per revolution
    t3_gearbox = 1
    t3_mm_per_rev = 1
    t3_edges_per_mm = t3_encoder * t3_gearbox / t3_mm_per_rev

    t4_encoder = 4*16000 'Encoder counts per revolution
    t4_gearbox = 50
    t4_edges_per_radian = t4_encoder * t4_gearbox / (2 * PI)
    t4_edges_per_degree = t4_encoder * t4_gearbox / (360)

    t5_encoder = 4*16000 'Encoder counts per revolution
    t5_gearbox = 50
    t5_edges_per_radian = t4_encoder * t4_gearbox / (2 * PI)
    t5_edges_per_degree = t4_encoder * t4_gearbox / (360)

    revolutions = 0
    prev_pos = 0

    group_size = 5

    TABLE(0, t1_edges_per_radian, t2_edges_per_mm, t3_edges_per_mm,
  revolutions, prev_pos)
    TABLE(5, linkx, linky, linkz, t4_edges_per_radian, t5_edges_per_
  radian)
  FRAME_GROUP(0,0,0,1,2,3,4)
  BASE(0)
    UNITS =FRAME_SCALE 'mm
    BASE(1)
    UNITS =FRAME_SCALE 'mm
    BASE(2)
    UNITS =FRAME_SCALE 'mm
```

```
BASE(3)
UNITS =(FRAME_SCALE * 2 * PI) / (360)'degrees
BASE(4)
UNITS =(FRAME_SCALE * 2 * PI) / (360)'degrees

BASE(0,1,2)
MOVE(-100,-100,100)
MOVE(200,0)
MOVE(-100,100,-100)
BASE(0,1,zrot)
MHELICAL(0,0,0,-50,0,360,1)
MOVE(0,25,0)
MOVECIRC(0,0,0,-75,0)
```

EXAMPLE 1:

This example sets up a 4 axis system

```
linkx = 50'mm
linky = 50'mm
linkz = 50'mm

t1_encoder = 4*17000 'Encoder counts per revolution
t1_gearbox = 50
t1_edges_per_radian = t1_encoder * t1_gearbox / (2 * PI)
t1_edges_per_degree = t1_encoder * t1_gearbox / (360)

t2_encoder = 4*250 'Encoder counts per revolution
t2_gearbox = 1
t2_mm_per_rev = 1
t2_edges_per_mm = t2_encoder * t2_gearbox / t2_mm_per_rev

t3_encoder = 4*250 'Encoder counts per revolution
t3_gearbox = 1
t3_mm_per_rev = 1
t3_edges_per_mm = t3_encoder * t3_gearbox / t3_mm_per_rev

t4_encoder = 4*16000 'Encoder counts per revolution
t4_gearbox = 50
t4_edges_per_radian = t4_encoder * t4_gearbox / (2 * PI)
t4_edges_per_degree = t4_encoder * t4_gearbox / (360)

revolutions = 0
prev_pos = 0

group_size = 4

TABLE(0, t1_edges_per_radian, t2_edges_per_mm, t3_edges_per_mm,
revolutions, prev_pos)
TABLE(5, linkx, linky, linkz, t4_edges_per_radian)
```

```
FRAME_GROUP(0,0,0,1,2,3)
BASE(0)
  UNITS =FRAME_SCALE 'mm
  BASE(1)
  UNITS =FRAME_SCALE 'mm
  BASE(2)
  UNITS =FRAME_SCALE 'mm
  BASE(3)
  UNITS =(FRAME_SCALE * 2 * PI) / (360)'degrees
```

FRAME_GROUP

TYPE:

System Command

SYNTAX:

FRAME_GROUP(group, [table_offset, [axis0, axis1 ...axisn]])

DESCRIPTION:

FRAME_GROUP is used to define the group of axes and the table offset which are used in a **FRAME** or **USER_FRAME** transformation. There are 8 groups available meaning that you can run a maximum of 8 FRAMEs on the controller.

FRAME_GROUP requires the kinematic runtime **FEC**

Although 8 **FRAME**s can be initialised on a controller it may not be possible to process all 8 at a given **SERVO_PERIOD**. The number that can be run depends on many factors including, which **FRAME** is selected, drive connection method, if **USER_FRAME** and **TOOL_OFFSET** are enabled and additional factory communications.

The number of axes in the group must match the number of axes used by the **FRAME**. The axes must also be ascending order though they do not have to be contiguous. If a group is deleted **FRAME** and **USER_FRAME** are set to 0 for those axes.

To maintain backward compatibility if the **FRAME** command is used on an axis that is not in a group, or no groups are configured then a default group is created using the lowest axes and table_offset=0. In this situation if **FRAME_GROUP**(0) is already configured it is overwritten.

When the group is deleted **FRAME** is set to 0, **USER_FRAME**(0) is activated, **TOOL_OFFSET**(0) is activated and **VOLUME_LIMIT**(0) is activated. This means you can delete the **FRAME_GROUP** to reset all of these commands.

PARAMETERS:

group:	The group number, 0-7. When used as the only parameter **FRAME_GROUP** prints the **FRAME_GROUP**, the active **USER_FRAME** and **TOOL_OFFSET** information to the currently selected output channel (default channel 0)
table_offset:	-1 = Delete group data
	0+ = The start position in the table to store the **FRAME** configuration.
axis0:	The first axis in the group
axis1:	The second axis in the group
axisn:	The last axis in the group

The text returned when only printing **FRAME_GROUP** is in the following format:

group [table_offset] : axes {**USER_FRAME**: **USER_FRAME** parameters} TO={**TOOL_OFFSET** : **TOOL_OFFSET** parameters} VL={**VOLUME_LIMIT** parameters}

EXAMPLES:

EXAMPLE 1:

Configure a **FRAME_GROUP** for axes 1,2 and 5 using table offset 100.

```
'Initialise the FRAME_GROUP
FRAME_GROUP(0,100, 1,2,5)

'Configure the axes, FRAME table data and home the robot
GOSUB configure_frame

'PRINT the FRAME_GROUP information to the command line
FRAME_GROUP(0)

'Enable the frame
FRAME AXIS(1)=14
```

EXAMPLE 2:

Reset the **FRAME_GROUP** to set: **USER_FRAME**(0), **TOOL_OFFSET**(0), **FRAME** = 0 and **VOLUME_LIMIT**(0)

```
BASE(0) 'Select an axis in the FRAME_GROUP
FRAME_GROUP(0,-1)
```

EXAMPLE 3:

Print the **FRAME_GROUP** in the terminal.

```
>>FRAME_GROUP(0,1,2,3,4,5)
>>PRINT FRAME_GROUP(0)
0 [1] : 2, 3, 4, 5 {0:0.00000, 0.00000, 0.00000, 0.00000, 0.00000,
0.00000} TO={0
```

```
:0.00000, 0.00000, 0.00000} VL={0, 0}
0
```

FRAME_REP_DIST

TYPE:

Axis Parameter

DESCRIPTION:

Orientation axes on a **FRAME** or **USER_FRAME** must be programmed between ± half a revolution (**UNITS** can be used to set radians, degrees etc). This cannot be done using **REP_DIST** and has to be done using **FRAME_REP_DIST** and **REP_OPTION** bit 3.

When this is configured the **DPOS** will wrap to ± half a revolution and **AXIS_DPOS** will not be wrapped so that the absolute axis position is maintained.

Wrapping will only occur when **FRAME** <> 0 or **USER_FRAME** <> 0. While both are set to zero the wrapping will be inhibited so that the absolute axis position is maintained.

With **REP_OPTION** bit 3 set and **DPOS** exceeding **FRAME_REP_DIST** it will wrap to –**FRAME_REP_DIST**. The same applies in reverse so when **DPOS** exceeds -**FRAME_REP_DIST** it will wrap to **FRAME_REP_DIST**.

VALUE:

The position in user **UNITS** where the axis position wraps.

FRAME_REP_DIST uses *UNITS*. You must remember to set **FRAME_REP_DIST** while the correct *UNITS* are active.

EXAMPLES:

A 4 axis delta robot has one orientation axis which is the angle of rotation about the Z axis. The user is programming in degrees so the **DPOS** must be limited to ±180 degrees.

```
BASE(axis_w)
UNITS = (FRAME_SCALE*2*PI) / 360 ‘degrees
FRAME_REP_DIST = 180
REP_OPTION = 8
```

SEE ALSO:

REP_OPTION

FRAME_ SCALE

TYPE:
Axis Parameter

DESCRIPTION:
FRAME_ SCALE is used to adjust the resolution of the high accuracy FRAMEs (100+). The default value is very large and so the accuracy is sufficient for most applications.

VALUE:
Default value 1000000000

FRAME_TRANS

TYPE:
Mathematical Function

SYNTAX:
FRAME_TRANS(frame, table_in, table_out, direction [,table_offset])

DESCRIPTION:
This function enables you to perform both the forward and inverse transformation calculations of a **FRAME**. One particular use is to check following errors in user units or to calculate positions outside of the **FRAME** working area.

FRAME_TRANS requires the kinematic runtime **FEC** to use a **FRAME** 14 and higher.

The **FRAME** calculations are performed on raw position data. When using a **FRAME** typically the raw position data for **DPOS** is micrometres and the raw position data for **MPOS** is encoder counts but this can vary depending on which **FRAME** you select.

PARAMETERS:

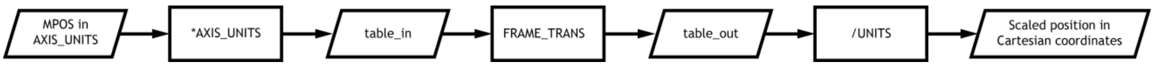
frame:	The **FRAME** number to run

table_in	The start position in the **TABLE** of the input positions
table_out	The start position in the **TABLE** of the generated positions
direction	1 = **AXIS_DPOS** to **DPOS** (Forward Kinematics)
	0 = **DPOS** to **AXIS_DPOS** (Inverse Kinematics)
table_offset	The first position in the table where the frame configuration is found (default 0)

EXAMPLES:

EXAMPLE 1:

Using **MPOS** calculate the Cartesian values so you can compare them to **DPOS**. This can be used to check the following error in the world coordinate system. The frame configuration is stored in the table starting at position 100.




```
'Load positions into the table
FOR x=0 TO 3
BASE(x)
TABLE(1000+x,MPOS AXIS(x)*UNITS AXIS(x))
NEXT x
'Calculate forward transform to see MPOS is Cartesian coordinates
FRAME_TRANS(15, 1000,2000,1,100)

TABLE(3000, TABLE(2000)/ UNITS AXIS(0))
TABLE(3001, TABLE(2001)/ UNITS AXIS(1))
TABLE(3002, TABLE(2002)/ UNITS AXIS(2))
PRINT «DPOS IN ENCODER COUNTS»,TABLE(2000),TABLE(2001),TABLE(2002)
PRINT «DPOS IN MM»,TABLE(3000),TABLE(3001),TABLE(3002)
PRINT «FE in world x = «, TABLE(3000) – DPOS AXIS(0)
PRINT «FE in world y = «, TABLE(3001) – DPOS AXIS(1)
PRINT «FE in world z = «, TABLE(3002) – DPOS AXIS(2)
```

EXAMPLE 2:

Use the inverse kinematics to confirm that a demand position will result in an axis position that the motors can achieve.


```
'Load positions into the table
TABLE(5000,100*UNITS AXIS(0),200*UNITS AXIS(1),400*UNITS AXIS(2))
```

```
‘Calculate reverse transform to see
FRAME_TRANS(14, 5000,6000,0)

‘Divide the result by the AXIS_UNITS to get
‘the MPOS in degrees
TABLE(7000, TABLE(6000)/ AXIS_UNITS)
TABLE(7001, TABLE(6001)/ AXIS_UNITS)
TABLE(7002, TABLE(6002)/ AXIS_UNITS)

PRINT “MPOS RAW ENCODER COUNTS”, TABLE(6000),TABLE(6001),TABLE(6002)
PRINT “MPOS degrees”, TABLE(7000),TABLE(7001),TABLE(7002)
```

FREE

TYPE:

System Parameter (Read Only)

DESCRIPTION:

Returns the amount of program memory available for user programs.

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

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

VALUE:

The amount of available user memory in bytes.

EXAMPLE:

Check the available memory on the command line

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

SEE ALSO:

DIR

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.

Bit 9 of the **AXISSTATUS** register is set when the axis position is greater than the **FS_LIMIT**.

When **DPOS** reaches **FS_LIMIT** the controller will cancel the move, so the axis will decelerate at **DECEL** or **FASTDEC**.

FS_LIMIT is disabled when it has a value greater than **REP_DIST**.

VALUE:
The absolute position of the software forward travel limit in user **UNITS**. (default = 200000000000)

EXAMPLES:

EXAMPLE 1:
Datum axis 1, then define a forward limit from this point.

```
BASE(1)
DATUM(3)
WAIT IDLE
FS_LIMIT=200
```

EXAMPLE 2:
Disable the **FS_LIMIT** by setting it greater than **REP_DIST**.

```
FS_LIMIT = REPDIST+10
```

SEE ALSO:
RS_LIMIT, FWD_IN, REV_IN

FULL_SP_RADIUS

TYPE:
Axis Parameter

DESCRIPTION:
This parameter is used with **CORNER_MODE**, it defines the minimum radius that will be executed at full speed. When a radius is smaller than **FULL_SP_RADIUS** the speed will be proportionally reduces so that:

VP_SPEED = **FORCE_SPEED** * radius/**FULL_SP_RADIUS**

Where radius is the radius of the corner that is executing.

VALUE:
The full speed radius in user **UNITS** (default = 0).

EXAMPLE:
In the following program, when the first **MOVECIRCSP** is reached the speed remains at 10 because the radius (8) is greater than that set in **FULL_SP_RADIUS**. For the second **MOVECIRCSP** the speed is reduced by 50% to a value of 5, because the radius is 50% of that stored in **FULL_SP_RADIUS**.

```
CORNER_MODE=8
MERGE=ON
SPEED=10
FULL_SP_RADIUS=6
DEFPOS(0,0)

MOVESP(10,10)
MOVESP(10,5)
MOVESP(5,5)
MOVECIRCSP(8,8,0,8,1)
MOVECIRCSP(3,3,0,3,1)
MOVESP(5,5)
MOVESP(10,5)
```

SEE ALSO:
CORNER_MODE

FWD_IN

TYPE:
Axis Parameter

DESCRIPTION:

This parameter holds the input number to be used as a forward limit input.

When the forward limit input is active any motion on that axis is **CANCELed.**

When **FWD_IN** is active **AXISSTATUS** bit 4 is set.

The input used for **FWD_IN** is active low.

When the forward limit input is active the controller will cancel the move, so the axis will decelerate at **DECEL** or **FASTDEC**.

VALUE:

-1	Disable the input as **FWD_IN** (default)
0-63	Input to use as forward input switch

Any type of input can be used, built in, Trio CAN I/O, CANopen or virtual.

EXAMPLE:

Initialise input 19 for the forward limit switch

```
FWD_IN AXIS(9)=19
```

SEE ALSO:

REV_IN, FS_LIMIT, RS_LIMIT

FWD_JOG

TYPE:

Axis Parameter

DESCRIPTION:

This parameter holds the input number to be used as a jog forward input.

When the **FWD_JOG** input is active the axis moves forward at **JOGSPEED**.

The input used for *FWD_IN* is active low.

It is advisable to use *INVERT_IN* on the input for **FWD_JOG** so that 0V at the input disables the jog.

FWD_JOG overrides **REV_JOG** if both are active

VALUE:

-1	Disable the input as **FWD_JOG** (default)
0-63	Input to use as datum input

EXAMPLE:

Initialise the **FWD_JOG** so that it is active high on input 7

```
INVERT_IN(7,ON)
FWD_JOG=7
```

GET

TYPE:
System Command

SYNTAX:
GET [#channel,] variable

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

Poll **KEY** to check to if a character has been received before performing a **GET**.

PARAMETERS:

#channel:	See # for the full channel list (default 0 if omitted)
variable:	The variable to store the received character, this may be local variable, **VR** or **TABLE**

Performing a **GET** or **GET**#0 will suspend the command line until a character is sent on that channel.

EXAMPLES:

EXAMPLE 1:
Ask a user to enter ‘y’ for yes or ‘n’ for no on channel 5

```
start:
  PRINT#5, “Press ‘y’ for YES or ‘n’ for NO.”
  GET#5, char
  IF char = 121 THEN
    PRINT#5, “YES selected”
  ELSEIF char = 110 THEN
    PRINT#5, “NO selected”
  ELSE
    PRINT#5, “BAD selection”
    GOTO start
  ENDIF
```

EXAMPLE 2:
Clear the serial buffer then request the user to enter a name

```
WHILE KEY#2
```

```
  GET#2, dump
WEND

PRINT#2, "ENTER NAME"
WAIT UNTIL KEY#2
count=0
WHILE char<> $D 'carrage return
  GET#2, char
  VR(count)=char
  count=count+1
WEND
```

SEE ALSO:

LINPUT, PRINT, KEY

GLOBAL

TYPE:

System Command

SYNTAX:

GLOBAL "name", vr_number

DESCRIPTION:

Up to 1024 GLOBALs can be declared in the controller, these are available to all programs. **GLOBAL** 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.

They should be declared on startup and for fast startup the program declaring **GLOBAL**s should also be the **ONLY** process running at power-up.

Once a **GLOBAL** has been assigned it cannot be changed, even if you change the program that assigns it.

While developing you may wish to clear or change a **GLOBAL**. You can clear a single **GLOBAL** by using the first parameter alone. All GLOBALs can be cleared by issuing **GLOBAL**. You can view all **GLOBALS** using **LIST_GLOBAL**.

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:

Initialise two GLOBALs and use then to adjust machine parameters.

```
GLOBAL "screw_pitch",12
GLOBAL "ratio1",534

ratio1 = 3.56
screw_pitch = 23.0
PRINT screw_pitch, ratio1
```

SEE ALSO:

CONSTANT, LIST_GLOBAL

GOSUB..RETURN

TYPE:

Program Structure

SYNTAX:

```
GOSUB label
…
label:
  commands
RETURN
```

DESCRIPTION:

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

GOSUB..**RETRUN** loops can be nested up to 8 deep in each program.

PARAMETERS:

commands:	TrioBASIC statements that you wish to execute
label:	A valid label that occurs in the program.

If the label does not exist an error message will be displayed at run time and the program execution halted.

You must not execute a **RETURN** without a **GOSUB** as a runtime error will be displayed and your program will stop.

EXAMPLES:

EXAMPLE 1:

```
      WHILE machine_active
        GOSUB routine1
        GOSUB routine2
      WEND
      STOP 'prevents running into subroutines when machine stopped.

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

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

EXAMPLE 2:

Calculating values in a subroutine.

```
    y=1
    z=4
    GOSUB calc
    PRINT "New value = ", x
    STOP

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

SEE ALSO:

GOTO

GOTO

TYPE:

Program Structure

SYNTAX:

GOTO label

```
…
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 at run time and the program execution halted.

EXAMPLE:
Use a **GOTO** to repeat a section of your program after a bad input

```
start:
PRINT#5, “Press ‘y’ for YES and ‘n’ for NO.”
GET#5,   char
IF char = 121 THEN
  PRINT#5, “YES selected”
ELSEIF char = 110 THEN
  PRINT#5, “NO selected”
ELSE
  PRINT#5, “BAD selection”
  GOTO start
ENDIF
```

SEE ALSO:
GOSUB

> Greater Than

TYPE:
Comparison Operator

SYNTAX:

```
<expression1> > <expression2>
```

DESCRIPTION:
Returns **TRUE** if expression1 is greater than expression2, otherwise returns **FALSE**.

PARAMETERS:

Expression1:	Any valid TrioBASIC expression
Expression2:	Any valid TrioBASIC expression

EXAMPLES:

EXAMPLE 1:
The program will wait until the measured position is greater than 200

```
WAIT UNTIL MPOS>200
```

EXAMPLE 2:
Set the value of **TRUE** into **VR** 0 as 1 is greater than 0

```
VR(0)=1>0
```

>= Greater Than or Equal

TYPE:
Comparison Operator

SYNTAX
<expression1> >= <expression2>

DESCRIPTION:
Returns **TRUE** if expression1 is greater than or equal to expression2, otherwise returns **FALSE**.

PARAMETERS:

Expression1:	Any valid TrioBASIC expression
Expression2:	Any valid TrioBASIC expression

EXAMPLE:
If variable target holds a value greater than or equal to 120 then move to the absolute position of 0.

```
IF target>=120 THEN MOVEABS(0)
```

HALT

TYPE:
System Command.

DESCRIPTION:
Halts execution of all running programs. You can use **HALT** in a program.

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

EXAMPLE:
Use the command line to stop two running programs:

```
>>HALT%[Process 20:Line 2] (31) - Program is stopped
%[Process 21:Line 1] (31) - Program is stopped
>>
```

SEE ALSO:
CANCEL, RAPIDSTOP, STOP

Hash

TYPE:
Special Character

SYNTAX:
command #channel

DESCRIPTION:
The # symbol is used to specify a communications channel to be used for serial input/output commands.

PARAMETERS:

Channel	Device
0	Ethernet port 0 (the command line)
1	RS232 port 1
2	RS485 port 2

Channel	Device
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
40	Channel configured using the **OPEN** command
41	Channel configured using the **OPEN** command
42	Channel configured using the **OPEN** command
43	Channel configured using the **OPEN** command
44	Channel configured using the **OPEN** command
45-49	Reserved
50	1st Anybus module
51	2nd Anybus module
52	3rd Anybus module
53	4th Anybus module
54	5th Anybus module
55	6th Anybus module
56	7th Anybus module

Channels 5 to 9 are logical channels which are superimposed on to Port 0 by *Motion* Perfect.

EXAMPLES:

EXAMPLE 1:

Printing Ascii strings to different channels

```
PRINT #1,"Printing data to RS232 Channel"
PRINT #5,"Printing data to Motion Perfect Terminal 5"
```

EXAMPLE 2:

Checking for and receiving characters on Channel 6

```
WHILE KEY #6
  GET #63, VR(123)
```

```
WEND
```

SEE ALSO:
GET, KEY, LINPUT, OPEN, PRINT

HEX

TYPE:
String Function

SYNTAX:
value = HEX(number)

DESCRIPTION:
HEX returns the hexadecimal value for the decimal number supplied as a **STRING** which can be assigned to a **STRING** variable or be PRINTed.

PARAMETERS:

number:	A decimal value
value:	A hexadecimal **STRING** of the number

EXAMPLES:

EXAMPLE 1:
Print **AXISSTATUS** as a hexadecimal value on the command line

```
>>PRINT HEX(AXISSTATUS)
10
>>
```

EXAMPLE 2:
Append a hexadecimal number to a **STRING** variable

```
DIM value AS STRING
value = value + HEX(number)
```

SEE ALSO:
PRINT, STRING

HLM_COMMAND

TYPE:
Remote Command

SYNTAX:
`HLM_COMMAND(command, port[, node[, mc_area/mode[, mc_offset ]]])`

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

PARAMETERS:

command:	The the Host Link operation to perform:		
	HLM_MREAD	0	This performs the Host Link PC **MODEL READ** (MM) command to read the CPU Unit model code. The result is written to the MC Unit variable specified by mc_area and mc_offset.
	HLM_TEST	1	This performs the Host Link **TEST** (TS) command to check correct communication by sending string “MCxxx **TEST STRING**” and checking the echoed string. Check the **HLM_STATUS** parameter for the result.
	HLM_ABORT	2	This performs the Host Link **ABORT** (XZ) command to abort the Host Link command that is currently being processed. The **ABORT** command does not receive a response.
	HLM_INIT	3	This performs the Host Link **INITIALIZE** (**) command to initialize the transmission control procedure of all Slave Units.
	HLM_STWR	4	This performs the Host Link **STATUS WRITE** (SC) command to change the operating mode of the CPU Unit.
port:	The specified serial port. (See specific controller specification for numbers)		
node:	(for **HLM_MREAD**, **HLM_TEST**, **HLM_ABORT** and **HLM_STWR**):		
	The Slave node number to send the Host Link command to. Range: [0, 31].		

mode:	(for **HLM_STWR**)		
	The specified CPU Unit operating mode.		
	0	**PROGRAM** mode	
	2	**MONITOR** mode	
	3	RUN mode	
mc_area:	(for **HLM_MREAD**)		
	The MC Unit's memory selection to write the received data to.		
	MC_TABLE	8	Table variable array
	MC_VR	9	Global (VR) variable array
mc_offset:	(for **HLM_MREAD**)		
	The address of the specified MC Unit memory area to read from.		

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

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

EXAMPLES:

EXAMPLE 1:

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

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

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

EXAMPLE 2:

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

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

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

EXAMPLE 3:

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

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

EXAMPLE 4:

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

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

HLM_READ

TYPE:

Remote Command

SYNTAX:

`HLM_READ(port,node,pc_area,pc_offset,length,mc_area,mc_offset)`

DESCRIPTION:

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

PARAMETERS:

port:	The specified serial port. (See specific controller specification for numbers)			
node:	The Slave node number to send the Host Link command to. Range: [0, 31].			
pc_area:	The PC memory selection for the Host Link command.			
	pc_area		data area	Hostlink command
	PLC_DM	0	DM	RD
	PLC_IR	1	CIO/IR	RR
	PLC_LR	2	LR	RL
	PLC_HR	3	HR	RH
	PLC_AR	4	AR	RJ
	PLC_EM	6	EM	RE
pc_offset:	The address of the specified PC memory area to read from. Range: [0, 9999].			

length:	The number of words of data to be transfered. Range: [1, 30].		
mc_area:	The MC Unit's memory selection to write the received data to.		
	MC_TABLE	8	Table variable array
	MC_VR	9	Global (VR) variable array
mc_offset:	The address of the specified MC Unit memory area to write to.		

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

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

HLM_STATUS

TYPE:
Port Parameter

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

HLM_TIMEOUT

TYPE:
System Parameter

DESCRIPTION:
Sets the timeout value for Hostlink communications.

VALUE:
Timeout in msec, default 500msec

EXAMPLE:
Set the Hostlink timeout to 600msec.

```
HLM_TIMEOUT = 600
```

HLM_WRITE

TYPE:

Remote Command

SYNTAX:

`HLM_WRITE(port,node,pc_area,pc_offset,length,mc_area,mc_offset)`

DESCRIPTION:

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

PARAMETERS:

port:	The specified serial port. (See specific controller specification for numbers)			
node:	The Slave node number to send the Host Link command to. Range: [0, 31].			
pc_area:	The PC memory selection for the Host Link command.			
	pc_area		data area	Hostlink command
	PLC_DM	0	DM	RD
	PLC_IR	1	CIO/IR	RR
	PLC_LR	2	LR	RL
	PLC_HR	3	HR	RH
	PLC_AR	4	AR	RJ
	PLC_EM	6	EM	RE
	PLC_REFRESH	7		
pc_offset:	The address of the specified PC memory area to write to. Range: [0, 9999].			
length:	The number of words of data to be transfered. Range: [1, 30].			

mc_area:	The MC Unit's memory selection to read the data from.		
	MC_TABLE	8	Table variable array
	MC_VR	9	Global (VR) variable array
mc_offset:	The address of the specified MC Unit memory area to read from.		

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

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

EXAMPLE:

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

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

HLS_MODEL

TYPE:

System Parameter

DESCRIPTION:

Defines the model number returned to a Hostlink Master.

VALUE:

The model number returned. Default 250

HLS_NODE

TYPE:

System Parameter

DESCRIPTION:

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

HMI_CONNECTIONS

TYPE:
System Parameter

SYNTAX:
HMI_CONNECTIONS

DESCRIPTION:
Return the connection strings for all currently connected clients.

VALUE:

value	A string that contains the connection strings for all the connected clients. Each connection string is on a separate line. Each line has the following structure: <session>;<major>;<minor>;<ip>;<platform>;<osversion>;<window> Where: <session> is the corresponding session id (0, 1, ...) <major> is the major version of the HMI Client <minor> is the minor version of the HMI Client <ip> is the IP address of the HMI Client <platform> is the definition of the hardware the HMI Client is running on. 1 => WindowsCE 2 => Windows Desktop <osversion> is the version reported by the platform. The major version number is stored in the most significant byte and the minor version number is stored in the least significant byte. <window> is the size of the HMI Client screen. The width is stored in the most significant byte and the height is stored in the least significant byte.

EXAMPLE:
Report the currently connected HMI Clients.

```
>>PRINT HMI_CONNECTIONS
0;1.22.4.502;127.0.0.1;2;60001;32001e0
1;1.22.3.500;192.168.2.53;1;50000;32001e0
```

SEE ALSO:
HMI_GET_PAGE, HMI_GET_STATUS, HMI_SERVER, HMI_SET_PAGE

HMI_GET_PAGE

TYPE:
System Function

SYNTAX:
`value = HMI_GET_PAGE[(<ip>)]`

DESCRIPTION:
Return the currently selected page on the given HMI Client. If the IP address is not specified then the current page for the lowest active session will be returned.

PARAMETERS:

value	A string that contains the name of the current page on the HMI Client.
IP	IP address of the HMI Client to which this message must be sent.

EXAMPLE:
Automatically reset the current page on the HMI Client.

```
WHILE(1)
  IF VR(0)<>0 AND HMI_GET_PAGE<>"PAGE1" THEN
    HMI_SET_PAGE("PAGE1")
    VR(0)=0
  ENDIF
WEND
```

SEE ALSO:
HMI_CONNECTIONS, HMI_GET_STATUS, HMI_SERVER, HMI_SET_PAGE

HMI_GET_STATUS

TYPE:
System Function

SYNTAX:
`value = HMI_GET_STATUS[(<ip>)]`

DESCRIPTION:
Return the status of the given HMI Client. If the IP address is not specified then the current page for the

lowest active session will be returned.

PARAMETERS:

value	-1	HMI Client is not connected
	1	HMI Client is Connected
	2	HMI Page is loading
	3	HMI Page is running
	4	HMI Client is in error
IP	IP address of the HMI Client to which this message must be sent.	

EXAMPLE:

Wait for the HMI Client to initialise correctly, change to the start page and wait for the change to complete.

```
WAIT UNTIL HMI_GET_STATUS=3
HMI_SET_PAGE("START")
WAIT UNTIL HMI_GET_STATUS=3 AND HMI_GET_PAGE="START"
```

SEE ALSO:

HMI_CONNECTIONS, HMI_GET_PAGE, HMI_SERVER, HMI_SET_PAGE

HMI_PROC

TYPE:

System Parameter (**MC_CONFIG**)

SYNTAX:

HMI_PROC=value

DESCRIPTION:

Sets the process number on which the HMI Server protocol will be initiated. This value must be set before the first HMI Client connection occurs. The default value at power up is -1, which will automatically select the process number according to the normal RUN command rules.

If this value is to be set, then it is recommended that it be set in the special **MC_CONFIG** program to insure that the value is valid before any HMI Client can connect to the *Motion Coordinator*.

HMI_SERVER

TYPE:
System Command

SYNTAX:
`HMI_SERVER[ (function [, parameters…])]`

DESCRIPTION:
This command allows the Trio HMI Server to be controlled, configured and interrogated from a TrioBASIC program.

If there are no parameters then the function is 0, and the parameter is 0.

PARAMETERS:

Function	0	Run the **HMI_SERVER** protocol
	1	Read the HMI Client error data
	2	Write the **HMI_SERVER** event flags
	3	Read the **HMI_SERVER** status data
	4	Set the HMI poll timeout
	5	Read the HMI Client version information

FUNCTION = 0:

SYNTAX:
`HMI_SERVER`
`HMI_SERVER(0[,debug])`

DESCRIPTION:
This function starts the **HMI_SERVER** protocol. This function never stops, so no TrioBASIC statement after this command in a program will be executed.

The **HMI_SERVER** program is normally started automatically when the **HMI** Client connects to the *Motion Coordinator*. You can call it manually if you wish to specify which process it should run on and whether it should print debug information.

If you execute **HMI_SERVER** manually the program it runs in will suspend at the **HMI_SERVER** line. The **HMI_SERVER** therefore should be the last line of the program to execute.

PARAMETERS:

Debug	0	No debug information
	1	Debug information printed to channel 0 (only use when requested by Trio)

FUNCTION = 1:

SYNTAX:

```
value = HMI_SERVER(1, error_parameter)
```

DESCRIPTION:

When an error occurs in the HMI Client, this event is sent to the HMI Server if possible. This command will return the data about the last error that occurred in the HMI Client.

PARAMETERS:

error_parameter	0	Error number	Specific to the HMI Client operating system
	1	Error string	Specific to the HMI Client operating system
	2	Error program	When applicable, the name of the program on the *Motion Coordinator* with which the HMI Client was communicating when the error occurred.
	3	Error process	When applicable, the process number of the program on the *Motion Coordinator* with which the HMI Client was communicating when the error occurred.

EXAMPLE:

Report an error on the HMI Client

```
'Check for error
IF HMI_SERVER(1,0) THEN
    PRINT "HMI Client reports error"
    PRINT "HMI Error=";HMI_SERVER(1,0)
    PRINT "HMI Description=";HMI_SERVER(1,1)
    PRINT "MC Program=";HMI_SERVER(1,2)
    PRINT "MC Process=";HMI_SERVER(1,3)
ENDIF
```

FUNCTION = 2:

SYNTAX:

`HMI_SERVER(2, parameter [, string [, client_ip]])`

DESCRIPTION:

The HMI Server can inform the HMI Client that certain events have occurred. These events are used by MotionPerfectV3. The optional client_ip is currently ignored by the **HMI_SERVER** command. The string parameter depends on value of parameter.

PARAMETERS:

parameter	0	No event
	1	The *Motion Coordinator* has an updated HMI Design file, the HMI Client must request it. String is the name of the file on the *Motion Coordinator* to be read.
	2	Request that the HMI Client send its' current configuration file. String is the name on the *Motion Coordinator* of the file to be written.
	4	The *Motion Coordinator* has an updated HMI configuration file, the HMI Client must request it. String is the name of the file on the *Motion Coordinator* to be read.
	8	The *Motion Coordinator* has an updated HMI Client firmware file, the HMI Client must request it. String is the name of the file on the *Motion Coordinator* to be read.
	32	Set the current page on the HMI Client, the next parameters specifies the page name. String is the name of the page to be selected.

EXAMPLE:

Automatically scroll through three pages at a time interval of 5 seconds. If a page is manually selected then hold a page for 30 seconds. The page value is set from the HMI to a value greater than 3 to put the page on manual mode.

```
page = 0
page_time = 5000
manual_time = 30000

WHILE(1)
  IF page = 0 THEN
    HMI_SERVER(2,32,”PAGE1”)
    page = 1
    WA(page_time)
  ELSEIF page = 1 THEN
    HMI_SERVER(2,32,”PAGE2”)
    page = 2
    WA(page_time)
  ELSEIF page = 2 THEN
    HMI_SERVER(2,32,”PAGE3”)
```

```
        page = 3
        WA(page_time)
      ELSE
        ‘in manual mode
        page = 0
        TICKS = manual_time
        WHILE TICKS>0
          IF page <> 0 THEN
            TICKS = manual_time
            page = 0
          ENDIF
          WA(1)
        WEND
      ENDIF
    WEND
```

FUNCTION = 3:

SYNTAX:

`value = HMI_SERVER(3, parameter, return_type)`

DESCRIPTION:

Read the HMI Client status information.

PARAMETERS:

parameter	0	Client status:	
		0	Disconnected
		1	Connected
		2	HMI page loaded
		3	Running
		4	In error
	1	Current HMI Design page	
return_type	0	Integer	
	1	String	

FUNCTION = 4:

SYNTAX:

```
HMI_SERVER(4, parameter)
```

DESCRIPTION:

Set the number of milliseconds without activity that the HMI Server will wait before aborting a client connection.

FUNCTION = 5:

SYNTAX:

```
value = HMI_SERVER(5, parameter)
```

DESCRIPTION:

Return the HMI Client description. The HMI Client sends this data to the HMI Server during the protocol initialisation.

PARAMETERS:

<table>
<tr><td rowspan="13">parameter</td><td>0</td><td colspan="2">HMI Client Engine major version number</td></tr>
<tr><td>1</td><td colspan="2">HMI Client Engine minor version number</td></tr>
<tr><td>2</td><td colspan="2">HMI Client Communications Protocol major version number</td></tr>
<tr><td>3</td><td colspan="2">HMI Client Communications Protocol minor version number</td></tr>
<tr><td rowspan="3">4</td><td colspan="2">HMI Client OS ID:</td></tr>
<tr><td>0</td><td>Windows CE</td></tr>
<tr><td>1</td><td>Windows Desktop</td></tr>
<tr><td rowspan="3">5</td><td colspan="2">HMI Client OS Version:</td></tr>
<tr><td>Bit 0-15</td><td>Minor number</td></tr>
<tr><td>Bit 16-31</td><td>Major number</td></tr>
<tr><td rowspan="3">6</td><td colspan="2">HMI Client Canvas Size:</td></tr>
<tr><td>Bit 0-15</td><td>Width in pixels</td></tr>
<tr><td>Bit 16-31</td><td>Height in pixels</td></tr>
</table>

SEE ALSO:

HMI_CONNECTIONS, HMI_GET_PAGE, HMI_GET_STATUS, HMI_SET_PAGE

HMI_SET_PAGE

TYPE:

System Command

SYNTAX:

`HMI_SET_PAGE(<name>[,<ip>])`

DESCRIPTION:

Request that the HMI Client change to the given page. If the IP address is not specified the request will be sent to all currently connected clients. This command will wait for all pending HMI Client requests to complete before submitting the new request, but it will not wait for the HMI Client to complete the request. This means the controller will continue to run the software without waiting for the requested page to show on the HMI Client.

PARAMETERS:

name	Name of the page in the HMI Design on the HMI Client. This name is case sensitive.
IP	IP address of the HMI Client to which this message must be sent.

EXAMPLE:

Automatically scroll through three pages at a time interval of 5 seconds. If a page is manually selected then hold a page for 30 seconds. The page value is set from the HMI to a value greater than 3 to put the page on manual mode.

```
page = 0
page_time = 5000
manual_time = 30000

WHILE(1)
  IF page = 0 THEN
    HMI_SET_PAGE("PAGE1")
    page = 1
    WA(page_time)
  ELSEIF page = 1 THEN
    HMI_SET_PAGE("PAGE2")
    page = 2
    WA(page_time)
  ELSEIF page = 2 THEN
    HMI_SET_PAGE("PAGE3")
    page = 3
    WA(page_time)
  ELSE
    'in manual mode
```

```
        page = 0
        TICKS = manual_time
        WHILE TICKS>0
          IF page <> 0 THEN
            TICKS = manual_time
            page = 0
          ENDIF
          WA(1)
        WEND
      ENDIF
    WEND
```

SEE ALSO:
HMI_CONNECTIONS, HMI_GET_PAGE, HMI_GET_STATUS, HMI_SERVER

HW_PSWITCH

TYPE:
Axis command

SYNTAX:
`HW_PSWITCH(mode, direction, opstate, table_start, table_end)`

DESCRIPTION:
The **HW_PSWITCH** command is used to control an output based on a position. It can either can either turn on the output when the start position is reached, and turn the output off when the next position is reached.

The output is a 24V output linked to the axis.

HW_PSWITCH outputs are assigned to the axes in a fixed way with one output per axis. See note 1.

The positions are defined as a sequence in the **TABLE** memory in range from table_start to table_end. On execution of the **HW_PSWITCH** command the positions are stored in a **FIFO** (first in - first out) queue.

The MC464 FlexAxis has 256 positions in the **FIFO**

The MC403 and MC405 have 512 positions in the **FIFO**

This command is applicable only to Flexible axes with ATYPEs that use incremental encoders, stepper or quadrature outputs.

When using a step direction output or encoder output **ATYPE** the positions do not take into account the 16 times multiplier. This means that you should enter your positions as ‘position * 16’.

The command can be used with either 1 or 5 parameters. Only 1 parameter is needed to disable the switch or clear **FIFO** queue. All five parameters are needed to enable the switch.

After loading the **FIFO** and going through the sequence of positions in it, if the same sequence has to be executed again, the **FIFO** must be cleared before executing another **HW_PSWITCH** command with the same parameters.

PARAMETERS:

mode:	0	Disable switch
	1	Toggles Digital Output at specified positions which are loaded into the HW **FIFO**.
	2	Clear **FIFO**
direction:	0	**MPOS** decreasing
	1	**MPOS** increasing.
opstate:	Output state to set in the first position in the **FIFO**; ON or OFF.	
table_start:	Starting **TABLE** address of the sequence.	
table_end:	Ending **TABLE** address of the sequence.	

NOTES:

NOTE 1:

The MC464 requires either the P874 or P879 Flexible Axis Module. The module has 4 digital outputs which are connected to the first 4 axes in the Flexaxis 8. In the Flexaxis 4, the first 2 axes have **HW_PSWITCH** circuits using the first 2 module outputs.

The MC405 has 5 **HW_PSWITCH** outputs. Axis 0 uses Output 8 and each axis in sequence uses the next output up to axis 4, which uses Output 12.

The MC403 has 3 **HW_PSWITCH** outputs. Axis 0 uses Output 8 and each axis in sequence uses the next output up to axis 2, which uses Output 10.

EXAMPLES:

EXAMPLE 1:

Load the table with 30 ON/OFF positions then run the command to load the **FIFO** with these positions. When the position stored in **TABLE**(21) is reached, the PSn output will be set ON and then alternatively OFF and ON on reaching the following positions in the sequence, until the position stored in **TABLE**(50) is reached.

```
TABLE(21,5,10,15,18,20,24,30,33,45,51,56,57,65,76,79,84,88,90,94)
TABLE(40,99,105,120,140,145,190,235,260,271,280,300)
HW_PSWITCH(1, 1, ON, 21, 50)
```

EXAMPLE 2:

Disable the switch if it was enabled previously. Does not clear the **FIFO** queue.

```
HW_PSWITCH(0)
```

EXAMPLE 3:

Clear the **FIFO** queue of a switch not on the **BASE** axis.

```
HW_PSWITCH(2) AXIS(8)
```

HW_TIMER

TYPE:

SLOT command

SYNTAX:

HW_TIMER(mode, cycleTime, <onTime, reps, > opState, opMode, opSel)

DESCRIPTION:

The **HW_TIMER** command turns ON/OFF a digital output or enable output of an axis for a specified length of ‘cycleTime’ (microseconds) in mode 1 or ‘onTime’ (microseconds) in mode 2 within the overall on/off time ‘cycleTime’.

The command can be used with either 1, 5 or 7 parameters. Only 1 parameter is needed to disable the timer. Five parameters are needed to enable the timer in mode 1, seven parameters for mode 2.

Note that the internal **FPGA** timer resolution is 10us so the requested time will be divided by 10 thus effectively truncating any remainder less than 10us e.g. 27 us will be interpreted as 20us. The user should also consider the rise/fall times of digital outputs, for highest performance then enable output selection should be used.

When using mode1 or 2 you must use an *ATYPE* with an enable output.

This command is only supported on controllers that have the correct *FPGA_PROGRAM*

PARAMETERS:

mode:	0	Disable timer
	1	Starts timer after which the selected output changes state.
	2	Starts timer after which the selected output changes state and then changes state again at the end of the overall cycle time and repeats for the given number of repetitions.
cycleTime:	Specifies in microseconds the timer cycle time to be used. For mode 1 this is effectively the ON time.	

<table>
<tr><td>onTime</td><td colspan="2">Mode 2 only, specifies in microseconds the timer ON time to be used within the overall ‘cycleTime’.</td></tr>
<tr><td>reps</td><td colspan="2">Mode 2 only, specified how many repetitions of the ‘cycleTime’ sequence are required.</td></tr>
<tr><td>opState:</td><td colspan="2">Initial state of selected output, ON or OFF.</td></tr>
<tr><td rowspan="3">opMode:</td><td>0</td><td>Indicates that a digital output is to be controlled.</td></tr>
<tr><td>1</td><td>Indicates that a Enable output output is to be controlled.</td></tr>
<tr><td>2</td><td>Indicates that a digital output and enable output output are to be controlled. These are only available in fixed pairs:
axis 0 + Digital Output 8
axis 1 + Digital Output 9
axis 2 + Digital Output 10
axis 3 + Digital Output 11
axis 4 + Digital Output 12</td></tr>
<tr><td>opSel:</td><td colspan="2">For opMode=0 this selects which digital output is to be controlled; valid range is 8..15.
For opMode=1 this selects which axis enable output (0..4) is to be controlled; valid range is 0..4.
For opMode=2 this selects which digital output and axis enable output is to be controlled; valid range is 0..4 which is interpreted as 8..12 for the corresponding digital output.</td></tr>
</table>

EXAMPLES:

EXAMPLE 1:

Request output 14 to be ON for 350us.

```
HW_TIMER(1,350,ON,0,14)
```

EXAMPLE 2:

Disable the timer after it was enabled previously.

```
HW_TIMER(0)
```

EXAMPLE 3:

Request enable output of axis 2 to be ON for 1.5s.

```
HW_TIMER(1,1500000,ON,1,2)
```

EXAMPLE 4:

Request digital output 9 and enable output of axis 1 to be OFF for 200ms.

```
HW_TIMER(1,200000,OFF,2,1) : WAIT UNTIL HW_TIMER_DONE
```

EXAMPLE 5:

Request a cycle time of 1s to be repeated 10 times with digital output 13 being ON for 3500us within each cycle.

```
HW_TIMER(2,1000000,3500,10,ON,0,13)
```

SEE ALSO:
HW_TIMER_DONE

HW_TIMER_DONE

TYPE:
SLOT command (Read Only)

SYNTAX:
HW_TIMER_DONE

DESCRIPTION:
Indicates whether or not a requested **HW_TIMER** is complete.

VALUE:

TRUE	The previous **HW_TIMER** request is complete
FALSE	The previous **HW_TIMER** request is NOT complete

EXAMPLE:
Request enable output of axis 4 to be ON for 500ms.

```
HW_TIMER(1,500000,ON,1,4) : WAIT UNTIL HW_TIMER_DONE
```

SEE ALSO:
HW_TIMER

I_GAIN

TYPE:
Axis Parameter

DESCRIPTION:
Used as part of the closed loop control, adding integral gain to a system reduces position error when at rest or moving steadily. It will produce or increase overshoot and may lead to oscillation.

For an integral gain Ki and a sum of position errors $\int_e$, the contribution to the output signal is:

$$O_i = K_i \times \int_e$$

VALUE:
The integral gain is a constant which is multiplied by the sum of following errors. Default value = 0

EXAMPLE:
Setting the gain values as part of a **STARTUP** program

```
P_GAIN=1
I_GAIN=0.01
D_GAIN=0
OV_GAIN=0
…
```

IDLE

TYPE:
Axis Parameter

DESCRIPTION:
Checks to see if an axis **MTYPE** is **IDLE**

VALUE:

TRUE	**MTYPE** is empty (**MTYPE**=0)
FALSE	**MTYPE** has a command loaded (**MTYPE**<>0)

EXAMPLES:

EXAMPLE 1:

Start a move and then suspend program execution until the move has finished. Note: This does not necessarily imply that the axis is stationary in a servo motor system.

```
MOVE(100)
WAIT IDLE
PRINT “Move Done”
```

EXAMPLE 2:

If the axis does not have any moves loaded then load a new sequence.

```
IF IDLE AXIS(1) THEN
  MOVE(100)
  MOVE(50)
  MOVE(-150)
ENDIF
```

IEEE_IN

TYPE:

Mathematical Function

SYNTAX:

IEEE_IN(byte0,byte1,byte2,byte3)

DESCRIPTION:

The **IEEE_IN** function returns the floating point number represented by 4 bytes which typically have been received over a communications link such as Modbus.

PARAMETERS:

byte0 - 3:	Any combination of 8 bit values that represents a valid **IEEE** floating point number.

Byte 0 is the high byte of the 32 bit floating point format.

EXAMPLE:

Take 4 bytes that have been sent over Modbus to **VR**s and recombine them into a floating point number.

```
VR(200) = IEEE_IN(VR(0),VR(1),VR(2),VR(3))
```

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 communication bus system. The function will typically be called 4 times to extract each byte in turn.

PARAMETERS:

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

Byte 0 is the high byte of the 32 bit floating point format.

EXAMPLE:
Extract the 4 bytes from **MPOS** and store then in local variables ready for transmission over a communications bus.

```
a = MPOS AXIS(2)
byte0 = IEEE_OUT(a, 0)
byte1 = IEEE_OUT(a, 1)
byte2 = IEEE_OUT(a, 2)
byte3 = IEEE_OUT(a, 3)
```

IF..THEN..ELSEIF..ELSE..ENDIF

TYPE:
Program Structure

SYNTAX:
```
IF condition THEN
  commands
ELSEIF expression THEN
  commands
```

```
ELSE
  commands
ENDIF
```

DESCRIPTION:

An IF program structure is used to execute a block of code after a valid expression. The structure will execute only one block of commands depending on the conditions. If multiple expressions are valid then the first will have its commands executed. If no expressions are valid and an **ELSE** is present the commands under the **ELSE** will be executed.

PARAMETERS:

condition:	Any valid logical TrioBASIC expression
commands:	TrioBASIC statements that you wish to execute

EXAMPLES:

EXAMPLE 1:

Check for the batch to be complete, if it is then tell the user and process the batch

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

EXAMPLE 2:

Use an IF statement to light a warning lamp when machine is running

```
IF WDOG=ON THEN
  OP(warning, ON)
ELSE
  OP(warning, OFF)
ENDIF
```

EXAMPLE 3:

Use an IF structure to report the operating state of a machine.

```
IF operating_state=0 THEN
  PRINT#5, "Machine Running"
ELSEIF operating_state=1 THEN
  PRINT#5, "Machine Idle"
ELSEIF operating_state=2 THEN
  PRINT#5, "Machine Jammed"
ELSE
  PRINT#5, "Machine in unknown state"
ENDIF
```

IN

TYPE:

System Function.

SYNTAX:

value = IN[(input_no[,final_input])]

DESCRIPTION:

IN is used to read the state of the inputs.

If called with no parameters, IN returns the binary sum of the first 32 inputs. If called with one parameter it returns the state (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 32 apart.

IN is equivalent to **IN**(0,31)

PARAMETERS:

value:	The state of the selected input or range of inputs
none:	Returns the binary sum of the first 32 inputs
input_no:	input to return the value of/start of input group
final_input:	last input of group

EXAMPLES:

EXAMPLE 1:

In this example a single input is tested:

```
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 binary coded decimal.

The move command is constructed in the following order:

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** to this position

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

EXAMPLE 3:

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
```

INCLUDE

TYPE:

System Command.

SYNTAX:

`INCLUDE "filename"`

DESCRIPTION:

The **`INCLUDE`** command resolves all local variable definitions in the included file at compile time and allows all the local variables to be declared "globally".

Whenever an included program is modified, all programs that depend on it are re-compiled as well, avoiding inconsistencies.

Nested **`INCLUDE`**s are not allowed.

The **`INCLUDE`** command must be the first BASIC statement in the program.

Only variable definitions and conditional logic are allowed in the include file. It cannot be used as a general subroutine with any other BASIC commands in it.

PARAMETERS:

filename:	The name of the program to be included

EXAMPLE:

Initialise all local variables with an include program.

PROGRAM "T1":

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

PROGRAM "GLOBAL_DEFS":

```
drive_axis=4
linked_axis=1
```

INDEVICE

TYPE:

Process Parameter

DESCRIPTION:

This parameter specifies the default active input device. Specifying an **INDEVICE** for a process allows the channel number for a program to set for all subsequent **GET**, **KEY**, **INPUT** and **LINPUT** statements.

This command is process specific so other processes will use the default channel.

This command is available for backward compatibility, it is currently recommended to use #channel, instead.

VALUE:

The channel number to use for any inputs

For a full list of communication channels see #

EXAMPLE:

Set up a program to use channel 5 by default for any **GET** commands

```
INDEVICE=5
' Get character on channel 5:
IF KEY THEN
  GET k
ENDIF
```

SEE ALSO:

#, GET, INPUT, KEY, LINPUT

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.

INITIALISE may reset a parameter relating to a digital drive communication or encoder causing you to lose the connection.

EXAMPLE:

When developing you wish to clear all parameters back to default using the command line.

```
>>INITIALISE
>>
```

INPUT

TYPE:

System Command.

SYNTAX:

INPUT [#channel,] variable [, variable…]

DESCRIPTION:

Waits for an **ASCII** 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 values may be requested on one line, the values are separated by commas, or by carriage returns <CR>.

Poll **KEY** to check to if a character has been received before performing an **INPUT**.

PARAMETERS:

#channel:	See # for the full channel list (default 0 if omitted)
variable:	The variable to store the received character, this may be local variable, **VR** or **TABLE**

Performing an **INPUT** or **INPUT**#0 will suspend the command line until a character is sent on that channel.

EXAMPLES:

EXAMPLE 1:

Receive a single value and store it in a local variable num

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

On terminal:
123 <CR>
BATCH COUNT=123
```

EXAMPLE 2:

Get the length and width variables using one **INPUT**.

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

This will display on terminal:

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

SEE ALSO:

#, KEY

INPUTS0 / INPUTS1

TYPE:

System Parameter

DESCRIPTION:

The INPUTS0/ INPUTS1 parameters holds the state of the Input channels as a system parameter.

Reading the inputs using these system parameters is not normally required. The **IN**(x,y) command should be used instead. They are made available in this format to make the input channels accessible to the **SCOPE** command which can only store parameters.

VALUE:

INPUTS0	The binary sum of IN(0)..IN(15)
INPUTS1	The binary sum of IN(16)..IN(31)

SEE ALSO:

IN

INSTR

TYPE:

String Function

SYNTAX:

```
INSTR(<offset index,>string, search string<,wild card char>)
```

DESCRIPTION:

Searches the input string looking for the search string and returns the (zero based) index of the first occurrence of the string or -1 if the string is not found.

PARAMETERS:

Offset index:	An integer offset into the string being searched
string:	String to be searched
Search string:	Search string to look for
Wild card char:	A single wild card character to use within the search string expressed as a single character string or as a numerical **ASCII** value

EXAMPLES:

EXAMPLE:

Pre-define a variable of type string and search it for various sub-strings:

```
DIM str1 AS STRING(32)
str1 = “TRIO MOTION TECHNOLOGY”
PRINT INSTR(str1, “MOTION”) ‘value = 5
PRINT INSTR(6, str1, “MOTION”) ‘value = -1
```

```
PRINT INSTR("Value = 123.45E10", "###.##E##", "#") 'Value = 8
PRINT INSTR("this is my string", "is *y", 42) 'Value = 5
PRINT INSTR(3, str1, "IO") 'Value = 8
```

SEE ALSO:
CHR, STR, VAL, LEFT, RIGHT, MID, LEN, LCASE, UCASE

INT

TYPE:
Mathematical Function

SYNTAX:
```
value = INT(expression)
```

DESCRIPTION:
The INT function returns the integer part of a number.

To round a positive number to the nearest integer value take the **INT** function of the (number + 0.5)

PARAMETERS:

expression:	Any valid TrioBASIC expression.
value:	The integer part of the expression

EXAMPLES:

EXAMPLE 1:
Print the integer part of a number on the command line
```
>>PRINT INT(1.79)
1.0000
>>
```

EXAMPLE 2:
Round a value to the nearest integer.
```
IF value>0 THEN
  rounded = INT(value + 0.5)
ELSE
  rounded = INT(value - 0.5)
ENDIF
```

INTEGER_READ

TYPE:
Mathematical Command

SYNTAX:
`INTEGER_READ(source, least_significant, most_significant)`

DESCRIPTION:
INTEGER_READ performs a low level access to the 64 bit register splitting it into two 32 bit segments.

This can be used to read the position from high resolution encoders

PARAMETERS:

source:	2 bit value that will be read, can be **VR**, **TABLE**, or system variable.
least_significant:	The variable to store the least significant (rightmost) 32 bits, this may be local variable, **VR** or **TABLE**
most_significant:	The variable to store the most significant (leftmost) 32 bits, this may be local variable, **VR** or **TABLE**

INTEGER_WRITE

TYPE:
Mathematical Command

SYNTAX:
`INTEGER_WRITE(destination, least_significant, most_significant)`

DESCRIPTION:
INTEGER_WRITE performs a low level write to a 64 bit register by combining two 32 bit segments.

PARAMETERS:

destination:	64 bit value that will be written, can be **VR**, **TABLE**, or system variable.
least_significant:	Least significant (rightmost) 16 bits, can be any valid TrioBASIC expression.
most_significant:	Most significant (leftmost) 16 bits, can be any valid TrioBASIC expression.

INTERP_FACTOR

TYPE:

Axis parameter

DESCRIPTION:

This parameter excludes the axis from the interpolated motion calculations so that it will become a following axis. This means that you can create an interpolated x,y move with z completing its movement over the same time period. The interpolated speed is calculated using any axes that have **INTERP_FACTOR** enabled. This means that at least one axis must be enabled and have a distance in the motion command otherwise the calculated speed will be zero and the command will complete immediately with no movement.

INTERP_FACTOR only operates with **MOVE**, **MOVEABS** and **MHELICAL** (on the 3rd axis) and their SP versions. All other motion commands require interpolated axes and so ignore this parameter.

EXAMPLE:

It is required to move a ‘z’ axis interpolated with x and y however we want the interpolated speed to only be active on the ‘x,y’ move. We disable the z axis from the interpolation group using **INTERP_FACTOR**. Remember when the movement is complete you must enable **INTERP_FACTOR** again.

```
BASE(2)
INTERP_FACTOR=0

‘Perform movement
BASE(0,1,2)
MOVEABS(x_offset, y_offset, z_offset)

WAIT IDLE
INTERP_FACTOR AXIS(2) = 1
```

INVERT_IN

TYPE:

System Function

SYNTAX:

`INVERT_IN(input, state)`

DESCRIPTION:

The **INVERT_IN** command allows the input channels to be individually inverted in software.

This is important as these input channels can be assigned to activate functions such as feedhold.

PARAMETERS:

input:	The input to invert	
state:	ON	the input is inverted in software
	OFF	the input is not inverted

EXAMPLE:

Invert input 7 so that when the input is low the **FWD_JOG** is off

```
INVERT_IN(7,ON)
FWD_JOG=7
```

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 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.

INVERT_STEP should be set with **WDOG=OFF**.

If the setting is incorrect, a stepper motor may lose position by one step when changing direction.

VALUE:

ON	**RISING** edge of the step signal the active edge
OFF	**FALLING** edge of the step signal the active edge (default)

EXAMPLE:

Set **INVERT_STEP** for axis 2 as part of a startup routine.

```
BASE(2)
INVERT_STEP = ON
```

IO_STATUS

TYPE:
System Function

SYNTAX:
value = IO_STATUS(slot, address, vr_index [, status_index])

DESCRIPTION:
This command reads the status of a remote IO device on EtherCAT.
Status bit representation depends on the device implementation.

PARAMETERS:

value:	-1	Success
	0	Failure
slot:	The slot which the Ethercat IO module is connected	
address:	Network address of the IO device from which the status is read.	
vr_index:	-1	Print to the terminal
	>=0	Index of the **VR** where the status is stored
status_index	Index of the status being read (default 0).	

An Omron “block-type” device has one general status value for all IO so only status_index 0 is valid. A Beckhoff E-bus device has one status value per channel/point. Therefore for each channel the status can be read by using the status index. Here the valid range of status_index is 0..(number of channels -1).

IO_STATUSMASK

TYPE:
System Function

SYNTAX:
value = IO_STATUSMASK(slot, address, read_write, vr_index or mask value [, status index])

DESCRIPTION:
This command reads or writes the status mask of a remote Ethercat IO device. With a status mask system,

errors triggered by an **IO_STATUS** of a device can be masked out thus preventing a **SYSTEM_ERROR**. If the same bit is set in **IO_STATUS** and **IO_STATUSMASK** on the same device, a system error is triggered.

Status bit representation depends on the device implementation.

PARAMETERS:

value:	-1	Success
	0	Failure
slot:	The slot which the Ethercat IO module is connected	
address:	Network address of the IO device from which the status is read.	
Function:	0	Read status mask
	1	Write status mask

An Omron “block-type” device has one general status value for all **IO** so only status_index 0 is valid. A Beckhoff E-bus device has one status value per channel/point. Therefore for each channel the status can be read by using the status index. Here the valid range of status_index is 0..(number of channels -1).

IOMAP

TYPE:

System Command (command line only)

SYNTAX:

IOMAP

DESCRIPTION:

Lists the current Digital IO map.

EXAMPLE:

```
>> IOMAP
Digital Input map :
     0-   7 : Built-in Inputs
     8-  15 : Built-in Bi-Directional IO
    16-  31 : CAN P318 @ Address 0 (fw=v1.3.0)
    32-1023 : Virtual

Digital Output map :
     0-   7 : Virtual
```

```
 8-  15 : Built-in Bi-Directional IO
16-  31 : CAN P327 @ Address 0 (fw=v1.3.0)
32-1023 : Virtual
```

IP_ADDRESS

TYPE:

System Parameter (**MC_CONFIG** / **FLASH**)

DESCRIPTION:

IP_ADDRESS is used to set the Ethernet IPv4 address of the main Ethernet port of the *Motion Coordinator*. This parameter uses the standard dot (.) notation to define the 4 separate octets of the IP address.

The value is held in flash EPROM and can be set in the **MC_CONFIG** script.

VALUE:

Network IP address in dot (.) format.

EXAMPLES:

EXAMPLE 1:

```
IP_ADDRESS = 192.168.0.250
```

EXAMPLE 2:

Set IP address in the **MC_CONFIG** file

```
‘ MC_CONFIG script file
IP_ADDRESS=192.168.2.100
```

IP_GATEWAY

TYPE:

System Parameter (**MC_CONFIG** / **FLASH**)

DESCRIPTION:

IP_GATEWAY is used to set the Ethernet network gateway address of the main Ethernet port of the *Motion Coordinator*. The Gateway is the IPv4 address of the internet access router on the factory network. It is only required if the *Motion Coordinator* is to be accessed via the internet. This parameter uses the standard dot (.) notation to define the 4 separate octets of the IP gateway address.

The value is held in flash EPROM and can be set in the **MC_CONFIG** script.

VALUE:
Network gateway address in dot (.) format.

EXAMPLES:

EXAMPLE 1:
```
IP_GATEWAY = 192.168.0.254
```

EXAMPLE 2:
Set IP gateway in the **MC_CONFIG** file
```
' MC_CONFIG script file
IP_GATEWAY=192.168.0.254
```

IP_MAC

TYPE:
System Parameter (**FLASH** / Read-only)

DESCRIPTION:
IP_MAC returns the configured MAC address of the main Ethernet port of the *Motion Coordinator*. The MAC address is set once at manufacture and is unique to that controller.

The value is held in flash EPROM and is normally read-only. If write access is available on older versions of firmware, do not change the MAC address under any circumstances without first consulting Trio.

VALUE:
Ethernet MAC address as a single 48 bit number.

EXAMPLES:

EXAMPLE 1:
```
>>PRINT IP_MAC
27648852217.0000
>>
```

EXAMPLE 2:
Get the MAC address in hexadecimal format
```
>>?hex(ip_mac)
6700000F9
>>
```
Converted to the 6 Octets format this is: 00 06 70 00 00 F9

IP_MEMORY_CONFIG

TYPE:

System Parameter (**MC_CONFIG**)

DESCRIPTION:

The MC464 Ethernet port has memory allocated to buffer the incoming and outgoing data telegrams. Each buffer page uses 1600 bytes of memory. If some ports are turned off using **IP_PROTOCOL_CONFIG**, then **IP_MEMORY_CONFIG** may be used to re-allocate the unused memory and give a larger buffer size to the incoming and outgoing data.

By default there are 2 x 1600 bytes allocated to Tx and 2 x 1600 allocated to Rx. The value of **IP_MEMORY_CONFIG** is $22. (or 2 + 32 in decimal) In most networks this buffer size is enough to handle all the network traffic.

VALUE:

The **IP_MEMORY_CONFIG** is a byte which is split into 2 nibbles.

Bits	Description	Value
0 - 3	Size of Rx buffer; number of 1600 byte pages.	$01 to $09
4 - 7	Size of Tx buffer; number of 1600 byte pages.	$10 to $90

Do not set either nibble to less than 1 otherwise there will be no memory allocated and *Motion* Perfect will not be usable.

EXAMPLE:

Allocate more buffer space for incoming Rx Ethernet traffic to cope with frequent broadcast telegrams on a busy network.

```
' Disable Ethernet IP and text file loader ports
IP_PROTOCOL_CONFIG = $37
' Allocate the freed memory space to Rx net-buffer
IP_MEMORY_CONFIG = $29
```

IP_NETMASK

TYPE:

System Parameter (**MC_CONFIG** / **FLASH**)

DESCRIPTION:

IP_NETMASK is used to set the Ethernet IPv4 subnet mask of the main Ethernet port of the *Motion Coordinator*. This parameter uses the standard dot (.) notation to define the 4 separate octets of the IP subnet mask.

The value is held in flash EPROM and can be set in the **MC_CONFIG** script.

VALUE:

Network subnet mask in dot (.) format.

EXAMPLES:

EXAMPLE 1:

```
IP_NETMASK = 255.255.255.0
```

EXAMPLE 2:

Set IP subnet mask in the **MC_CONFIG** file

```
' MC_CONFIG script file
IP_NETMASK=255.255.255.0
```

IP_PROTOCOL_CONFIG

TYPE:

System Parameter (**MC_CONFIG**)

DESCRIPTION:

The MC464 is limited to 7 communication ports on Ethernet, **IP_PROTOCOL_CONFIG** allows the user to select which ports they would like to use.

By default all ports except the transparent protocol text file loader port are enabled. It is recommended to use the MC4xx protocol which is enabled by default.

VALUE:

Up to 7 bits can be selected, the default value is 575 ($23F).

Bit	Description	Value
0	*Motion* Perfect (Telnet)	1
1	PCMotion	2
2	Modbus	4

Bit	Description	Value
3	EthernetIP	8
4	IEC61131-3 programming	16
5	Uniplay	32
6	Transparent protocol text file loader	64
7	Reserved bit	128
8	Reserved bit	256
9	MC4xx protocol text file loader	512

Do not disable bit 0 otherwise the command line and *Motion* Perfect will not be usable.

EXAMPLE:
Enable the standard ports using bits 0-5 and the transparent protocol text file loader ports.

```
IP_PROTOCOL_CONFIG = 1+2+4+8+16+32+64
' or
IP_PROTOCOL_CONFIG = $7F
```

IP_PROTOCOL_CTRL

TYPE:
System Parameter (**MC_CONFIG**)

DESCRIPTION:
This parameter mirrors the **IP_PROTOCOL_CONFIG** bit pattern to allow the user to disable the operation of one or more of the MC464 communication ports on Ethernet. If a bit is at 0, the port is enabled. If the bit is a 1, then the port is disabled and will not respond when a client tries to open it.

By default all ports are enabled.

VALUE:
Up to 2 bits can be selected, the default value is 0.

Bit	Description	Value
0	*Motion* Perfect (Telnet)	n/a
1	PCMotion	n/a
2	Modbus	4

Bit	Description	Value
3	EthernetIP	8
4	IEC61131-3 programming	n/a
5	Uniplay	n/a
6	Transparent protocol text file loader	n/a
7	Reserved bit	n/a
8	Reserved bit	n/a
9	MC4xx protocol text file loader	n/a

It is not possible to disable any port marked as n/a.

EXAMPLE 1:

Disable the Modbus TCP port until it has been set up for 32 bit data size in the **BASIC** startup program.

a) In the MC_**CONFIG** set:

```
IP_PROTOCOL_CTRL = 4
```

b) In the Startup BASIC program set:

```
ETHERNET(1, -1, 14, 0, 2, 1) ‘ 32 bit integer support
ETHERNET(1, -1, 14, 0, 1, 4) ‘ data to/from TABLE memory
ETHERNET(1, -1, 14, 0, 6, 1) ‘ Use “Address Halving”

IP_PROTOCOL_CTRL = 0 ‘ start the Modbus TCP protocol
```

EXAMPLE 2:

Disable the Ethernet IP port until the data end-points have been set up in the **BASIC** startup program.

a) In the MC_**CONFIG** set:

```
IP_PROTOCOL_CTRL = 8
```

b) In the Startup BASIC program set:

```
‘--Config *PLC INPUT* Instance (100), data to PLC from Trio.
ETHERNET(1, -1, 14, 1, 0, 200) ‘200 = set VR starting address
ETHERNET(1, -1, 14, 1, 1, 3) ‘3 = use VR for data location
ETHERNET(1, -1, 14, 1, 2, 1) ‘1 = use 32 bit integer data
ETHERNET(1, -1, 14, 1, 3, 120) ‘120 = number of data values

‘--Config *PLC OUTPUT* Instance (101), data from PLC to Trio.
ETHERNET(1, -1, 14, 2, 0, 400) ‘400 = set VR starting address
```

```
ETHERNET(1, -1, 14, 2, 1, 3) '3 = use VR for data location
ETHERNET(1, -1, 14, 2, 2, 1) '1 = use 32 bit integer data
ETHERNET(1, -1, 14, 2, 3, 120) '120 = number of data values

'-------------------------------------------------------------

IP_PROTOCOL_CTRL = 0 'enable Ethernet IP
```

IP_TCP_TIMEOUT

TYPE:
System Parameter (**MC_CONFIG**, MC464 only)

DESCRIPTION:
IP_TCP_TIMEOUT defines the time period (in msec) for which the TCP connections (EtherNet/IP, ModbusTCP, HMI, Token and Telnet) will stay open without any activity. When this period is exceeded, the TCP connection will be closed by the controller. The default is 3600 seconds.) The parameter must be in the **MC_CONFIG** to be effective.

VALUE:

Size	Bits	Value (hexadecimal)	Function
Long word	Bit 0..11	$0000000000000ttt	Telnet TCP timeout
	Bits 12..23	$0000000000ttt000	Token system timeout
	Bits 24..35	$0000000ttt000000	Modbus TCP timeout
	Bits 36..47	$0000ttt000000000	Ethernet IP timeout
	Bits 48..59	$0ttt000000000000	Uniplay HMI channel timeout
	Bits 60..63	$x000000000000000	Not used

Setting this value away from the default may make the connection to *Motion* Perfect unstable.

Each 12 bits of this value sets the timeout period (in seconds) for that part of the Ethernet. If it is left at 0, then it becomes the default of 3600 seconds.

There is also a built-in timeout in the Ethernet stack. The default is approximately 8 seconds, so when you set the value in **IP_TCP_TIMEOUT** to 2 seconds, the total is 10.

EXAMPLE 1:

Force the Ethernet processor to close the Modbus TCP socket after 20 seconds when there is no activity from the master. This enables the master to re-open the connection and continue after a break in communications.

```
' Modbus socket will close after 20 seconds (12 + 8)
IP_TCP_TIMEOUT = $00C000000
```

EXAMPLE 2:

Set the Ethernet processor to close the Ethernet IP TCP socket after 12 seconds when there is no activity from the master. This enables the master to re-open the connection and continue after a break in communications.

```
' Modbus socket will close after 12 seconds (4 + 8)
IP_TCP_TIMEOUT = $004000000000
```

IP_TCP_TX_THRESHOLD

TYPE:

System Parameter (**MC_CONFIG**)

DESCRIPTION:

IP_TCP_TX_THRESHOLD defines the number of bytes in the TCP socket transmit buffer which will trigger a telegram transmit. The default is 32. This value applies to all the TCP protocols.

VALUE:

Please consult Trio before changing this value.

Size	Description	Value	Default
word	Number of bytes in TCP socket transmit buffer which triggers a transmission.	1 to 1023	32

Setting this value away from the default may make the connection to *Motion* Perfect unstable.

EXAMPLE:

Force the Ethernet processor to transmit TCP packets immediately when the data size is small, so as not to wait for the timeout before sending.

```
IP_TCP_TX_THRESHOLD = 16
```

IP_TCP_TX_TIMEOUT

TYPE:
System Parameter (**MC_CONFIG**)

DESCRIPTION:
IP_TCP_TX_TIMEOUT defines the time period (in msec) at which a TCP telegram will be transmitted after receiving the first byte if the number of bytes threshold is not reached. The default is 20msec. This value applies to all the TCP protocols.

VALUE:

Please consult Trio before changing this value.

Size	Description	Value	Default
Long word	Time after which telegram will be transmitted if the data size threshold is not reached. (milliseconds)	1 to 2^32-1	20

Setting this value away from the default may make the connection to *Motion* Perfect unstable.

EXAMPLE:
Force the Ethernet processor to transmit TCP packets only after 1 second when the data size threshold is not reached.

```
IP_TCP_TX_TIMEOUT = 1000
```

JOGSPEED

TYPE:
Axis Parameter

DESCRIPTION:
Sets the jog speed in user units for an axis to run at when performing a jog.

You can set a faster jog speed using **SPEED** and the **FAST_JOG** input

VALUE:
The speed in user **UNITS**/second which an axis will use when being jogged

EXAMPLE:

Configure an input to be the jog input at 20mm/sec on axis 12

```
BASE(12)
SPEED=3000
FWD_JOG = 12
JOGSPEED = 20
```

SEE ALSO:

FAST_JOG, FWD_JOG, REV_JOG

KEY

TYPE:

System Function.

SYNTAX:

`value = KEY [#channel]`

DESCRIPTION:

Key is used to check if there are characters in a channel buffer. 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**.

PARAMETERS:

#channel:	See # for the full channel list (default 0 if omitted)
value:	A negative value representing the number of characters in the channel buffer

EXAMPLE:

Call a subroutine if a character has been received on channel 1

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

SEE ALSO:

GET

LAST_AXIS

TYPE:
System Parameter

DESCRIPTION:
The *Motion coordinator* keeps a list of axes that are currently in use. **LAST_AXIS** is used to read the number of the highest axis in the list.

LAST_AXIS is set automatically by the system software when an axis is written to; this can include setting **BASE** for the axis.

Axes higher than **LAST_AXIS** are not processed. Not all axis lower than **LAST_AXIS** are processed.

VALUE:
The highest axis in the axis list that is processed.

EXAMPLE:
Check **LAST_AXIS** to ensure that the digital network has configured enough drives.

```
IF LAST_AXIS <> 26 THEN
  PRINT#user, “Digital Drives not initialised”
ENDIF
```

LCASE

TYPE:
String Function

SYNTAX:
LCASE(string)

DESCRIPTION:
Returns a new string with the input string converted to all lower case.

PARAMETERS:

string:	String to be used

EXAMPLES:

EXAMPLE 1:

Pre-define a variable of type string and later print it in all lower case characters:

```
DIM str1 AS STRING(32)
str1 = “TRIO MOTION TECHNOLOGY”
PRINT LCASE(str1)
```

SEE ALSO:

CHR, STR, VAL, LEFT, RIGHT, MID, LEN, UCASE, INSTR

LCDSTR

TYPE:

String Function

SYNTAX:

LCDSTR = string

DESCRIPTION:

Allows the currently displayed character string on display to be read from or written to when under user control. This will only be allowed when the display is in normal display mode, for example if the user removes and replaces the EtherNET cable then the displaying of IP address data will take priority before returning to the previous display string again.

This function is available on the MC405 only.

VALUE:

The string is predefined with a length of 3 and reflects the currently displayed 7-segment characters.

EXAMPLES:

EXAMPLE 1:

Take user control of 7-segement characters and display integer value of **VR**(100).

```
DISPLAY.16 = 1 ‘Enable user control of 7-segment chars
vr(100) = -88
LCDSTR = STR(VR(100),0,3)
```

SEE ALSO:

DISPLAY

LEFT

TYPE:
String Function

SYNTAX:
`LEFT(string, length)`

DESCRIPTION:
Returns the left most section of the specified string using the length specified.

PARAMETERS:

string:	String to be used
length:	Length of string to be returned

EXAMPLES:

EXAMPLE 1:
Pre-define a variable of type string and later print its left most 4 characters:

```
DIM str1 AS STRING(32)
str1 = “TRIO MOTION TECHNOLOGY”
PRINT LEFT(str1, 4)
```

SEE ALSO:
CHR, STR, VAL, RIGHT, MID, LEN, LCASE, UCASE, INSTR

LEN

TYPE:
String Function

SYNTAX:
`LEN(string)`

DESCRIPTION:
Returns length of the specified string

PARAMETERS:

string:	String to be measured.

EXAMPLES:

EXAMPLE 1:
Pre-define a variable of type string and later determine its length:

```
DIM str1 AS STRING(20)
Str1="MyString"
x=LEN(str1) ' x will be 8
```

SEE ALSO:
CHR, STR, VAL, LEFT, RIGHT, MID, LCASE, UCASE, INSTR

< Less Than

TYPE:
Comparison Operator

SYNTAX:
```
<expression1> < <expression2>
```

DESCRIPTION:
Returns **TRUE** if expression1 is less than expression2, otherwise returns **FALSE**.

PARAMETERS:

Expression1:	Any valid TrioBASIC expression
Expression2:	Any valid TrioBASIC expression

EXAMPLE:
Check that the value from analogue input 1 is less than 10, if it is then execute the sub routine 'rollup'.

```
IF AIN(1)<10 THEN GOSUB rollup
```

<= Less Than or Equal

TYPE:
Comparison Operator

SYNTAX:
```
<expression1> <= <expression2>
```

DESCRIPTION:
Returns **TRUE** if expression1 is less than or equal to expression2, otherwise returns **FALSE**.

PARAMETERS:

Expression1:	Any valid TrioBASIC expression
Expression2:	Any valid TrioBASIC expression

EXAMPLE:
1 is not less than or equal to 0 and therefore variable maybe holds the value 0 (**FALSE**)
```
maybe=1<=0
```

LIMIT_BUFFERED

TYPE:
System Parameter

DESCRIPTION:
This sets the maximum number of move buffers available in the controller.

You can increase the machine speed when using *MERGE* or *CORNER_MODE* by increasing the number of buffers.

VALUE:

1..64	The number of move buffers (default = 1)

EXAMPLE:
Configure the *Motion Coordinator* to have 10 move buffers so a large sequence of small moves can be merged together.
```
LIMIT_BUFFERED = 10
```

_ (Line Continue)

TYPE:
Special Character

SYNTAX:
```
ExpressionStart _
ExpressionEnd
```

DESCRIPTION:
The line extension allows the user to split a long expression or command over more than one lines in the TrioBASIC program.

The split must be at the end of a parameter or keyword.

PARAMETERS:

ExpressionStart:	The start of the command or expression.
ExpressionEnd:	The end of the command or expression.

EXAMPLE:
Split the **SERVO_READ** command over 2 lines so you can use all 8 parameters.
```
SERVO_READ(123, MPOS AXIS(0), MPOS AXIS(1), MPOS AXIS(2), _
MPOS AXIS(3), MPOS AXIS(4), MPOS AXIS(5), MPOS AXIS(6))
```

LINK_AXIS

TYPE:
Axis Parameter (Read Only)

ALTERNATIVE FORMAT:
LINKAX

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*

VALUE:

-1	Axis is not linked
Number	Axis number the **BASE** axis is linked to

EXAMPLE

CONNECT an axis , then check that it is linked.

```
>>BASE(0)
>>CONNECT(12,4)
>>PRINT LINK_AXIS
4.0000
>>
```

LINPUT

TYPE:

System Command

SYNTAX:

LINPUT [#channel,] 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.

You can print the string from the VRs using **VRSTRING**

PARAMETERS:

#channel:	See # for the full channel list (default 0 if omitted)
variable:	The **VR** variable to store the received character

EXAMPLE:

Use **LINPUT** to receive a string of characters on channel 5 and place then into a series of **VR**s starting at **VR**(0)

```
LINPUT#5, VR(0)
```

Now entering: **START**<CR> on channel 5 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
```

SEE ALSO:
#, CHANNEL_READ, VRSTRING

LIST

TYPE:
System Command (command line only)

SYNTAX:
`LIST ["program"]`

DESCRIPTION:
Prints the current **SELECTed** program or a specified program to the current output channel.

Usually you will view a program by using *Motion* Perfect.

PARAMETERS:

none:	Prints the selected program
program:	The name of the program to print

LIST_GLOBAL

TYPE:
System Command (command line only)

SYNTAX:
`LIST_GLOBAL`

DESCRIPTION:
Prints all the **GLOBAL** and **CONSTANT**s to the current output channel

EXAMPLE:

Check all global data in an application where the following **GLOBAL** and **CONSTANT** have been set.

```
CONSTANT “cutter”, 23
GLOBAL “conveyor”,5

>>LIST_GLOBAL
Global                          VR
----------------                ----
conveyor                        5
Constant                        Value
----------------                -------
cutter                          23.00000
>>
```

LN

TYPE:

Mathematical Function

SYNTAX:

```
value = LN(expression)
```

DESCRIPTION:

Returns the natural logarithm of the expression.

PARAMETER:

value:	The natural logarithm f the expression
expression:	Any valid TrioBASIC expression.

EXAMPLE:

Storing the natural logarithm of a value in **VR**(0)

```
VR(0) = LN(a*b)
```

LOAD_PROJECT

TYPE:

System Command

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

If you wish to load projects outside of *Motion* Perfect use the Autoloader ActiveX

LOADED

TYPE:
Axis Parameter

DESCRIPTION:
Checks if all the movements have been loaded into the **MTYPE** buffer so will return a **TRUE** value when there are no buffered movements.

Although it is possible to use **LOADED** as part of any expression it is typically used with a *WAIT*.

VALUE:

TRUE	when there are no buffered moves
FALSE	when there are buffered moves.

EXAMPLE:
Continue to load a sequence of moves when the **NTYPE** buffer is free

```
WHILE machine_on =TRUE
  WAIT UNTIL LOADED or machine_off=FALSE
  IF machine_on=TRUE THEN
    MOVE(TABLE(position)
    position=position+1
  ENDIF
WEND
```

SEE ALSO:
MOVES_BUFFERED, WAIT

LOADSYSTEM

TYPE:
System Command

DESCRIPTION:
Used by *Motion* Perfect to load Firmware to the controller

If you wish to load firmware without *Motion* Perfect you can use the SD card (**FILE** command)

SEE ALSO:
FILE

LOCK

TYPE:
System Command (command line only)

SYNTAX:
LOCK(code)

DESCRIPTION:
The **LOCK** copmmand 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. The **CONTROL** value has 1000 added to it when the controller is **LOCK**ed.

You should use *Motion* Perfect to **LOCK** and **UNLOCK** your controller.

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.

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. If you want better security consider encrypting your project.

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

PARAMETERS:

code:	Any 7 digit integer number

SEE ALSO:
UNLOCK

LOOKUP

TYPE:
Process Command

SYNTAX:
LOOKUP(format,entry) <PROC(process#)>

DESCRIPTION:
The **LOOKUP** command is used by *Motion* Perfect to access the local variables on an executing process.

You should use the variable watch window in *Motion* Perfect to access the variables on an executing process.

PARAMETERS:

format:	0	Prints (in binary) floating point value from an expression
	1	Prints (in binary) integer value from an expression
	2	Prints (in binary) local variable from a process
	3	Returns to **BASIC** local variable from a process
	4	Write
entry:	Either an expression string (format=0 or 1) or the offset number of the local variable into the processes local variable list.	

MARK

M

TYPE:
Axis Parameter (Read Only)

DESCRIPTION:
This parameter can be polled to determine if the registration event has occurred.

MARK is reset when **REGIST** is executed

VALUE:

FALSE	The registration event has not occurred
TRUE	The registration event has occurred (default)
< -1	Quantity of registration events have been logged to the **TABLE**

When **TRUE** the **REG_POS** is valid.

EXAMPLE:
Apply an offset to the position of the axis depending on the registration position.

```
loop:
  WAIT UNTIL IN(punch_clr)=ON
  MOVE(index_length)
  REGIST(20, 0, 0, 0, 0) 'rising edge of R
  WAIT UNTIL MARK
  MOVEMODIFY(REG_POS + offset)
  WAIT IDLE
GOTO loop
```

SEE ALSO:
REGIST, REG_POS

MARKB

TYPE:
Axis Parameter (Read Only)

DESCRIPTION:
This parameter can be polled to determine if the registration event has occurred on the second registration

channel.

MARKB is reset when **REGIST** is executed

VALUE:

FALSE	The registration event has not occurred
TRUE	The registration event has occurred (default)
< -1	Quantity of registration events have been logged to the **TABLE**

When **TRUE** the **REG_POSB** is valid.

SEE ALSO

REGIST, REG_POSB

MERGE

TYPE:

Axis Parameter

DESCRIPTION:

Velocity profiled moves can be MERGEd together so that the speed will not ramp down to zero between the current move and the buffered move.

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:

- The next move is loaded into the buffer
- The axis group does not change on multi-axis moves

Velocity profiled moves (**MOVE**, **MOVEABS**, **MOVECIRC**, **MHELICAL**, **REVERSE**, **FORWARD**) cannot be merged with linked moves (**CONNECT**, **MOVELINK**, **CAMBOX**)

When merging multi-axis moves only the base axis **MERGE** flag needs to be set.

If you are merging short moves you may need to increase the number of buffered moves by increasing **LIMIT_BUFFERED**

VALUE:

ON	motion commands are merged
OFF	motion commands decelerate to zero speed

EXAMPLE:

Turn on **MERGE** before a sequence of moves, then disable at the end.

```
BASE(0,1) ‘set base array
MERGE=ON ‘set MERGE state
MOVEABS(0,50) ‘run a sequence of moves
MOVE(0,100)
MOVECIRC(50,50,50,0,1)
MOVE(100,0)
MOVECIRC(50,-50,0,-50,1)
MOVE(0,-100)
MOVECIRC(-50,-50,-50,0,1)
MOVE(-100,0)
MOVECIRC(-50,50,0,50,1)
WAIT IDLE
MERGE=OFF
```

MHELICAL

TYPE:

Axis Command.

SYNTAX:

`MHELICAL(end1, end2, centre1, centre2, direction, distance3 [,mode])`

ALTERNATE FORMAT:

`MH()`

DESCRIPTION:

Performs a helical move.

Moves 2 orthogonal axes in such a way as to produce a circular arc at the tool point with a simultaneous linear move on a third axis. The first 5 parameters are similar to those of an **MOVECIRC** command. The sixth parameter defines the simultaneous linear move.

PARAMETERS:

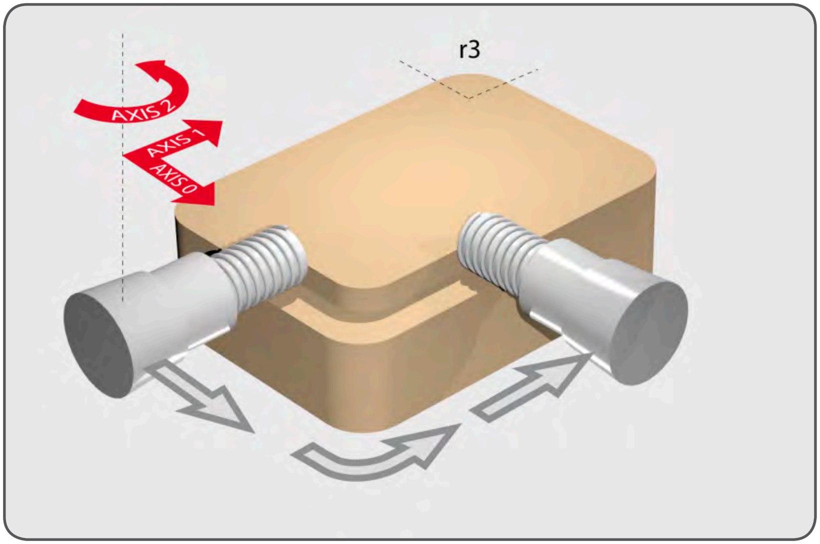
end1:	position on **BASE** axis to finish at.	
end2:	position on next axis in **BASE** array to finish at.	
centre1:	position on **BASE** axis about which to move.	
centre2:	position on next axis in **BASE** array about which to move.	
direction:	0	Arc is interpolated in an anti-clockwise direction
	1	Arc is interpolated in a clockwise direction
distance3:	The distance to move on the third axis in the **BASE** array axis in user units	
mode:	0	Interpolate the 3rd axis with the main 2 axes when calculating path speed. (True helical path)
	1	Interpolate only the first 2 axes for path speed, but move the 3rd axis in coordination with the other 2 axes. (Circular path with following 3rd axis)

The first 4 distance parameters are scaled according to the current unit conversion factor for the *BASE* axis. The sixth parameter uses its own axis units.

EXAMPLES:

EXAMPLE1:

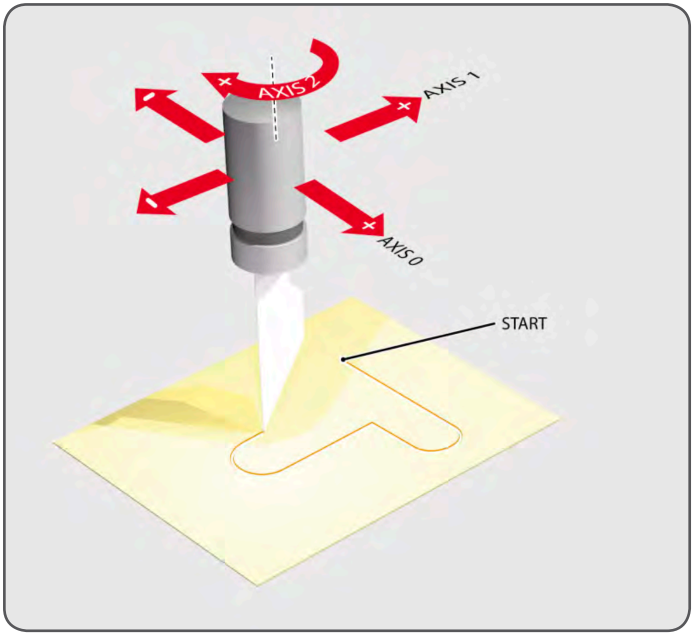
The command sequence follows a rounded rectangle path with axis 1 and 2. Axis 3 is the tool rotation so that the tool is always perpendicular to the product. The **UNITS** for axis 3 are set such that the axis is calibrated in degrees.




```
REP_DIST AXIS(3)=360
REP_OPTION AXIS(3)=ON ‘all 3 axes must be homed before starting
MERGE=ON
MOVEABS(360) AXIS(3) ‘point axis 3 in correct starting direction
WAIT IDLE AXIS(3)
MOVE(0,12)
MHELICAL(3,3,3,0,1,90)
MOVE(16,0)
MHELICAL(3,-3,0,-3,1,90)
MOVE(0,-6)
MHELICAL(-3,-3,-3,0,1,90)
MOVE(-2,0)
MHELICAL(-3,3,0,3,1,90)
```

EXAPMLE 2:

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




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

SEE ALSO:

MOVECIRC

MHELICALSP

TYPE:

Axis Command.

SYNTAX:

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

DESCRIPTION:

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

EXAMPLE:

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

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

SEE ALSO:

MHELICAL

MID

TYPE:

STRING Function

SYNTAX:

MID(string, start[, length])

DESCRIPTION:

Returns the mid-section of the specified string using the optional length specified, or defaults to the remainder of the string when not specified.

PARAMETERS:

string:	String to be used
start	Start index of string

length:	Length of string to be returned, if not specified then the remainder of the string will be used

EXAMPLES:

EXAMPLE 1:

Pre-define a variable of type string and later print characters: from index 5 to 10

```
DIM str1 AS STRING(32)
str1 = “TRIO MOTION TECHNOLOGY”
PRINT MID(str1, 5, 6)
```

SEE ALSO:

CHR, STR, VAL, LEN, LEFT, RIGHT, LCASE, UCASE, INSTR

MOD

TYPE:

Mathematical Operator

SYNTAX:

`value = expression1  MOD(expression2)`

DESCRIPTION:

Returns the integer modulus of an expression, this is the value after the integer has wrapped around the modulus

PARAMETERS:

value:	the modulus of expression 1
expression1:	Any valid TrioBASIC expression used as the value to apply the modulus to.
expression2:	Any valid TrioBASIC expression used as the modulus

EXAMPLE:

Use the MOD(12) to turn a 24 hour value into 12 hour.

```
>>PRINT 18 MOD(12)
6.0000
>>
```

MODBUS

TYPE:

System Function

SYNTAX:

`MODBUS(function, slot [, parameters…])`

DESCRIPTION:

This function allows the user to configure the Ethernet port to run as a Modbus TCP Client (Master). Using the **MODBUS** command, the user can open a connection to a remote server, transfer data using a sub-set of Modbus Function Numbers and check for errors.

PARAMETERS:

function:	0	Open a ModbusTCP client connection
	1	Close connection
	2	Check connection status
	3	Send commands (Modbus functions)
	$10	Get Error Log Entry
	$11	Get Error Log Count

FUNCTION = 0;

SYNTAX:

`value = MODBUS(0,slot , ip address 1...4 [, port number [,vr_index]])`

DESCRIPTION:

Attempt to open a ModbusTCP client connection to the given remote server.

PARAMETERS:

value:	**TRUE** = the command was successful
	FALSE = the command was unsuccessful
slot:	Module slot in which the communication port is fitted

ip address:	Server's IP address as 4 octets separated by commas
port number:	Optional port number. Default is port 502 if none given.
vr_index:	Index number of the **VR** where the connection handle will be written. Default value is -1. -1 means print to the standard output stream. (normally terminal 0)

EXAMPLE:

```
'IP Address 192.168.0.185, Port Number 502
IF MODBUS(0,-1,192,168,0,185,502,20)=TRUE THEN
  PRINT "Modbus port opened OK"
  modbus_handle = VR(20)
ELSE
  PRINT "Error, Modbus server not found"
ENDIF
```

FUNCTION = 1:

SYNTAX:

`value = MODBUS(1,slot,handle)`

DESCRIPTION:

Close ModbusTCP client connection if open.

PARAMETERS:

value:	**TRUE**	the command was successful
	FALSE	the command was unsuccessful or the connection was already closed
slot:	Module slot in which the communication port is fitted	
handle:	number that was returned by the previous "open" function	

EXAMPLE:

```
'Close Modbus connection
MODBUS(1,-1,modbus_handle)
```

FUNCTION = 2:

SYNTAX:

`value = MODBUS(2, slot, handle [,VR index])`

DESCRIPTION:

Return connection status (0 = closed, 1 = open)

PARAMETERS:

value:	**TRUE**	the command was successful
	FALSE	the command was unsuccessful
slot:	Module slot in which the communication port is fitted	
handle:	number that was returned by the previous “open” function or 0 which checks for any open handle	
VR index:	VR number which will hold the returned value. If set to -1 or not included, then the value is printed to the command-line terminal	

EXAMPLE:

EXAMPLE 1

```
'Is Modbus connection open?
MODBUS(2, -1, 200)
IF VR(200)=1 THEN
  PRINT “Modbus port is open”
ELSE
  PRINT “Modbus port is closed”
ENDIF
```

EXAMPLE 2

```
>>MODBUS(2, -1, -1)
1
```

FUNCTION = 3:

SYNTAX:

`value = MODBUS(3, slot, handle, modbus function code [, parameters])`

DESCRIPTION:

Execute the given Modbus function if the connection is open. The parameters vary depending upon the function required. Holding Registers are mapped to the corresponding **VR** in the client. IO functions use the **VR**s to hold the remote IO states when reading from the remote server, or as the IO source when writing to the remote server. Each **VR** entry is used to hold up to 32 IO bits. The Modbus functions supported are defined below.

PARAMETERS:

value:	**TRUE**	the command was successful
	FALSE	the command was unsuccessful
slot:	Module slot in which the communication port is fitted	
handle:	Handle of the previously opened connection	
Modbus function code:	A recognised valid Modbus function code number	
Other parameters:	See table below	

Function	#	Parameters	Notes
Read Coils	1	Start Address	
		Number of values	
		Result start address	VR index for response values
Read Discrete Inputs	2	Start Address	
		Number of values	
		Result start address	VR index for response values
Read Holding Registers	3	Start Address	Modbus register start address in Server. Data read is mapped directly to same **VR**s in the client unless Local Address is set.
		Number of values	
		Local Address	If set, this is the target **VR** start address in the *Motion Coordinator* client.
Read Input Registers	4	Start Address	Data read directly into **VR**s
		Number of values	
Write Single Coil	5	Address	
		Value	1 (on) or 0 (off)
Write Single Register	6	Address	Modbus register address in server. Value is taken from the same client **VR** unless Local Address is set.
		Local Address	If set, this is the target **VR** address in the *Motion Coordinator* client.

Function	#	Parameters	Notes
Write Multiple Coils	15	Start Address	
		Number of coils	
		Source address	VR start address containing required coil state values.
Write Multiple Registers	16	Start Address	Modbus register start address in server. Values are copied from the same **VR** address in the client unless the Local Address is set.
		Number of registers	
		Local Address	If set, this is the target **VR** start address in the *Motion Coordinator* client.
Read Write Multiple Registers	23	Read Start address	Mapped to same **VR**s in Client
		Number of Read registers	
		Write Start address	Mapped from same **VR**s in Client.
		Number of Write registers	

EXAMPLE

```
my_slot=-1

open_modbus = $00
close_modbus = $01
get_status = $02
ex_modbus_func = $03
get_error_log = $10


‘ check if Modbus is already open
MODBUS(get_status, my_slot, 100)
IF VR(100)=1 THEN
    ‘ close the connection so that it can be re-opened
    MODBUS(close_modbus, my_slot)
ENDIF

‘ open the modbus server (remote slave) & put handle in VR(20)
MODBUS(open_modbus, my_slot, 192,168,000,249,502,20)

REPEAT
    ‘ get 10 values from holding registers 1000 to 1009
    MODBUS(ex_modbus_func, my_slot, VR(20), 3, 1000, 10)
    ‘ send 10 values to holding registers 1010 to 1019
```

```
        MODBUS(ex_modbus_func, my_slot, VR(20), 16, 1010, 10)
        WA(200)
    UNTIL FALSE
```

FUNCTION = $10:

SYNTAX:

MODBUS($10, slot, handle [,entry offset [,VR index]])

DESCRIPTION:

Returns the error log entry. If no entry offset is supplied, then the last entry (offset = 0) is returned. Otherwise, 1 will return the previous entry, 2 will return the last one but 2 etc.

PARAMETERS:

value:	**TRUE**	the command was successful
	FALSE	the command was unsuccessful
slot:	Module slot in which the communication port is fitted	
handle:	Handle of the connection whose error log entry is required. If -1 then access general protocol errors (for example failed to open connection.)	
entry offset:	Entry in the error log. If not supplied then entry 0 is returned.	
VR index:	VR number which will hold the returned value. If set to -1 or not included, then the value is printed to the command-line terminal.	

EXAMPLE:

EXAMPLE 1

```
    'Get error log entries 0 to 4 and put in VR(100) to VR(104)
    FOR i=0 to 4
      error_flag = MODBUS($10, -1, modbus_handle, i, 100+i)
      IF error_flag = FALSE THEN
        PRINT "Error fetching error log entry ";i[0]
      ENDIF
    NEXT i
```

EXAMPLE 2

```
    'Get an error log entry from the terminal
    >>MODBUS($10, -1, modbus_handle, 0, -1)
    19
```

FUNCTION = $11:

SYNTAX:
MODBUS($11, slot, handle [,vr_index])

DESCRIPTION:
Return the count of the number of error codes logged for the given handle.

PARAMETERS:

value:	**TRUE**	the command was successful
	FALSE	the command was unsuccessful
slot:	Module slot in which the communication port is fitted	
handle:	Handle of the connection whose error log entry is required. If -1 then access general protocol errors (for example failed to open connection.)	
VR index:	VR number which will hold the returned value. If set to -1 or not included, then the value is printed to the command-line terminal.	

MODULE_IO_MODE

TYPE:
System Parameter (**MC_CONFIG** / **FLASH**)

DESCRIPTION:
This parameter sets the start address of any expansion module I/O channels. You can also turn off module I/O for backwards compatibility.

Note that extended IO mapping functionality is available using **MC_CONFIG** parameters **CANIO_BASE**, **DRIVEIO_BASE**, **MODULEIO_BASE** and **NODE_IO**. These replace the need to use **MODULE_IO_MODE** and provide control over exactly where IO points are positioned within the Controller IO map. However, if **MODULE_IO_MODE** is set to 2 then this takes precedence over the positioning of **CANIO** and **MODULE** IO via **CANIO_BASE** and **MODULEIO_BASE**.

This parameter is stored in Flash EPROM and can be included in the MC_**CONFIG** script.

VALUE:

0	Module I/O disabled

1	Module I/O is after controller I/O and before CAN I/O (default)
2	Module I/O is at the end of the I/O sequence
3	Module I/O disabled and CAN I/O starts at 32

If you are upgrading the firmware in an existing controller, this parameter may be set to 0. The default of 1 is on a factory installed system.

EXAMPLE:

A system with MC464, a Panasonic module (slot 0), a FlexAxis (slot 1) and a **CANIO** Module will have the following I/O assignment:

MODULE_IO_MODE=1 (default) + **DRIVEIO_BASE**=-1 + **CANIO_BASE**=0 + **MODULEIO_BASE**=0

0-7	Built in inputs
8-15	Built in bi-directional I/O
16-23	Panasonic inputs
24-27	FlexAxis inputs
28-31	FlexAxis bi-directional I/O
32-47	**CANIO** bi-directional I/O
48-1023	Virtual I/O

MODULE_IO_MODE=0 (off) + **DRIVEIO_BASE**=-1 + **CANIO_BASE**=0 + **MODULEIO_BASE**=0

0-7	Built in inputs
8-15	Built in bi-directional I/O
16-31	**CANIO** bi-directional I/O
32-1023	Virtual I/O

MODULE_IO_MODE=2 (end)

0-7	Built in inputs
8-15	Built in bi-directional I/O
16-31	**CANIO** bi-directional I/O

32-39	Panasonic inputs
40-43	FlexAxis inputs
44-47	FlexAxis bi-directional I/O
48-1023	Virtual I/O

SEE ALSO:
CANIO_BASE, DRIVEIO_BASE, MODULEIO_BASE, NODE_IO

MODULEIO_BASE

TYPE:
System Parameter (**MC_CONFIG**)

DESCRIPTION:
This parameter sets the start address of any expansion module I/O channels. Together with **CANIO_BASE**, **DRIVEIO_BASE** and **NODE_IO** the I/O allocation scheme can replace and expand the behaviour of **MODULE_IO_MODE**, however **MODULE_IO_MODE** takes precedence if its value has been changed to 2 (**CANIO** followed by **MODULE** IO).

VALUE:

-1	Module I/O disabled
0	Module I/O allocated automatically (default)
>= 8	Module I/O is located at this IO point address, truncated to the nearest multiple of 8

EXAMPLE:
A system with MC464, a Panasonic module (slot 0) and a **CANIO** Module will have the following I/O assignment:

MODULEIO_BASE=0 + **DRIVEIO_BASE**=0 + **CANIO_BASE**=0

0-7	Built in inputs
8-15	Built in bi-directional I/O
16-23	Panasonic module inputs
24-39	**CANIO** bi-directional I/O
40-47	Panasonic drive inputs
48-1023	Virtual I/O

MODULEIO_BASE=-1 + **DRIVEIO_BASE**=0 + **CANIO_BASE**=0

0-7	Built in inputs
8-15	Built in bi-directional I/O
16-31	**CANIO** bi-directional I/O
32-39	Panasonic drive inputs
40-1023	Virtual I/O

MODULEIO_BASE=200 + **DRIVEIO_BASE**=0 + **CANIO_BASE**=0

0-7	Built in inputs
8-15	Built in bi-directional I/O
16-31	**CANIO** bi-directional I/O
32-39	Panasonic drive inputs
40-199	Virtual I/O
200-207	Panasonic module inputs
208-1023	Virtual I/O

SEE ALSO:
CANIO_BASE, DRIVEIO_BASE, NODE_IO, MODULE_IO_MODE

MOTION_ERROR

TYPE:
System Parameter (read only)

DESCRIPTION:
The **MOTION_ERROR** provides a simple single indicator that at least one axis is in error and can indicate multiple axes that have an error.

VALUE:
A sum of the bits representing each axis that is in error.

Bit	Value	Axis
0	1	0
1	2	1
2	4	2
3	8	3
...		

EXAMPLE:

MOTION_ERROR=11 and **ERROR_AXIS**=3 indicates axes 0, 1 and 3 have an error and the axis 3 occurred first.

SEE ALSO:

AXISSTATUS, ERROR_AXIS

MOVE

TYPE:

Axis Command

SYNTAX:

MOVE(distance1 [,distance2 [,distance3 [,distance4...]]])

ALTERNATE FORMAT:

MO()

DESCRIPTION:

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

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

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

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

PARAMETERS:

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

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

EXAMPLES

EXAMPLE 1:

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

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

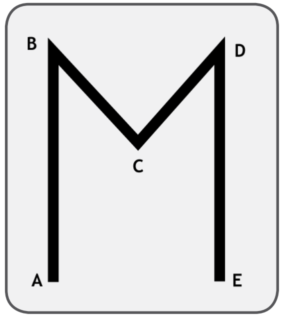
EXAMPLE 2:

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

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

EXAMPLE 3:

An X-Y plotter can write text at any position within its working envelope. Individual characters are defined as a sequence of moves relative to a start point so that the same commands may be used regardless of the plot origin. The command subroutine for the letter ‘M’ might be:




```
write_m:
  MOVE(0,12) ‘move A >  B
  MOVE(3,-6) ‘move B >  C
  MOVE(3,6)  ‘move C >  D
  MOVE(0,-12)’move D > E
  RETURN
```

MOVE_COUNT

TYPE:

Axis Parameter

DESCRIPTION:

MOVE_COUNT increments every time a motion command loads into the **MTYPE** buffer or when a command is automatically re-loaded such as **FLEXLINK**.

MOVE_COUNT can be written to set an initial value.

VALUE:

The number of movements loaded into the **MTYPE** buffer.

EXAMPLE:

Run the motion program and then turn on the OP(11) after 10 moves have been loaded.

```
MOVE_COUNT = 0
RUN “MOTION”
WAIT UNTIL MOVE_COUNT > 10
OP(11,ON)
```

MOVEABS

TYPE:

Axis Command.

SYNTAX:

`MOVEABS(position1[, position2[, position3[, position4...]]])`

ALTERNATE FORMAT:

`MA()`

DESCRIPTION:

Absolute position move. Move one axis or multiple axes to position(s) referenced with respect to the zero (home) position. The first parameter in the list is sent to the axis specified with the **AXIS** command or to the current **BASE** axis, the second to the next axis, and so on.

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

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

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

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

PARAMETERS:

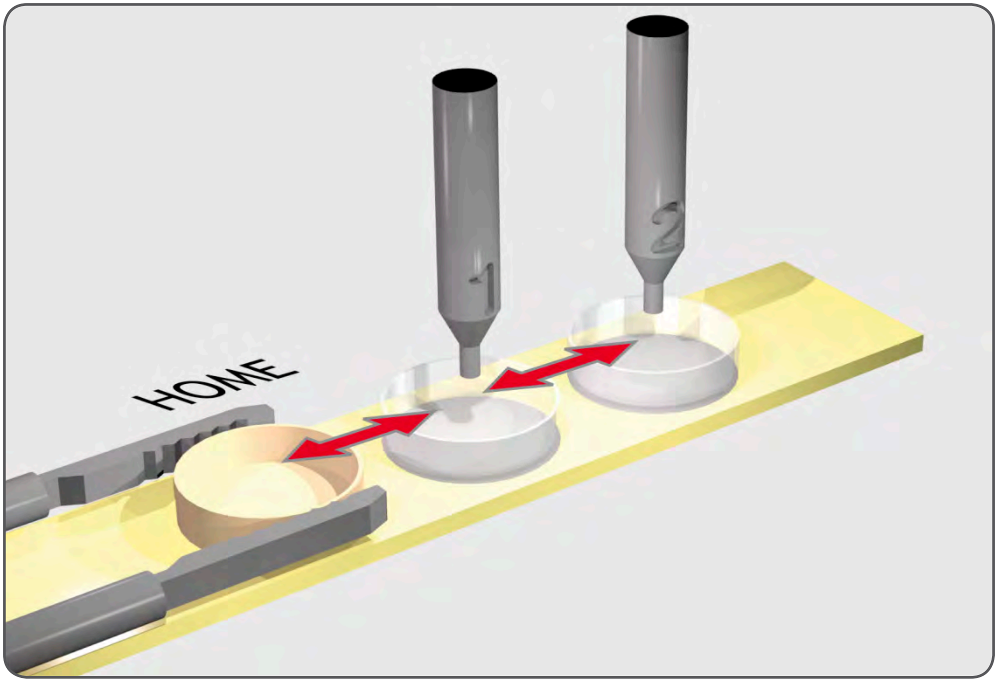
position1:	position to move to on base axis.
position2:	position to move to on next axis in **BASE** array.
position3:	position to move to on next axis in **BASE** array.
position4:	position to move to on next axis in **BASE** array

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

EXAMPLES:

EXAMPLE 1:

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




```
'define absolute positions
home=1000
position_1=2000
position_2=3000
WHILE IN(run_switch)=ON
  IF IN(6)=ON THEN       'switch 6 selects position 2
    MOVEABS(position_2)
    WAIT IDLE
  ELSEIF IN(7)=ON THEN   'switch 7 selects position 1
    MOVEABS(position_1)
    WAIT IDLE
  ELSE
    MOVEABS(home)
    WAIT IDLE
  ENDIF
WEND
```

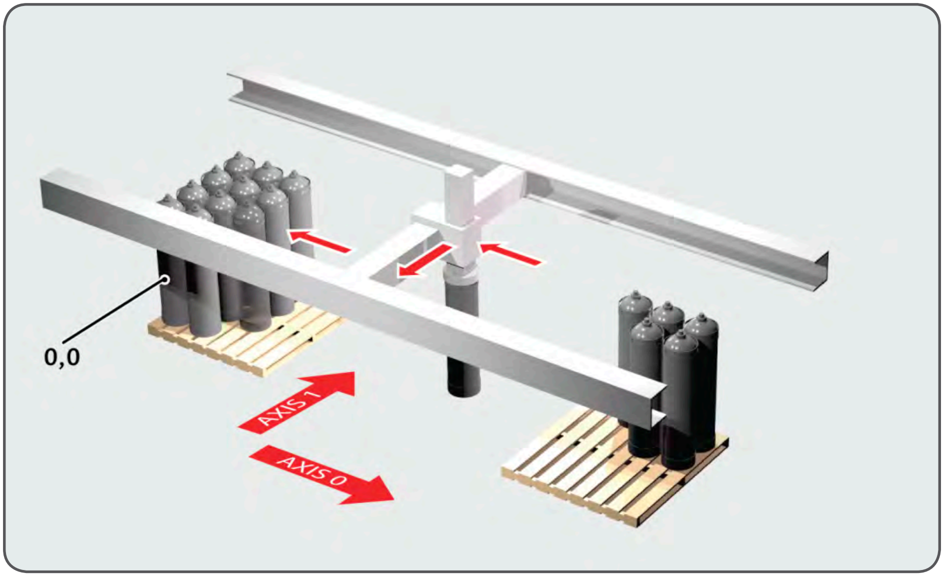
EXAMPLE 2:

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

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

EXAMPLE 3:

A pallet consists of a 6 by 8 grid in which gas canisters are inserted 185mm apart by a packaging machine. The canisters are picked up from a fixed point. The first position in the pallet is defined as position 0,0 using the **DEFPOS**() command. The part of the program to position the canisters in the pallet is:




```
FOR x=0 TO 5
  FOR y=0 TO 7
    MOVEABS(-340,-516.5)      ‘move to pick-up point
    WAIT IDLE
    GOSUB pick                ‘call pick up subroutine
    PRINT “Move to Position: “;x*6+y+1
    MOVEABS(x*185,y*185)      ‘move to position in grid
    WAIT IDLE
    GOSUB place               ‘call place down subroutine
  NEXT y
NEXT x
```

EXAMPLE 4:

Using **MOVEABS** with **REP_DIST** to move to a final position.

```
REPDIST = 360
DEFPOS(0)
MOVEABS(300)  'will move through 300d egrees to 300
MOVEABS(200)  'will move back 100 degrees to 200
MOVEABS(370)  'will move through 170 degrees to 10 crossing repdist
MOVEABS(350)  'will move through 340 degrees to 350
```

if you want to move in the shortest direction to the absolute position use **MOVETANG**

SEE ALSO:

MOVETANG

MOVEABSSEQ

TYPE:

Axis Command

SYNTAX:

MOVEABSSEQ(table pointer, axes, npoints, options, radius)

DESCRIPTION:

The **MOVEABSSEQ** command allows a sequence of 2 or 3 axis movements to be loaded via **TABLE** values. The moves can be automatically merged together using a circular or spherical arc.

The **MOVEABSSEQ** is loaded into the controller move buffers as a sequence of **MOVEABS**->**MOVECIRC**-> moves if 2 axes are specified and **MOVEABS**->**MSPHERICAL**-> if 3 axes are specified. The linear move may be omitted if the arcs blend together. If "Options" is set to 1 the move sequence loaded will be a sequence of **MOVEABSSP**->**MOVECIRCSP**-> moves if 2 axes are specified and **MOVEABSSP**->**MSPHERICALSP**-> if 3 axes are specified.

MOVE_COUNT is incremented on every move loaded.

The fillet Radius will automatically be reduced to the maximum possible if the points specified are insufficiently far apart to apply the fillet.

The current axes positions at the start of the **MOVEABSSEQ** are used for calculating the first fillet.

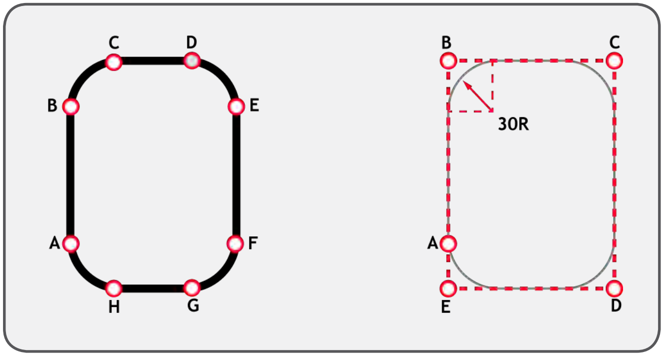
PARAMETERS:

Table pointer:	Location of the absolute points in **TABLE** memory.
Axes:	Number of axes 2 or 3.

Npoints:	The number of points, each point requires 2 or 3 table values.
Options	0 sets to load **MOVEABS** etc, 1 set to load embedded speed moves **MOVEABSSP** etc.
Radius	The merging/filleting radius to be applied. 0 for no filleting.

EXAMPLE:

Draw O using separate **MOVE** and **MOVECIRC**(see Trio Manual **MOVECIRC**), and draw similar O using **MOVEABSSEQ**.




```
'MOVE and MOVECIRC:
MOVE(0,60) ' move A -> B
MOVECIRC(30,30,30,0,1) ' move B -> C
MOVE(20,0) ' move C -> D
MOVECIRC(30,-30,0,-30,1)' move D -> E
MOVE(0,-60) ' move E -> F
MOVECIRC(-30,-30,-30,0,1)' move F -> G
MOVE(-20,0) ' move G -> H
MOVECIRC(-30,30,0,30,1) ' move H -> A
WAIT IDLE
DEFPOS(100,30)
WAIT UNTIL OFFPOS=0

' MOVEABSSEQ:
TABLE(1000,100,120)
TABLE(1002,180,120)
TABLE(1004,180,0)
TABLE(1006,100,0)
TABLE(1008,100,30)

MOVEABSSEQ(1000,2,5,0,30)
```

MOVEABSSP

TYPE:
Axis Command.

SYNTAX:
MOVEABSSP(position1[, position2[, position3[, position4…]]])

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

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

PARAMETERS:

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

The maximum number of parameters is the number of axes available on the controller.

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

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

SEE ALSO:
MOVEABS

MOVECIRC

TYPE:
Axis Command.

SYNTAX:
MOVECIRC(end1, end2, centre1, centre2, direction)

ALTERNATE FORMAT:
MC()

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

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

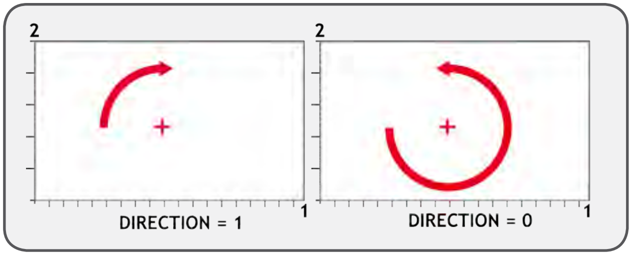
If the end point specified is not on the circular arc. The arc will end at the angle specified by a line between the centre and the end point.

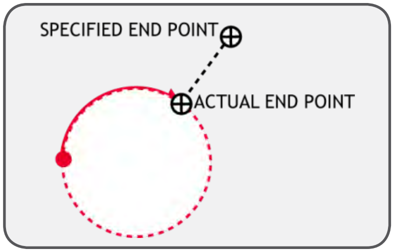
Neither axis may cross the set absolute repeat distance (**REP_DIST**) during a **MOVECIRC**. Doing so may cause one or both axes to jump or for their **FE** value to exceed **FE_LIMIT**.

PARAMETERS:

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

direction:	0	Arc is interpolated in an anti-clockwise direction
	1	Arc is interpolated in a clockwise direction
	2	Arc is interpolated using the shortest path to endpoint
	3	Arc is interpolated using the longest path to endpoint



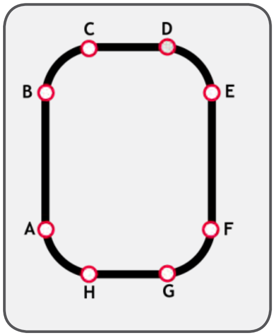


EXAMPLES:

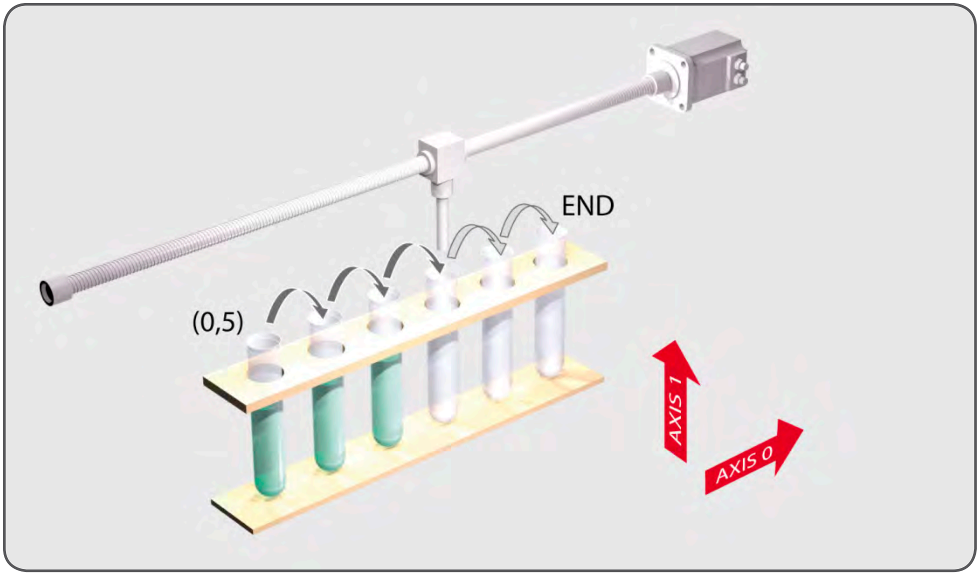
EXAMPLE 1:

The command sequence to plot the letter ‘0’ might be:

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



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




```
BASE(0,1)
MOVEABS(0,5)            'move to position above first tube
MOVEABS(0,0)            'lower for first drop
WAIT IDLE
OP(15,ON)               'apply dropper
WA(20)
OP(15,OFF)
```

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

MOVECIRCSP

TYPE:

Axis Command.

SYNTAX:

MOVECIRCSP(end1, end2, centre1, centre2, direction)

DESCRIPTION:

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

EXAMPLE:

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

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

SEE ALSO:

MOVECIRC

MOVELINK

TYPE:

Axis Command.

SYNTAX:

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

ALTERNATE FORMAT:

ML()

DESCRIPTION:

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

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

The motion consists of a linear movement with separately variable acceleration and deceleration phases linked via a software gearbox to the **MEASURED** position (**MPOS**) of another axis. The command uses the **BASE**() and **AXIS**(), and unit conversion factors in a similar way to other move commands.

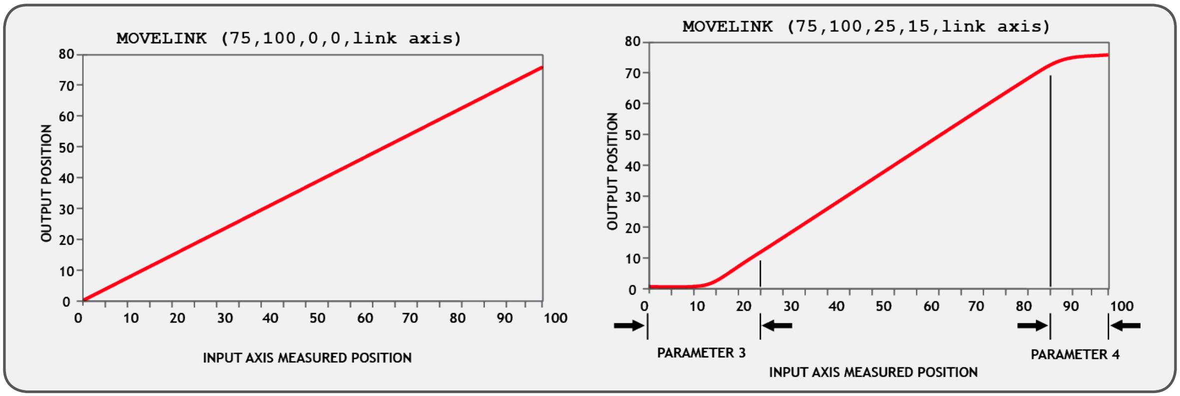
The “link” axis may move in either direction to drive the output motion. The link distances specified are always positive.

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.
link axis:	Specifies the axis to “link” to. It should be set to a value between 0 and the number of available axes.

link_options:	Bit value options to customize how your **MOVELINK** operates		
	Bit 0	1	link commences exactly when registration event **MARK** occurs on link axis
	Bit 1	2	link commences at an absolute position on link axis (see link_pos for start position)
	Bit 2	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)
	Bit 4	16	If this bit is set the **MOVELINK** acceleration and deceleration phases are constructed using an "S" speed profile not a trapezoidal speed profile
	Bit 5	32	Link is only active during a positive move on the link axis
	Bit 8	256	link commences exactly when registration event **MARKB** occurs on link axis
	Bit 9	512	link commences exactly when registration event **R_MARK** occurs on link axis. (see link_pos for channel number)
link_pos:	link_option bit 1 - the absolute position on the link axis in user **UNITS** where the **CAMBOX** is to be start. link_option bit 9 – the registration channel to start the movement on		

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.



The link_dist is in the user units of the link axis and should always be specified as a positive distance.

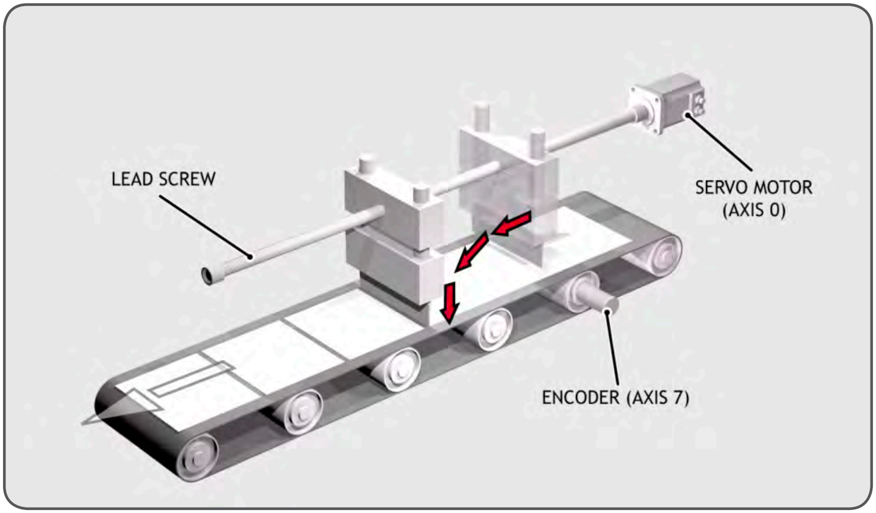
The link options for start (bits 1, 2, 8 and 9) may be combined with the link options for repeat (bits 4 and 8) and direction.

start_pos cannot be at or within one servo period's worth of movement of the **REP_DIST** position.

EXAMPLES:

EXAMPLE 1:

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




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

In this program the controller firstly waits for the roll to feed out 150m in the first line. After this distance the shear accelerates up to match the speed of the paper, moves at the same speed then decelerates to a stop within the 1m stroke. This movement is specified using two separate **MOVELINK** commands. This allows the program to wait for the next move buffer to be clear, **NTYPE**=0, which indicates that the acceleration phase is complete. Note that the distances on the measurement axis (link distance in each **MOVELINK** command): 150, 0.8, 1.0 and 8.2 add up to 160m.

To ensure that speed and positions of the cutter and paper match during the cut process the parameters of the **MOVELINK** command must be correct: It is normally easiest to consider the acceleration, constant speed and deceleration phases separately then combine them as required:

RULE 1:

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

```
MOVELINK(0.3,0.6,0.6,0,1)' move is all accel
```

RULE 2:

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

```
MOVELINK(0.4,0.4,0,0,1)' all constant speed
```

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

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

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

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

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

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

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

EXAMPLE 2:

EXACT RATIO GEARBOX

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

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

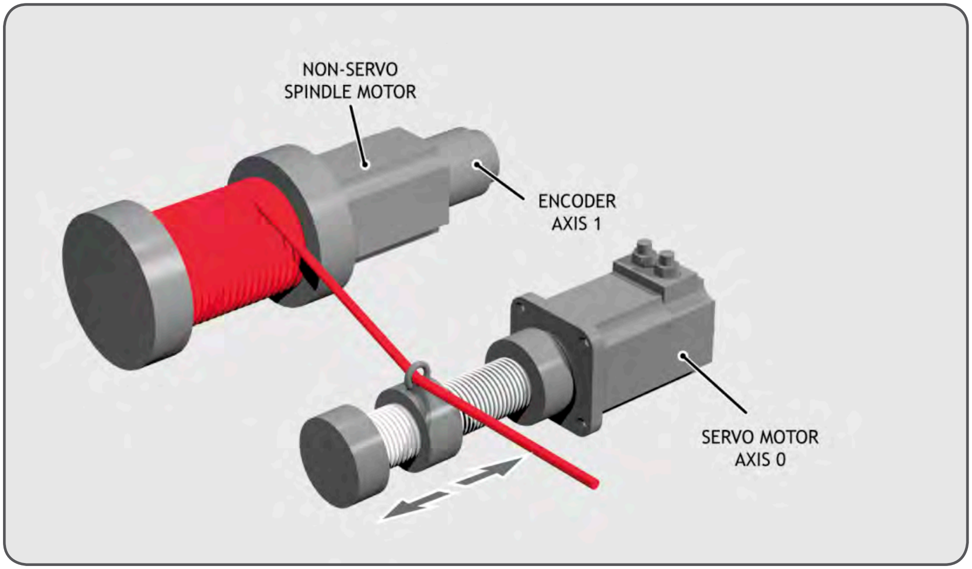
EXAMPLE 3:

COIL WINDING

In this example the unit conversion factors **UNITS** are set so that the payout movements are in mm and the spindle position is measured in revolutions. The payout eye therefore moves 50mm over 25 revolutions of the spindle with the command:

```
MOVELINK(50,25,0,0,linkax).
```

If it were desired to accelerate up over the first spindle revolution and decelerate over the final 3 the command would be




```
MOVELINK(50,25,1,3,linkax)
OP(motor,ON)   '- Switch spindle motor on
FOR layer=1 TO 10
  MOVELINK(50,25,0,0,1)
  MOVELINK(-50,25,0,0,1)
NEXT layer
WAIT IDLE
OP(motor,OFF)
```

MOVEMODIFY

TYPE:
Axis Command.

SYNTAX:
`MOVEMODIFY(position)`

ALTERNATE FORMAT:
`MM()`

DESCRIPTION:

MOVEMODIFY will change the absolute end position of a single axis **MOVE**, **MOVEABS**, **MOVESP**, **MOVEABSSP** or **MOVEMODIFY** that is in the last position in the movement buffer. If there is no motion command in the movement buffers or the last movement is not a single axis linear move then **MOVEMODIFY** is loaded.

If the change in end position requires a change in direction the move in **MTYPE** is **CANCELed.** This will use **DECEL** unless **FASTDEC** has been specified.

If there are multiple buffered linier moves the **MOVEMODIFY** will only act on the command in front of it in the buffer.

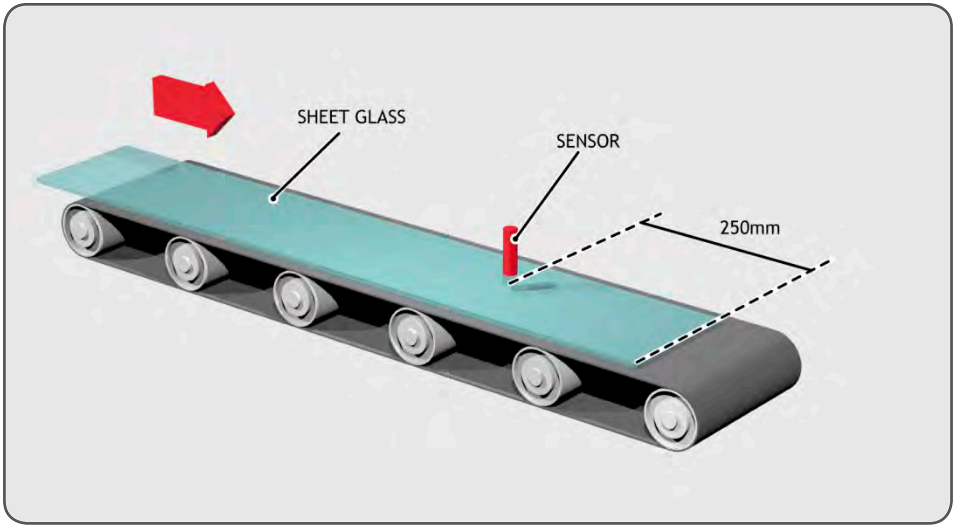
PARAMETERS:

position:	Absolute position for the current move to complete at.

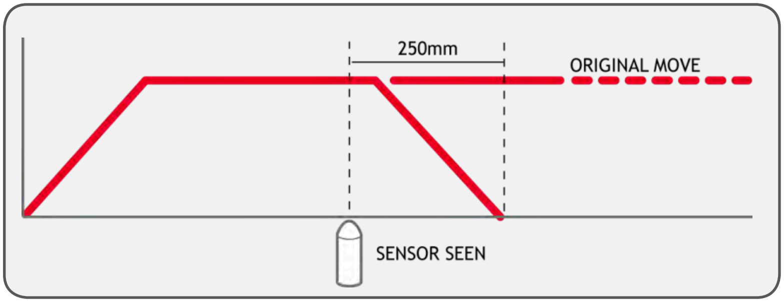
EXAMPLES:

EXAMPLE 1:

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

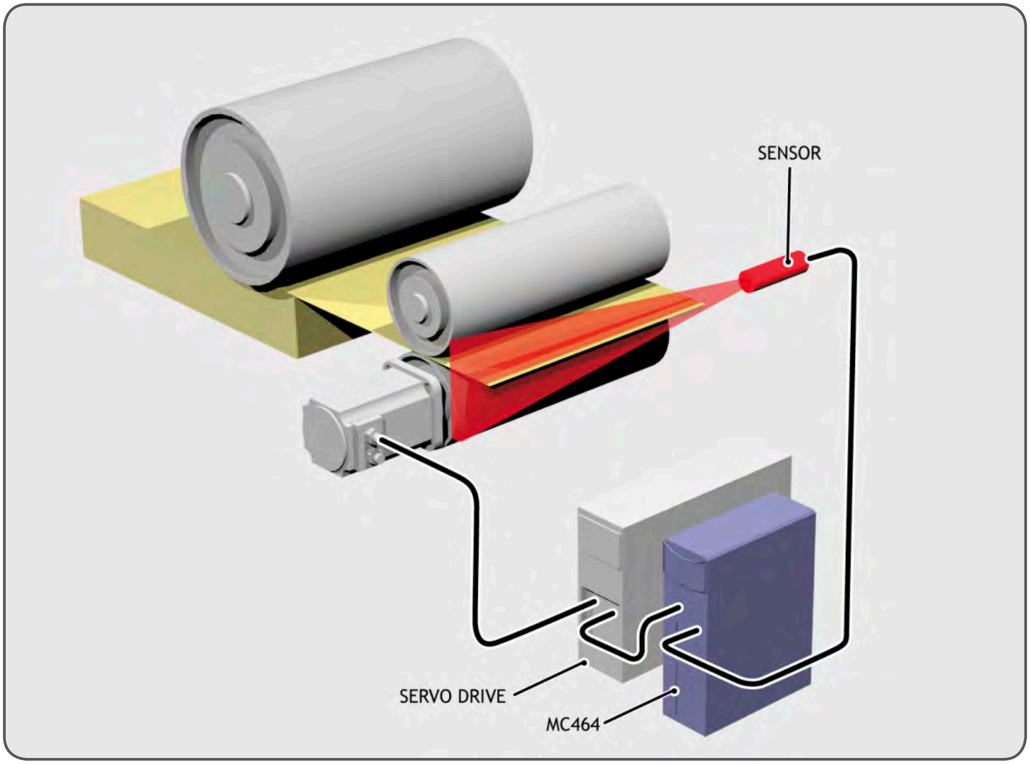



```
MOVE(10000)          'Start a long move on conveyor
REGIST(3)            'set up registration
WAIT UNTIL MARK      'MARK goes TRUE when sensor detects glass edge
OFFPOS = -REG_POS    'set position where mark was seen to 0
WAIT UNTIL OFFPOS=0  'wait for OFFPOS to take effect
MOVEMODIFY(250)      'change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



EAMPLE 2:

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

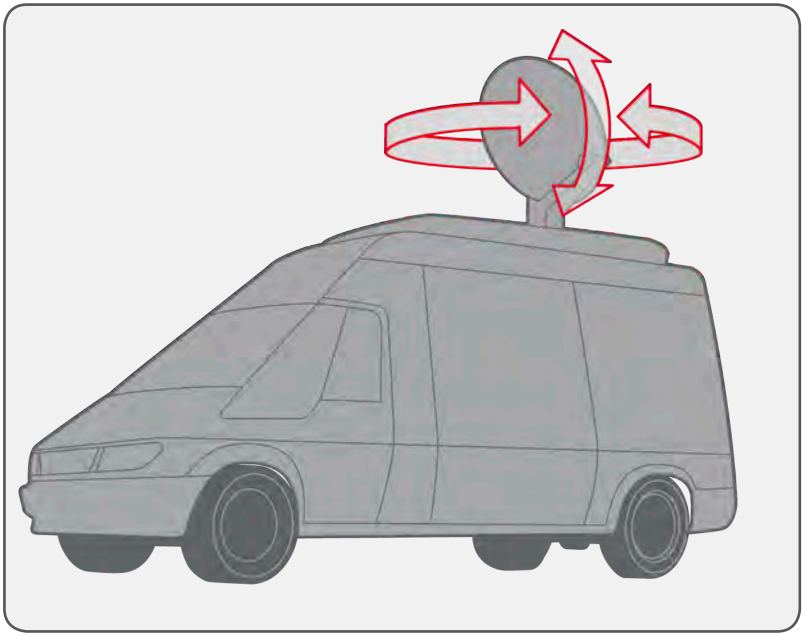



```
paper_length=4000
```

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

EXAMPLE 3:

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



```
bearing=0           ‘set labels for VRs
elevation=1
UNITS AXIS(0)=360/counts_per_rev0
UNITS AXIS(1)=360/counts_per_rev1
WHILE IN(2)=ON
  MOVEMODIFY(VR(bearing))AXIS(0)   ‘adjust bearing to match VR0
  MOVEMODIFY(VR(elevation))AXIS(1) ‘adjust elevation to match VR1
  WA(250)
WEND
```

```
RAPIDSTOP              'stop movement
WAIT IDLE AXIS(0)
MOVEABS(0) AXIS(0)     'return to transport position
WAIT IDLE AXIS(1)
MOVEABS(0) AXIS (1)
```

SEE ALSO:
ENDMOVE

MOVES_BUFFERED

TYPE:
Axis Parameter (Read only)

DESCRIPTION:
This returns the number of moves being buffered by the axis.

The value does not include the move in the *MTYPE* buffer.

PARAMETERS:

value:	number of commands in the move buffers.

EXAMPLE:
Check if there is room in the move buffer before adding in another command.

```
IF MOVES_BUFFERED < 64 THEN
  xpos = TABLE(count+x)
  ypos = TABLE(count+y)
  MOVEABS(xpos, ypos)
  count=count + 1
ENDIF
```

MOVESEQ

TYPE:
Axis Command

SYNTAX:
MOVESEQ(table pointer, axes, npoints, options, radius)

DESCRIPTION:

The **MOVESEQ** command allows a sequence of 2 or 3 axis movements to be loaded via **TABLE** values. The moves can be automatically merged together using a circular or spherical arc.

The **MOVESEQ** is loaded into the controller move buffers as a sequence of **MOVE**->**MOVECIRC**-> moves if 2 axes are specified and **MOVE**->**MSPHERICAL**-> if 3 axes are specified. The linear move may be omitted if the arcs blend together. If “Options” is set to 1 the move sequence loaded will be a sequence of **MOVESP**->**MOVECIRCSP**-> moves if 2 axes are specified and **MOVESP**->**MSPHERICALSP**-> if 3 axes are specified.

MOVE_COUNT is incremented on every move loaded.

The fillet Radius will automatically be reduced to the maximum possible if the points specified are insufficiently far apart to apply the fillet.

The current axes positions at the start of the **MOVESEQ** are used for calculating the first fillet.

PARAMETERS:

Table pointer:	Location of the absolute points in **TABLE** memory.
Axes:	Number of axes 2 or 3.
Npoints:	The number of points, each point requires 2 or 3 table values.
Options	0 sets to load **MOVE** etc, 1 set to load embedded speed moves **MOVESP** etc.
Radius	The merging/filleting radius to be applied. 0 for no filleting.

EXAMPLE:

Draw a sequence of movements using **MOVESEQ**:

```
FOR x = 0 TO 2
    BASE(x)
    ATYPE = 0
    UNITS = 100
    ACCEL = 500
    DECEL = ACCEL
    SERVO = ON
    SPEED = 100
NEXT x

BASE(0,1,2)

DEFPOS(100,0,0)
WAIT UNTIL OFFPOS=0

TABLE(1000,-100,0,0)
TABLE(1003,0,200,0)
TABLE(1006,200,0,0)
TABLE(1009,0,200,0)
```

```
TABLE(1012,150,0,0)
TABLE(1015,-50,-400,0)
TABLE(1018,-300,-200,0)

TRIGGER
WA(10)

MOVESEQ(1000,3,7,1,300)
WAIT IDLE
```

MOVESP

TYPE:
Axis Command

SYNTAX:
MOVESP(distance1[,distance2[,distance3[,distance4…]]])

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

PARAMETERS:

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

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

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

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

SEE ALSO:
MOVE

MOVETANG

TYPE:
Axis Command

SYNTAX:
MOVETANG(absolute_position, [link_axis])

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

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

MOVETANG does not get cleared from the **MTYPE** when it has completed its movement. This is so that you can use it in a tight loop which updates the end position by calling the **MOVETANG** again. When using the link_axis the end position is automatically updated from **TANG_DIRECTION** of the link axis.

PARAMETERS:

absolute_position:	The absolute position to be set as the endpoint of the move. Value must be within the range –PI to +PI in the units of the rotary axis. For example if the rotary axis has 4000 pulses/turn, the **UNITS** value=1 and the angle required is PI/2 (90 deg) the position value would be 1000.
link_axis	An optional link axis may be specified. When a link_axis is specified the system software calculates the absolute position required each servo cycle based on the link axis **TANG_DIRECTION**. The **TANG_DIRECTION** is multiplied by the **REP_DIST**/PI to calculate the required position. Note that when using a link_axis the absolute_ position parameter becomes unused. The position is copied every servo cycle until the **MOVETANG** is CANCELled.

EXAMPLES:

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

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

EXAMPLE 2:

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

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

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

EXAMPLE 3:

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

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

PROGRAM TC_SETUP.BAS:

```
‘Set up 3 axes for Tangential Control
WDOG=OFF

BASE(0)
P_GAIN=0.9
VFF_GAIN=12.85
UNITS=50  ‘set units for mm
SERVO=ON

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

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

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

‘start the tangential control routine
BASE(0,1) ‘define the pair of axes which are for X and Y
```

```
'start the tangential control
BASE(2)
MOVETANG(0, 0) 'use axes 0 and 1 as the linked pair
```

PROGRAM MOTION.BAS:

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

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

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

MOVEABS(10,10+c)
REPEAT
  MOVE(0,size)
  MOVECIRC(c,c,c,0,1)
  IF nobuf THEN WAIT IDLE:WA(2)
  MOVE(size,0)
  MOVECIRC(c,-c,0,-c,1)
  IF nobuf THEN WAIT IDLE:WA(2)
  MOVE(0,-size)
  MOVECIRC(-c,-c,-c,0,1)
  IF nobuf THEN WAIT IDLE:WA(2)
  MOVE(-size,0)
  MOVECIRC(-c,c,0,c,1)
  IF nobuf THEN WAIT IDLE:WA(2)
UNTIL FALSE
```

MPE

TYPE:

System Command

SYNTAX:

MPE(mode)

DESCRIPTION:

Sets the type of channel handshaking to be performed on the command line.

This is normally only used by the *Motion* Perfect program, but can be used for user applications with the PC*Motion* ActiveX control in asynchronous mode.

PARAMETERS:

mode:	0	No channel handshaking, XON/**XOFF** controlled by the port. When the current output channel is changed then nothing is sent to the command line. 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.
	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.
	4	As mode 1 but with extra error reporting from the *Motion Coordinator*.

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

EXAMPLE:

Use the command line to demonstrate mode 0 and 1

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

MPOS

TYPE:
Axis Parameter (Read Only)

DESCRIPTION:
This parameter is the position of the axis as measured by the encoder or resolver.

Unless using an absolute encoder **MPOS** is reset to 0 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**.

VALUE:
Actual axis position in user **UNITS**.

EXAMPLE:
```
WAIT UNTIL MPOS>=1250
SPEED=2.5
```

MSPEED

TYPE:
Axis Parameter (Read Only)

DESCRIPTION:
MSPEED can be used to represent the speed measured as it represents the change in measured position in user **UNITS** (per second) in the last servo period.

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.

VALUE:
Change in measured position per second in user **UNITS**.

EXAMPLE:
Average **MSPEED** using a filter algorithm.
```
‘ VR(10) filter output

c = 0.005 ‘filter coefficient (0<c<1)
```

```
VR(10)=MSPEED  'initialise filter output to MSPEED

WHILE TRUE
  WA(1)
  VR(10)=(1-c)*VR(10)+c*MSPEED
WEND
```

MSPHERICAL

TYPE:

Axis Command

SYNTAX:

`MSPHERICAL({parameters}, mode [, gtpi][, rotau][, rotav][, rotaw])`

DESCRIPTION:

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

PARAMETERS:

mode:	0	specify end point and mid point on curve.
	1	specify end point and centre of sphere.
	2	two mid point are specified and the curve completes a full circle.
	3	mid point on curve and centre of sphere are specified and the curve completes a full circle.
gtpi:		If this optional parameter is non zero, modes 0 and 1 will perform a move taking the opposite way around a 360 degree circle to the same endpoint.
rotau:		If this optional parameter is non zero, a 4th axis will perform linear interpolation at the same time as the spherical move. The axis is the next in the **BASE** sequence. The move distance does not affect the path length or time taken for the movement. The path length is calculated just from the spherical distance.
rotav:		If this optional parameter is non zero, a 5th axis will perform linear interpolation at the same time as the spherical move.
rotaw:		If this optional parameter is non zero, a 6th axis will perform linear interpolation at the same time as the spherical move.

If you specify the parameters for the third axis as 0 and assign it to a virtual, you can use **MSPHERICAL** to perform circular movements. This allows you to specify the arc without knowing the centre point.

MODE = 0:

SYNTAX:

```
MSPHERICAL(endx, endy, endz, midx, midy, midz, 0)
```

DESCRIPTION:

Move the three axis, set in the **BASE** array through a section of a sphere by specifying the end point and a mid point on the curve.

PARAMETERS:

endx:	End position of the first axis
endy:	End position of the second axis
endz:	End position of the third axis
midx:	Mid position of the first axis
midy:	Mid position of the second axis
midz:	Mid position of the third axis

MODE = 1:

SYNTAX:

```
MSPHERICAL(endx, endy, endz, centrex, centrey, centrez, 1)
```

DESCRIPTION:

Move the three axis, set in the **BASE** array through a section of a sphere by specifying the end point and the centre of the sphere. The profile will always go the shortest path to the endpoint, this may be clockwise or counterclockwise.

The coordinates of the centre point and end point must not be co-linear. Semi-circles cannot be defined by using mode 1 because the sphere centre would be co-linear with the endpoint. If co-linier points are specified the controller will stop the program with a **RUN_ERROR**.

PARAMETERS:

endx:	End position of the first axis
endy:	End position of the second axis
endz:	End position of the third axis
centrex:	position of the first axis
centrey:	Centre position of the second axis
centrez:	Centre position of the third axis

MODE = 2:

SYNTAX:
`MSPHERICAL(midx1, midy1, midz1, midx, midy, midz, 2)`

DESCRIPTION:
Move the three axis, set in the **BASE** array through a full circle on a sphere by specifying two mid points of the curve. The profile will move through the first mid position, then the second and finally back to the start point.

PARAMETERS:

midx1:	Second mid position of the first axis
midy1:	Second mid position of the second axis
midz1:	Second mid position of the third axis
midx:	First mid position of the first axis
midy:	First mid position of the second axis
midz:	First mid position of the third axis

MODE = 3:

SYNTAX:
`MSPHERICAL(midx, midy, midz, centrex, centrey, centrez, 3)`

DESCRIPTION:

Move the three axis, set in the **BASE** array through a full circle on a sphere by specifying a mid point and the centre of the sphere. The profile will start by heading in the shortest distance to the mid point, this enables you to define the direction.

The coordinates of the centre point and mid point must not be co-linear. If co-linier points are specified the controller will stop the program with a **RUN_ERROR**.

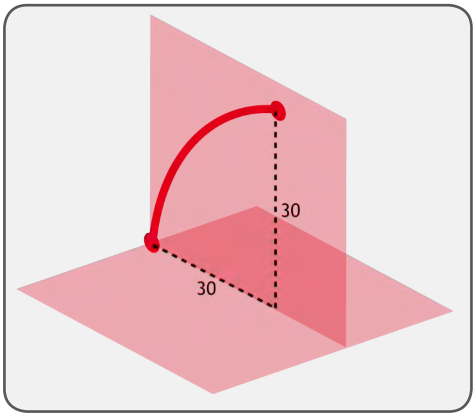
PARAMETERS:

midx:	Mid position of the first axis
midy:	Mid position of the second axis
midz:	Mid position of the third axis
centrex:	position of the first axis
centrey:	Centre position of the second axis
centrez:	Centre position of the third axis

EXAMPLES:

EXAMPLE 1:

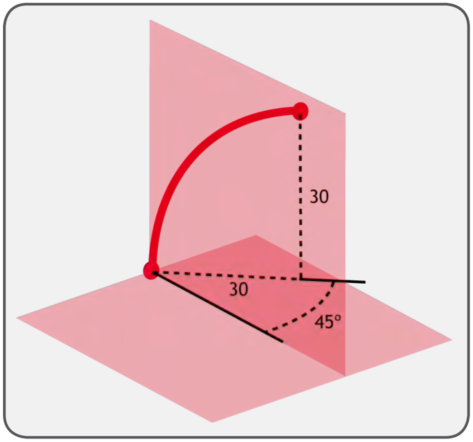
A move is needed that follows a spherical path which ends 30mm up in the Z direction:




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

EXAMPLE 2:

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




```
BASE(0,1,2)
MSPHERICAL(21.2132,21.2132,30,6.2132,6.2132,21.213
```

MSPHERICALSP

TYPE:

Axis Command

SYNTAX:

`MSPHERICAL({parameters}, mode [, gtpi][, rotau][, rotav][, rotaw])`

DESCRIPTION:

Performs a spherical move the same as **MSPHERICAL** and additionally allows vector speed to be changed when using multiple moves in the look ahead buffer when **MERGE**=ON, using additional parameters **FORCE_SPEED**, **ENDMOVE_SPEED** and **STARTMOVE_SPEED**

EXAMPLE:

A move is needed that follows a spherical path which ends 30mm up in the Z direction, the profile should decelerate from the previous move so that it is performed at 30UNITS/second:

```
BASE(3,4,5)
FORCE_SPEED=30
ENDMOVE_SPEED=30
MSPHERICALSP(30,0,30,8.7868,0,21.2132,0)
```

SEE ALSO:
MSPHERICAL

MTYPE

TYPE:
Axis Parameter (read only)

DESCRIPTION:
This parameter holds the type of move currently being executed.

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.

It takes a servo period before a motion command is loaded into the buffer, so checking **MTYPE** immediately after a motion command will probably fail. You should use **WAIT LOADED** or **WAIT IDLE** to check that a command is loaded or complete

VALUE:

Value	*Motion* command in progress
0	Idle (No move)
1	MOVE
2	MOVEABS
3	MHELICAL
4	MOVECIRC
5	MOVEMODIFY
6	MOVESP
7	MOVEABSSP
8	MOVECIRCSP
9	MHELICALSP
10	FORWARD

Value	*Motion* command in progress
11	**REVERSE**
12	**DATUM**
13	CAM
14	**FWD_JOG**
15	**REV_JOG**
20	**CAMBOX**
21	**CONNECT**
22	**MOVELINK**
23	**CONNPATH**
24	**FLEXLINK**
30	**MOVETANG**
31	**MSPHERICAL**

EXAMPLE:

Load another move if the existing move has finished

```
IF MTYPE AXIS(2) = 0 THEN
  MOVE (TABLE(count)) AXIS(2)
  count = count + 1
ENDIF
```

SEE ALSO:

WAIT

* Multiply

TYPE:

Mathematical operator

SYNTAX

<expression1> * <expression2>

DESCRIPTION:

Multiplies expression1 by expression2

PARAMETERS:

expression1:	Any valid TrioBASIC expression
expression2:	Any valid TrioBASIC expression

EXAMPLE:

Calculate the value of ‘factor’ by multiplying 10 by the sum of 2.1 and 9. the value stored in ‘factor’ will be 111.

```
factor=10*(2.1+9)
```

N_ANA_IN

TYPE:
System Parameter (read only)

ALTERNATIVE FORMAT:
NAIO

DESCRIPTION:
This parameter returns the number of analogue input channels available to the *Motion Coordinator*. This includes all built in and external inputs.

VALUE:
The number of analogue inputs

EXAMPLE:
Check the system configuration in the command line for the correct number of analogue inputs.

```
>>PRINT N_ANA_IN
10
>>
```

N_ANA_OUT

TYPE:
System Parameter (Read Only)

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

VALUE:
The number of analogue outputs

EXAMPLE:
Use the command line to check that the system has detected the correct number of analogue outputs:

```
>>PRINT N_ANA_OUT
12
>>
```

NEG_OFFSET

TYPE:
Axis Parameter

DESCRIPTION:
For Piezo Motor Control. This sets an offset to the DAC output when the position loop is demanding a negative voltage output. **NEG_OFFSET** is applied after **DAC_SCALE** so is always a value appropriate to the D to A converter resolution. The negative offset must be a negative value.

EXAMPLE:
An offset of -0.1 volts is required on an axis with a 16 bit D to A converter. With a 16 bit DAC, -10V is commanded with the value -32768 so for -0.1V need -32768 / 100.

```
NEG_OFFSET = -328
```

POS_OFFSET and **NEG_OFFSET** are normally used together. It is suggested that the offset is 65% to 70% of the value required to make the stage move in an open loop situation.

```
POS_OFFSET = 450
NEG_OFFSET = -395
```

NEW

TYPE:
System Command

SYNTAX:
NEW [item]

DESCRIPTION:
Deletes a program or table from the controller memory. If you are deleting a program from within a TrioBASIC program it is recommended to use the DEL command as makes easier to read code.

When deleting the table all the values are set to 0

Do not delete programs when connected to *Motion* Perfect as it will cause a controller mismatch and you will be disconnected.

PARAMETERS:

none	deletes the currently selected program	
item	**“TABLE”**	sets all table values to 0
	“name”	deletes a named program
	ALL	deletes all programs

Quotes (“) are required when deleting the table or a named program.

EXAMPLE:

EXAMPLE1:
Delete a named program on the command line:

```
>>NEW "NAMEDPROGRAM"
OK
>>
```

EXAMPLE 2:
Clear all table values to 0

```
>>NEW "TABLE"
OK
>>
```

SEE ALSO:
DEL

NIN

TYPE:
System Parameter

DESCRIPTION:
This parameter returns the number of inputs fitted to the system. The value is normally set by the firmware taking into consideration the total IO detected; including module IO, CAN IO, Fieldbus IO and CanOpen IO.

VALUE:
The highest input point + 1 that is in use.

EXAMPLE:

There are 24 external Output points in addition to the 16 built-in IO points on the controller. Typing ?NIN in the terminal:

```
>>?NIN
40.0000
>>
```

Note; in this case the last input point addressable is IN(39).

NIO

TYPE:

System Parameter

DESCRIPTION:

This parameter returns the number of inputs/outputs fitted to the system. The value is normally set by the firmware taking into consideration the total IO detected; including module IO, CAN IO, Fieldbus IO and CanOpen IO.

Inputs / Outputs outside of **NIO** can be used as virtual

VALUE:

The highest input / output point + 1 that is in use. If the number of Inputs is not the same as the number of Outputs then the higher count is returned in the NIO parameter.

EXAMPLE:

There are 32 external IO points in addition to the 16 built-in IO points on the controller. Typing ?NIO in the terminal:

```
>>?NIO
48.0000
>>
```

Note; in this case the last IO point addressable is IN(47) and OP(47,state)

NODE_AXIS

TYPE:

System Array (**MC_CONFIG**)

SYNTAX:

`NODE_AXIS(slot, node)= value`

DESCRIPTION:

This 2D array can be used to over-ride the drive addressing of any EtherCAT node axis. This can be used to define a user specific axis map to fix axes from different sources in place.

The array is 2-dimensional, the first dimension is the master slot identifier, the second dimension is the position of the node within that master network.

An error is raised if the axis requested is already in use when the EtherCAT protocol is started.

VALUE:

0	EtherCAT axis is allocated automatically (default)
>= 1	EtherCAT drive is located at this axis

SEE ALSO:

NODE_AXIS_COUNT, NODE_INDEX, NODE_PROFILE,

NODE_AXIS_COUNT

TYPE:

System Array (**MC_CONFIG**)

SYNTAX:

`NODE_AXIS_COUNT(slot, node)= value`

DESCRIPTION:

This 2D array can be used to set the number of axes that are located at a single EtherCAT node. This can be used to define a user specific axis map when using multi-axis drives.

The array is 2-dimensional, the first dimension is the master slot identifier, the second dimension is the position of the node within that master network.

VALUE:

1	Single axis EtherCAT node (default)

2 - n	Number of axes allocated to the EtherCAT node

SEE ALSO:
NODE_AXIS, NODE_INDEX, NODE_PROFILE,

NODE_INDEX

TYPE:
System Array (**MC_CONFIG**)

SYNTAX:
NODE_INDEX(slot, node)= value

DESCRIPTION:
This 2D array can be used to set the pointer to a block of **VR**s used by the EtherCAT node. It can be used to define a user specific Input Output map from different data sources including Boolean and Integer data within the EtherCAT node.

There is one **VR** mapped per PDO object, starting with the values from slave to master, (eg slave actual values, DIN, status word, actual position etc.) then the values from master to slave (eg slave target values, **DOUT**, control word, target position etc.)

The array is 2-dimensional, the first dimension is the master slot identifier, the second dimension is the position of the node within that master network.

VALUE:

0 to 65535	EtherCAT cyclic data is mapped to a block of **VR**s starting at this **VR** index. (MC464)
0 to 4095	EtherCAT cyclic data is mapped to a block of **VR**s starting at this **VR** index. (MC4N)

SEE ALSO:
NODE_AXIS, NODE_AXIS_COUNT, NODE_PROFILE,

NODE_IO

TYPE:
System Parameter (**MC_CONFIG**)

DESCRIPTION:
This 2D array can be used to set the start address of any EtherCAT node I/O channels. This can be used to

define a user specific IO map to fix IO points from different sources in place.

The array is 2-dimensional, the first dimension is the master slot identifier, the second dimension is the position of the node within that master network.

VALUE:

0	EtherCAT I/O allocated automatically (default)
>= 8	EtherCAT I/O is located at this IO point address

EXAMPLE:

A system with MC464, an EtherCAT module (slot 0) and a **CANIO** Module will have the following I/O assignment:

MODULEIO_BASE=0 + **DRIVEIO_BASE**=0 + **CANIO_BASE**=0

0-7	Built in inputs
8-15	Built in bi-directional I/O
16-23	Panasonic module inputs
24-39	**CANIO** bi-directional I/O
40-47	Panasonic drive inputs
48-1023	Virtual I/O

MODULEIO_BASE=-1 + **DRIVEIO_BASE**=0 + **CANIO_BASE**=0

0-7	Built in inputs
8-15	Built in bi-directional I/O
16-31	**CANIO** bi-directional I/O
32-39	Panasonic drive inputs
40-1023	Virtual I/O

MODULEIO_BASE=200 + **DRIVEIO_BASE**=0 + **CANIO_BASE**=0

0-7	Built in inputs
8-15	Built in bi-directional I/O
16-31	**CANIO** bi-directional I/O
32-39	Panasonic drive inputs

40-199	Virtual I/O
200-207	Panasonic module inputs
208-1023	Virtual I/O

SEE ALSO:
CANIO_BASE, MODULEIO_BASE, DRIVEIO_BASE, NODE_IO, MODULE_IO_MODE

NODE_PROFILE

TYPE:
System Array (**MC_CONFIG**)

SYNTAX:
NODE_PROFILE(slot, node)= value

DESCRIPTION:
This 2D array is used to set the EtherCAT profile within the internal database to use the selected profile. Each profile gives extra functionality and is vendor and product code specific. Consult the extra technical notes made available for your connected slave device.

The array is 2-dimensional, the first dimension is the master slot identifier, the second dimension is the position of the node within that master network.

VALUE:

0	Use the default node profile / configuration (default)
>= 1	Use the specified EtherCAT profile / configuration

SEE ALSO:
NODE_AXIS, NODE_INDEX, NODE_AXIS_COUNT,

NOP

TYPE:
System Parameter

DESCRIPTION:

This parameter returns the number of outputs fitted to the system. The value is normally set by the firmware taking into consideration the total IO detected; including module IO, CAN IO, Fieldbus IO and CanOpen IO.

VALUE:

The highest output point + 1 that is in use.

EXAMPLE:

There are 64 external Output points in addition to the 8 built-in IO points on the controller. Typing ?NOP in the terminal:

```
>>?NOP
80.0000
>>
```

Note; in this case the last output point addressable is OP(79,state) and **READ_OP**(79). The outputs start at OP(8,state) so the NOP value is not the total output points, it is the number at which the output map has as the highest available.

NOT

TYPE:

Logical and Bitwise functions

SYNTAX:

```
NOT expression
```

DESCRIPTION:

The NOT function truncates the number and inverts all the bits of the integer remaining.

PARAMETER:

expression:	Any valid TrioBASIC expression.

EXAMPLES:

EXAMPLE 1:

Bitwise AND 7 with NOT 1.5. This truncates 1.5 to 1 then ANDs it with 7.

```
    PRINT 7 AND NOT(1.5)
      6.0000
```

EXAMPLE 2:
If a function fails then print an error message and stop the program

```
IF NOT CAN(0,9,13,1,8,$6060,0,$02) THEN
  PRINT#user, "Failed to set velocity mode"
  STOP
ENDIF
```

<> Not Equal

TYPE:
Comparison Operator

SYNTAX:

```
<expression1> <> <expression2>
```

DESCRIPTION:
Returns **TRUE** if expression1 is not equal to expression2, otherwise returns **FALSE**.

PARAMETERS:

Expression1:	Any valid TrioBASIC expression
Expression2:	Any valid TrioBASIC expression

EXAMPLE:
Run the Scoop subroutine if axis is not idle (**MTYPE**=0 indicates axis idle)

```
IF MTYPE<>0 THEN GOTO scoop
```

NTYPE

TYPE:
Axis Parameter (Read Only)

DESCRIPTION:
This parameter holds the type of the first buffered move.

The **NTYPE** buffer can be cleared using **CANCEL**(1)

VALUE:

The numerical value of the move type

See **MTYPE** for a list of return values.

EXAMPLE:

If the first move buffer (**NTYPE**) is empty apply another move from a table

```
IF MTYPE = 0 THEN
  MOVE( TABLE(count)
  count = count +1
ENDIF
```

SEE ALSO:

MTYPE

OFF

TYPE:
Constant

DESCRIPTION:
OFF returns the value 0

EXAMPLES:

EXAMPLE 1:
Run the subroutine “tiger” if input 56 is off.

```
IF IN(56)=OFF THEN GOSUB tiger
```

EXAMPLE 2:
Turn the watchdog relay off

```
WDOG = OFF
```

OFFPOS

TYPE:
Axis Parameter

DESCRIPTION:
The **OFFPOS** parameter allows the axis position value to be offset by any amount without affecting the motion which is in progress. **OFFPOS** can therefore be used to effectively datum a system at full speed. Values loaded into the **OFFPOS** axis parameter are reset to 0 by the system software after the axis position is changed.

VALUE:
The distance to offset the current position

EXAMPLES:

EXAMPLE 1:
Change the current position by 125, using the command line terminal:

```
>>PRINT DPOS
300.0000
>>OFFPOS=125
>>PRINT DPOS
425.0000
```

```
>>
```

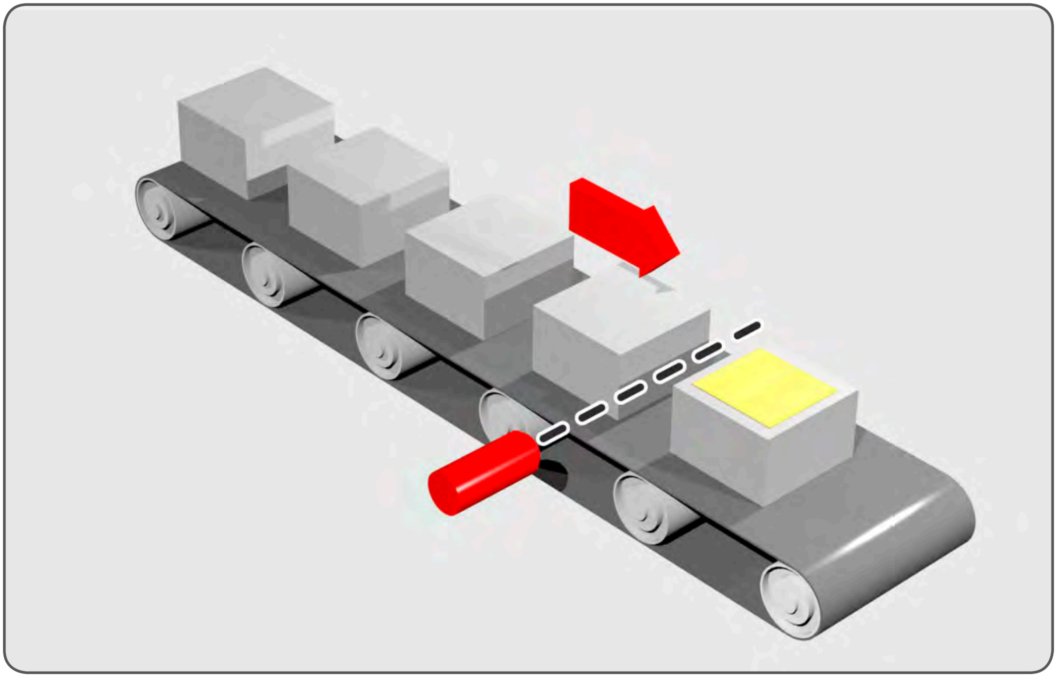
EXAMPLE 2:

Define the current demand position as zero:

```
OFFPOS=-DPOS  'This is equivalent to DEFPOS(0)
```

EXAMPLE 3:

A conveyor is used to transport boxes onto which labels must be applied.



Using the **REGIST**() function, we can capture the position at which the leading edge of the box is seen, then by using **OFFPOS** we can adjust the measured position of the axis to be zero at that point. Therefore, after the registration event has occurred, the measured position (seen in **MPOS**) will actually reflect the absolute distance from the start of the box, the mechanism which applies the label can take advantage of the absolute position start mode of the **MOVELINK** or **CAMBOX** commands to apply the label.

```
BASE(conv)
REGIST(3)
WAIT UNTIL MARK
OFFPOS = -REG_POS ' Leading edge of box is now zero
```

ON

TYPE:
Constant

DESCRIPTION:
ON returns the value 1.

EXAMPLE:
This sets the output named lever to ON.

```
OP(lever,ON)
```

ON.. GOSUB/ GOTO

TYPE:
Program Structure

SYNTAX:

```
ON expression GOxxx label[,label1[,...]]
…
label:
commands
RETURN
…
label1:
commands
RETURN
```

Where GOxxx can be **GOSUB** or **GOTO**

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. Once a label is selected it is used with either **GOSUB** or **GOTO**

If the value of the expression is less than 1 or greater than the number of labels the command is stepped through with no action. Once the label is selected a *GOSUB* is performed.

PARAMETERS:

expression:	Any valid TrioBASIC expression, should return a value 1 or greater
commands:	TrioBASIC statements that you wish to execute
label:	A valid label that occurs in the program.
GOxxx	**GOSUB** or **GOTO**

If the label does not exist an error message will be displayed at run time and the program execution halted.

EXAMPLES:

EXAMPLE 1:

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

EXAMPLE 2:

Use inputs from a PLC to determine which program to run.

```
  ON (IN(4,6)+1)GOTO prog0, prog1, prog2, prog3, prog ‘ select program
  GOTO continue ‘skip progs if unknown input selected
prog0:
  RUN “tuning”,2
  GOTO continue
prog1:
  RUN “cutting”,2
  GOTO continue
prog2:
  RUN “packing”,2
  GOTO continue
prog3:
  RUN “moving”,2
  GOTO continue
Prog4:
  RUN “lifting”,2
  GOTO continue

continue:
  …
```

SEE ALSO:

GOSUB, GOTO,

OP

TYPE:
System Command

DESCRIPTION:
Sets output(s) and allows the state of the first 32 outputs to be read back.

There are four modes of operation for the OP command, using up to three parameters:

- Read Base Block
- Write Base Block
- Set Single Output
- Write Block

MODE = READ BASE BLOCK:

SYNTAX:
```
value = OP
```

DESCRIPTION:
Return the state of the first 32 outputs as a binary pattern.

PARAMETERS:

value	Binary pattern of the first 32 outputs

MODE = WRITE BASE BLOCK:

SYNTAX:
```
OP(state)
```

DESCRIPTION:
Simultaneously set the first 32 outputs with the binary pattern of the state.

PARAMETERS:

State	Decimal equivalent of binary number to set on outputs

MODE = SET SINGLE OUTPUT:

SYNTAX:
`OP(output, state)`

DESCRIPTION:
Set the state of an individual output

PARAMETERS:

output	Output number to set.
state	0 or OFF
	1 or ON

MODE = WRITE BLOCK:

SYNTAX:
`OP(start, end, state)`

DESCRIPTION:
Simultaneously set a defined group of outputs with the binary pattern of the state.

PARAMETERS:

start	First output in the group
end	Last output in the group
state	Decimal equivalent of binary number to set on the group

EXAMPLES:

EXAMPLE 1:
Turn on a single output 44

```
OP(44,1)
```

This is equivalent to:

```
OP(44,ON)
```

EXAMPLE 2:

Sets the bit pattern 10010 on the first 5 physical outputs, outputs 13-31 will be cleared. Note how the bit pattern is shifted 8 bits by multiplying by 256 to set the first available outputs as 0 to 7 do not exist.

```
OP (18*256)
```

EXAMPLE 3:

Read the first 32 outputs, clear 0-7 as they are only inputs and 16-32. Then set 16-32 leaving 8-15 in their original state.

```
read_output:
  VR(0)=OP
  ‘clear 0-7 and 16-32
  VR(0)=VR(0) AND $0000FF00
  ‘set $1A42 in outputs 16-32,
  ‘8-15 will remain in their original state
  VR(0)=VR(0) OR $1A420000
  OP(VR(0))
```

EXAMPLE 4

Simultaneously setting outputs 10 to 13 all on.

```
OP(10,13, $F)
```

SEE ALSO:

READ_OP()

OPEN

TYPE:

Command

SYNTAX:

OPEN # channel AS “[location:]name” FOR access

DESCRIPTION:

OPEN will provide access to a text file on the controller. The text file can be initialised as a file that *Motion* Perfect can synchronise with, a temporary file, a file on the SD card or as a **FIFO** buffer. All files are in the controller file directory however only a text file can be viewed or edited in *Motion* Perfect.

Once the file has been opened then it can be manipulated by the standard TrioBASIC channel commands. If the file is opened with read access then any TrioBASIC **GET** type commands such as **GET**, **INPUT**, **LINPUT** and **KEY** can be used on the channel. If the file is opened with write access then the **PRINT** type commands can be used on the channel.

The channel should be closed using TrioBASIC command **CLOSE** when you have finished with it.

PARAMETERS:

<table>
<tr><td>channel:</td><td colspan="3">The TrioBASIC # channel to be associated with the file. It is in the range 40 to 44.</td></tr>
<tr><td rowspan="8">access:</td><td colspan="3">The operations permitted on the file.</td></tr>
<tr><td>INPUT</td><td colspan="2">The file will be opened for reading. When the end of the file is reached KEY will return FALSE, and the GET and INPUT functions will fail.</td></tr>
<tr><td rowspan="4">OUTPUT(mode)</td><td colspan="2">The file will be opened for writing. If the file does not exist then it will be created. If the file does exist then it will be cleared.</td></tr>
<tr><td>mode</td><td>function</td></tr>
<tr><td>0</td><td>Opens a text file that Motion Perfect can read, edit and save into the project.</td></tr>
<tr><td>1</td><td>Opens a temporary file that is only accessible by the controller.</td></tr>
<tr><td>FIFO_READ</td><td colspan="2">The file will be opened for reading and will be managed as a circular buffer. This is only valid for files stored in internal RAM.</td></tr>
<tr><td>FIFO_WRITE(size)</td><td colspan="2">The file will be opened for writing and will be managed as a circular buffer. This is only valid for files in internal RAM. If the file does not exist it will be created (size) bytes long.
If the file does exist then it must be of type FIFO, the size parameter is ignored and the contents are cleared.</td></tr>
<tr><td>name:</td><td colspan="3">Name of the file to be opened. The format is “[RAM|SD:]filename”. If the prefix is omitted or is RAM: then filename refers to an internal controller memory directory entry. If the prefix is SD: then filename refers to an SDCARD directory entry.</td></tr>
</table>

If you are creating a file on the SD card you will need to append the file extension. A text file stored in controller memory will be saved as a .txt file in the project by *Motion* Perfect. This enables you to generate and read files on the SD card in any text based format.

If you are writing to a text file that *Motion* Perfect can read then be aware that *Motion* Perfect will not see the changes until you perform a Project Check. Be very careful when writing to a text file while connected to Motion perfect. If it is required to write to a file while connected to Motion perfect it is recommended to use the temp file, or one on the SD card.

EXAMPLES:

EXAMPLE 1:

Open a file that can be used to log information to a .txt file on the SD card then print end of shift information to the file.

```
OPEN#40 AS “SD:product_log.txt” FOR OUTPUT (0)
PRINT#40, DATE$ ‘Print the date
PRINT#40, products_complete[0]; “ products completed”
PRINT#40, product_failures[0]; “ products failed”
CLOSE#40
```

EXAMPLE 2:

A G-Code file is loaded from a serial port into the controller, it is saved into a temp file on the controller for use later on.

```
OPEN#41 AS “gcodeprogram” for OUTPUT (1)
WHILE file_downloading
  IF KEY#1
    GET#1, char
    PRINT#41, char;
  ENDIF
  Length=length + 1
WEND
CLOSE#41
```

EXAMPLE 3:

The G-Code program has been downloaded to a temp file, it then should be transferred to a **FIFO** so that it can be interpreted into motion.

```
OPEN#41 AS “gcodeprogram” for INPUT
OPEN#42 AS “gcodefifo” for FIFO_WRITE(length)
WHILE KEY#41
  GET#41, char
  PRINT#42, char;
WEND
CLOSE#42
CLOSE#41
```

SEE ALSO:

CLOSE, GET, INPUT, LINPUT, KEY

OPEN_WIN

TYPE:

Axis Parameter

ALTERNATE FORMAT:

OW

DESCRIPTION:

This parameter defines the first position of the window which will be used for registration marks if windowing is specified by the **REGIST**() command.

VALUE:

Absolute position of the first registration window

EXAMPLE:

Enable registration but only look for registration marks between 170 and 230mm

```
OPEN_WIN=170.00
CLOSE_WIN=230.0
REGIST(256+3)
WAIT UNTIL MARK
```

SEE ALSO:

CLOSE_WIN, REGIST

OR

TYPE:

Logical and Bitwise operator

SYNTAX:

<expression1> OR <expression2>

DESCRIPTION:

This performs an OR function between corresponding bits of the integer part of two valid TrioBASIC 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

EXAMPLES:

EXAMPLE 1:

Use OR to allow the program to progress if there is a **MOTION_ERROR** or an input is pressed

```
WAIT UNTIL IN(2)=ON OR MOTION_ERROR
```

EXAMPLE 2:

Calculate the bitwise OR between values

```
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

OUTDEVICE

TYPE:

Process Parameter

DESCRIPTION:

The value in this parameter determines the default active output device. Specifying an **OUTDEVICE** for a process allows the channel number to set for all subsequent **GET**, **KEY**, **INPUT** and **LINPUT** statements.

This command is process specific so other processes will use the default channel.

This command is available for backward compatibility, it is currently recommended to use #channel, instead.

VALUE:

The channel number to use for any inputs

For a full list of communication channels see #

EXAMPLE:

Set up a program to print all data to channel 5

```
OUTDEVICE = 5

IF error THEN
  PRINT "Error Detected"
ENDIF
```

SEE ALSO:

#, GET, INPUT, KEY, LINPUT

OUTLIMIT

TYPE:

Axis Parameter

DESCRIPTION:

The output limit restricts the DAC output to a lower value than the maximum. This can be used to limit the analogue outputs or demand value to a digital drive. **OUTLIMIT** will always limit the DAC output if you are using a servo control or just manually setting DAC.

As it is applied to the output of the closed loop algorithm it is not applied to position based axis.

VALUE:

The range that the DAC is limited to

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.

EXAMPLE:

Limit a 12bit DAC to ±5V (±1023)

```
OUTLIMIT AXIS(0)=1023
```

OV_GAIN

TYPE:

Axis Parameter

DESCRIPTION:

The Output Velocity (OV) 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. 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 DPm, the contribution to the output signal is:

$$O_{ov} = K_{Ov} \times \delta P_m$$

VALUE:

Output velocity gain constant (default = 0)

Negative values are normally required.

P_GAIN

TYPE:
Axis Parameter

DESCRIPTION:
The Proportional gain sets the 'stiffness' of the servo response. Values that are too high will produce oscillation. Values that are too low will produce large following errors.

For a proportional gain K_p and position error E, its contribution to the output signal is:

$$O_p = K_p \times E$$

VALUE:
Proportional gain constant (default =1)

EXAMPLE:
Set the **P_GAIN** on axis 11 to be a value smaller than the default

```
P_GAIN AXIS(11)=0.25
```

PEEK

TYPE:
System Function

SYNTAX:

```
value = PEEK(address [,mask])
```

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

PEEK is only normally used for de-bugging purposes and should only be used under the instruction of **Trio Motion Technology**

PARAMETERS:

value:	The value returned from the memory location
address:	The memory address to read
mask:	A value so you can filter particular bits of the address

PI

TYPE:
Constant

DESCRIPTION:
PI is the circumference/diameter constant of approximately 3.14159

EXAMPLES:

EXAMPLE 1:
To print the radius of a circle of given circumference.

```
circum=100
PRINT “Radius = “;circum /(2*PI)
```

EXAMPLE 2:
Set the axis calibration to work in user **UNITS** of Radians.

```
‘Motor has 8192 counts per turn.
UNITS = 8192 / (2*PI)
```

PLC_CONFIG

TYPE:
System Parameter (**MC_CONFIG**)

DESCRIPTION:
The **PLC_CONFIG** parameter controls optional features and modes in the IEC61131-3 runtime environment. When a bit is set in the **PLC_CONFIG**, the corresponding mode of operation will be applied to all PLC tasks running in the *Motion Coordinator*.

VALUE:

Bit	Description	Value
0	PLC outputs go OFF when the PLC program is stopped.	1
	PLC outputs stay in the last state when the program is stopped.	0

Outputs may be set **ON** by a BASIC program or by the firmware (e.g. with **PSWITCH**) even when the **PLC** requests to set it **OFF** .

EXAMPLE:

In the **MC_CONFIG** script, set up the PLC system so that all outputs under PLC control will go to the OFF state whenever the program is stopped.

```
PLC_CONFIG = 1
```

Setting this bit affects the action on **STOP** or **HALT**. In the IEC61131-3 environment, not all run-time errors will stop the program. Run-time errors should be explicitly handled in a suitable exception handler.

PLC_ERROR

TYPE:

System Parameter

DESCRIPTION:

PLC_ERROR shows a bit pattern to indicate which processes in the multitasking system, which are running IEC61131-3 PLC tasks, have raised a run-time error flag. There is one bit per PLC task running in the *Motion Coordinator*.

VALUE:

Bit	Description	Value
n	The PLC task running on process *n* has a run-time error.	

EXAMPLE:

In a MC464, IEC61131-3 PLC tasks are set to run on Processes 21 and 20. In the command line terminal, check the value of **PLC_ERROR**. The IEC PLC task on process 20 has a run-time error.

```
>>?HEX(PLC_ERROR)
100000
>>
```

Checking the value in Hexadecimal shows the bit positions clearly. $100000 shows that bit 20 is set. If preferred, the value can be shown in decimal by leaving off the **HEX** modifier. In this case the value 1048576 will be returned.

PLC_OVERFLOW

TYPE:

System Parameter

DESCRIPTION:

PLC_OVERLOW can be used to check that PLC tasks are not exceeding the PLC scan time that has been set for the task. There is one bit per PLC task running in the *Motion Coordinator*.

VALUE:

Bit	**Description**	**Value**
n	PLC task running on process *n* has overflowed the configured PLC scan time.	

EXAMPLE:

An IEC61131-3 PLC task is set to run on Process 5 with a scan time of 5 msecs. In the command line terminal, check the value of **PLC_OVERFLOW**. Bit 5 is set, so the PLC task needs to be made smaller or the Scan Time must be increased.

```
>>?HEX(PLC_OVERFLOW)
20
>>
```

Checking the value in Hexadecimal shows the bit position clearly. $20 = 0010 0000 in binary. If preferred, the value can be shown in decimal by leaving off the **HEX** modifier. In this case the value 32 will be returned.

PLC_RUN

TYPE:

System Parameter

DESCRIPTION:

PLC_RUN shows a bit pattern to indicate which processes in the multitasking system are running IEC61131-3 PLC tasks. There is one bit per PLC task running in the *Motion Coordinator*.

VALUE:

Bit	Description	Value
n	A PLC task is running on process *n.*	

EXAMPLE:

IEC61131-3 PLC tasks are set to run on Processes 2, 3 and 6. In the command line terminal, check the value of **PLC_RUN**.

```
>>?HEX(PLC_RUN)
```

```
4c
>>
```

Checking the value in Hexadecimal shows the bit positions clearly. $4c = 0100 1100 in binary. If preferred, the value can be shown in decimal by leaving off the **HEX** modifier. In this case the value 76 will be returned.

PLM_OFFSET

TYPE:

Axis Parameter

DESCRIPTION:

This axis parameter is used exclusively for the SLM interface module and only in PLM (position mode). The parameter allows for an offset between the absolute position within one turn held by the SLM/PLM motor encoder and the zero position in the controller.

It is not normally required to set this parameter as it is configured during the initialisation if the **PLM**.

VALUE:

The offset between the absolute position and the controller zero position.

PMOVE

TYPE:

Process Parameter (Read Only)

DESCRIPTION:

Returns the state of the process move buffer.

When one of the 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 period 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.

VALUE:

1	the process move buffer is occupied
0	the process move buffer is empty

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.

PARAMETERS:

address:	The memory address to read
mask:	A value so you can filter particular bits of the address

PORT

TYPE:
Modifier

SYNTAX:
PORT(channel)

DESCRIPTION:
Assigns ONE command, function or port parameter operation to a particular communication **PORT**.

PARAMETERS:

channel:	The channel number to use

See the # entry for full listings of all available channels.

POS_OFFSET

TYPE:

Axis Parameter

DESCRIPTION:

For Piezo Motor Control. This sets an offset to the DAC output when the position loop is demanding a positive voltage output. **POS_OFFSET** is applied after **DAC_SCALE** so is always a value appropriate to the D to A converter resolution.

EXAMPLES:

EXAMPLE 1:

An offset of 0.1 volts is required on an axis with a 16 bit D to A converter. With a 16 bit DAC, +10V is commanded with the value 32767 so for 0.1V need 32767 / 100.

```
POS_OFFSET = 328
```

EXAMPLE 2:

POS_OFFSET and **NEG_OFFSET** are normally used together. It is suggested that the offset is 65% to 70% of the value required to make the stage move in an open loop situation.

```
POS_OFFSET = 300
NEG_OFFSET = -270
```

^ Power

TYPE:

Mathematical operator

SYNTAX:

```
<expression1> ^ <expression2>
```

DESCRIPTION:

Raises expression1 to the power of expression2

PARAMETERS:

Expression1:	Any valid TrioBASIC expression
Expression2:	Any valid TrioBASIC expression

EXAMPLE:

Raises the first number (2) to the power of the second number (6).and store it in local variable ‘x’. Then print the value of ‘x’ which is 64.

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

POWER_UP

TYPE:

Reserved Keyword

PP_STEP

TYPE:

Axis parameter

DESCRIPTION:

PP_STEP is an integer multiplier on the encoder value

UNITS and **ENCODER_RATIO** should be used in preference to **PP_STEP**

VALUE:

Integer multiplier range (default = 1)

It is recommended to only use values between -1024 and 1023

PRINT

TYPE:

Command.

ALTERNATIVE FORMAT:

?

SYNTAX:

`PRINT [#channel,] print_expression`

DESCRIPTION:

The **PRINT** command allows the TrioBASIC program to output a series of characters to a channel. A channel may be a serial port or some other type of connection to the *Motion Coordinator*.

A print_expression may include parameters, fixed **ASCII** strings, single **ASCII** characters and the returned values from functions. Multiple items to be printed can be put on the same **PRINT** line provided they are separated by a comma or semi-colon. The items can be modified using print formatters including HEX, CHR and [w,x]

Any value larger than 1e19 and smaller than 1e-18 will be printed in scientific format. You can still use [w,x] to format how this is displayed. A value is normally printed to 4 decimal places.

PARAMETERS:

#channel,	See # for the full channel list (default 0 if omitted)
print_expression:	A list of variable names (with or without print formatters) and quoted string seperated by commas and/or semicolons

The following elements may be seen in a print_expression:

;	Separates items with no space, omits carriage return line feed if used after the last item.	
,	Separates items with a tab space.	
number[w,x]	Prints a number with a specified width and number of decimal places.	
	w	total number of characters to display, 29 maximum (optional).
	x	number of decimal places to use, 15 maximum.
“string”	Prints the string contained in the quotes .	

When using value[w,x], if the number is too big the field will be filled with question marks to signify that there was not sufficient space to display the number. The numbers are right justified in the field with any unused leading characters being filled with spaces.

EXAMPLES:

EXAMPLE 1:

Print a string using quotation marks.

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

EXAMPLE 2:

Print a number and a value from a **VR**, separated by a comma to make the **VR** value in the next tab space.

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

EXAMPLE 3:

Print a **VR** with 4 characters and 1 decimal place, then in the next tab a local variable with 2 decimal places.

```
VR(1)=6
variable=410.5:
PRINT VR(1)[4,1],variable[2]
```

print output will be:

```
6.0     410.50
```

EXAMPLE 4:

Print a string directly followed by a numerical value. 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.

```
>>PRINT "DISTANCE=";MPOS
DISTANCE=123.0000
>>
```

EXAMPLE 5:

Print a carriage return and no line feed at the end of a message. 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. The string is to output from serial port channel 1.

```
PRINT #1,"ITEM ";total;" OF ";limit;CHR(13);
```

EXAMPLE 6:

Print the status of inputs 8-16 in hexadecimal format to terminal channel 5 in *Motion* Perfect.

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

EXAMPLE 7:

Print **AXISSTATUS** for axis 6 in the hexadecimal format on the command line. (bits 1 and 8 are set)

```
>>?hex(AXISSTATUS AXIS(6))
102
>>
```

SEE ALSO:
#, CHR, HEX, DATE$, DAY$, TIME$

PRMBLK

TYPE:
Reserved Keyword

PROC

TYPE:
Modifier

DESCRIPTION:
Allows a particular process to be specified when using a Process Parameter, Function or Command.

EXAMPLE:
Run a program on a particular process then watch that process to see when it finishes.

```
RUN “MOTION”,2
‘Wait for the program to start running
WAIT UNTIL PROC_STATUS PROC(2) <>0
‘Wait for the program to complete and flash an OP
REPEAT
  OP(10,ON)
  WA(100)
  OP(10,OFF)
  WA(50)
UNTIL PROC_STATUS PROC(2) = 0
```

PROC_LINE

TYPE:
Process Parameter (Read Only)

DESCRIPTION:
Allows the current line number of another executing program to be obtained.

EXAMPLE:

Find out which line is being executed on the program running in process 2.

```
>>PRINT PROC_LINE PROC(2)
12
>>
```

PROC_STATUS

TYPE:

Process Parameter (Read Only)

DESCRIPTION:

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

VALUE:

0	Process Stopped
1	Process Running
2	Process Stepping
3	Process Paused
4	Process Pausing
5	Process Stopping

EXAMPLE:

Run a program in process 12, check for it to start and then for it to complete.

```
RUN “progname”,12
WAIT UNTIL PROC_STATUS PROC(12)<>0 ‘ wait for program to start
WAIT UNTIL PROC_STATUS PROC(12)=0
‘ Program “progname” has now finished.
```

PROCESS

TYPE:

System Command (Command line only)

DESCRIPTION:

Displays information about the running processes.

There are some housekeeping process that you cannot stop.

RETURNED VALUES:

Process:	The process number
Type:	The Type of process executing
Status:	The execution state of the process
Program:	The name of the program running in the process
Line:	The line number of a program that is executing
Time:	The length of time that the process has been running
CPU:	The percentage of CPU time used by the process

EXAMPLE:

Check the state of the processes in the command line.

```
>>process
Process   Type   Status    Program     Line     hhhh:mm:ss.ms    [CPU  %]
-------   ----   ------    --------    -----    -------------    --------
21        Fast   Sleep     [0] TEST    1        0000:00:02.634   [ 0.23%]
22        SYS    Run       Command Line         0001:14:05.570   [ 0.16%]
23        SYS    Run       IO Server            0001:14:01.183   [90.46%]
24        SYS    Sleep[8]  MPE                  0001:14:05.571   [ 0.00%]
25        SYS    Sleep[6]  CAN Server           0001:14:05.571   [ 0.00%]
KERNEL    SYS    Run       Motion/Housekeeping  0001:14:05.571   [ 9.16%]
>>
```

PROCNUMBER

TYPE:

System Parameter

DESCRIPTION:

Returns the process on which a TrioBASIC program is running. This is normally required when multiple copies of a program are running on different processes.

VALUE:
The process number the current program is running on

EXAMPLE:
Running the same program on processes 0 to 3 to use axes 0-3, **PROCNUMBER** is used to specify which axis the program is using.

```
MOVE(length) AXIS(PROCNUMBER)
```

PROJECT_KEY

TYPE:
System Command

SYNTAX:
```
PROJECT_KEY key_string security_code_type
```

DESCRIPTION:
Used in the **TRIOINIT**.BAS script file on an SD card to enable loading of an encrypted project.

The project key is generated by *Motion* Perfect when encrypting a project

PARAMETERS:

key_string	A string which is the project key generated by *Motion* Perfect	
security_code_type	0 (optional)	Controller security code
	1	OEM security code
	2	User security code

EXAMPLES:

EXAMPLE 1:
Use the SD card to load a project that was previously encrypted by the *Motion* Perfect using the controller security code.

```
'============================================================
' Application: SDCARD startup file
' Filename: TRIOINIT.BAS
' Platform: MC4xx
'
```

```
' Use the Project Encryptor to generate the PROJECT_KEY which
' is specific to the target Motion Coordinator's serial number.
'
'-------------------------------------------------------------
PROJECT_KEY "MyKey"
FILE "LOAD_PROJECT" "MyEncryptedProject" 'load desired project
```

EXAMPLE 2:

Use the SD card to load a project that was previously encrypted by the *Motion* Perfect using the user security code.

```
'=============================================================
' Application: SDCARD startup file
' Filename: TRIOINIT.BAS
' Platform: MC4xx
'
' Use the Project Encryptor to generate the PROJECT_KEY which
' is specific to the target Motion Coordinator's serial number.
'
'-------------------------------------------------------------
PROJECT_KEY " c8NaHIvA.tU"2
FILE "LOAD_PROJECT" "MyEncryptedProject" 'load desired project
```

SEE ALSO:

FILE, VALIDATE_ENCRYPTION_KEY, SET_ENCRYPTION_KEY

PROTOCOL

TYPE:

Port Parameter

DESCRIPTION:

This parameter allows the user to check which protocol is running on the specified **PORT**.

You can write to this parameter however it is advisable to initialise the communication protocol through **SETCOM**, **ANYBUS** etc.

Do not write a value to **PORT**(0) as you will disable communications with *Motion* Perfect.

VALUE:

0	None
1	Download
2	MPE
3	**MODBUS**
4	Transparent
5	HostLink

EXAMPLE:

Check that Modbus is running on the RS485 channel (**PORT**(2))

```
IF PROTOCOL PORT(2) <>3 THEN
  PRINT#user, "MODBUS has stopped"
ENDIF
```

SEE ALSO:

ANYBUS, SETCOM

PS_ENCODER

TYPE:

Axis Parameter (Read Only)

DESCRIPTION:

The **PS_ENCODER** axis parameter holds a raw copy of the positional feedback device used for the hardware p-switch.

VALUE:

The 30bit value used for hardware p-switch encoder

SEE ALSO:

HW_PSWITCH

PSWITCH

TYPE:

Command

SYNTAX:

PSWITCH(switch, enable [,axis, output, state, setpos, resetpos])

PSWITCH(switch, OFF [, hold])

DESCRIPTION:

The **PSWITCH** command allows an output to be set when a predefined position is reached, and to be reset when a second position is reached. There are 64 position switches each of which can be assigned to any axis and to any output, virtual or real.

Multiple **PSWITCH**'s can be assigned to a single output.

The actual output is the **OR** of all position switches on the output **OR** the **OP** setting. This means that **OP**(output,**ON**) can override a **PSWITCH**.

When switching the **PSWITCH** *OFF*, the output will remain at the current state unless the hold parameter is set to 1. (Hold requires firmware 2.0226 or later)

PARAMETERS:

switch:	The switch number in the range 0..63	
enable:	1 or ON	Enable software **PSWITCH** (requires all parameters)
	0 or OFF	Disable **PSWITCH**
	5	Enable **PSWITCH** on **DPOS**
axis:	Axis to link the **PSWITCH** to, may be any real or virtual axis.	
output:	Selects the output to set, can be any real or virtual output.	
state:	1 or ON	turn the output ON at setpos
	0 or OFF	turn the output OFF at setpos
setpos:	The position at which output is set, in user units	
resetpos:	The position at which output is reset, in user units	

hold:	0	The **PSWITCH** output will hold in the same state it was when the **PSWITCH** is set to OFF. (Default)
	1	The **PSWITCH** output is forced OFF even if it was ON when the **PSWTICH** is set to OFF.

EXAMPLE 1:

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.

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

axis	We are told that the shaft is controlled by axis 0, thus axis is set to 0.
output	We are told that output 11 is the one to fire, so this is 11.
state	When the output is set it should be ON.
setpos	The output is to fire at 80° after TDC hence the set position is 80 as we are working in degrees.
resetpos	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.

```
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.

EXAMPLE 2:

PSWITCH number 7 has been running on axis 5 controlling output 14. It must be disabled and the output set to OFF at the same time.

```
PSWITCH(7,OFF,1)
```

Or the same **PSWITCH** must be disabled but the output not changed until some event later. The later event is controlled by a reset push button on input 23.

```
PSWITCH(7,OFF,0)
WA(1) ‘ wait one servo cycle for the PSWITCH to disable
IF READ_OP(14)=ON THEN
  WAIT UNTIL IN(23)=ON
  OP(14,OFF)
ENDIF
```

‘ Quote

TYPE:

Special Character

SYNTAX:

```
‘text
```

DESCRIPTION:

A single quote ‘ is used to mark the rest of a line as being a comment only with no execution significance.

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.

PARAMETERS:

Text	any text string

EXAMPLE:

Adding comment lines and comments after executable sections of code.

```
‘PROGRAM TO ROTATE WHEEL
turns=10
‘turns contains the number of turns required
MOVE(turns)’ the movement occurs here
```

R_MARK

TYPE:
Axis Parameter (Read Only)

SYNTAX:
`R_MARK(expression)`

DESCRIPTION:
This parameter can be polled to determine if the registration event has occurred.

This is an **AXIS** parameter, you need to ensure that you are using this parameter with the same **AXIS** that you used to set the **REGIST**.

R_MARK is reset when **REGIST** is executed

PARAMETERS:

Expression:	Any valid TrioBASIC expression. The result of the expression should be a valid integer channel number.

VALUE:

FALSE	The registration event has not occurred
TRUE	The registration event has occurred (default)
< -1	Quantity of registration events have been logged to the **TABLE**

When **TRUE** the **R_REGPOS** is valid.

EXAMPLE:
Apply an offset to the position of the axis depending on the registration position.

```
loop:
  WAIT UNTIL IN(punch_clr)=ON
  MOVE(index_length)
  REGIST(21, 1, 0, 0) ‘rising edge input channel 1
  WAIT UNTIL R_MARK(1)
  MOVEMODIFY(R_REGPOS(1) + offset)
  WAIT IDLE
GOTO loop
```

SEE ALSO:

REGIST, R_REGPOS, R_REGISTSPEED

R_REGISTSPEED

TYPE:

Axis Parameter (Read Only)

SYNTAX:

R_REGISTSPEED(expression)

DESCRIPTION:

Stores the speed of the axis when a registration mark was seen. Value is in user units per millisecond. This parameter is used with the time based registration channel set with the **REGIST** command.

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

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

This is an **AXIS** parameter, you need to ensure that you are using this parameter with the same **AXIS** that you used to set the **REGIST** so to ensure that the correct **UNITS** are used.

PARAMETERS:

Expression:	Any valid TrioBASIC expression. The result of the expression should be a valid integer channel number.

VALUE:

The speed of the axis in user units per millisecond at which the registration event occurred.

This parameter has the units of **UNITS**/msec at all **SERVO_PERIOD** settings.

EXAMPLE:

Compensate for fixed delays in the registration circuit using **R_REGISTSPEED**.

```
fixed_delays=0.012 ‘ circuit delays in milliseconds
REGIST(21, 3, 0, 0, 0) ‘ registration on time based channel 3
WAIT UNTIL R_MARK(3)
```

```
captured_position = R_REGPOS(3)-(R_REGISTSPEED(3)*fixed_delays)
```

SEE ALSO:
REGIST, REGIST_SPEED, REGIST_SPEEDB

R_REGPOS

TYPE:
Axis Parameter (Read Only)

SYNTAX:
R_REGPOS(expression)

DESCRIPTION:
Stores the latest position at which a registration mark was seen on the axis in user units. This parameter is used with the time based registration channel that was set by the **REGIST** command.

This is an **AXIS** parameter, you need to ensure that you are using this parameter with the same **AXIS** that you used to set the **REGIST** so to ensure that the correct **UNITS** are used.

PARAMETERS:

Expression:	Any valid TrioBASIC expression. The result of the expression should be a valid integer channel number.

VALUE:
The absolute position in user **UNITS** at which the registration event occurred.

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=100
  CLOSE_WIN=130
  GOSUB Initial
Loop:
  TICKS=0            'Set millisecond counter to 0
  IF R_MARK(0) THEN
    offset=R_REGPOS(0)
```

```
    '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(32,0,0,0,1) 'Allow mark to be seen between 100 and 130
  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)

SEE ALSO:

REGIST, REG_POS, REG_POSB

RAISE_ANGLE

TYPE:

Axis Parameter

DESCRIPTION:

This parameter is used with **CORNER_MODE**, it defines the maximum change in direction of a 2 axis interpolated move before **CORNER_STATE** is triggered. When the change in direction is greater than this angle **CORNER_STATE** will change state so the system can interact with a program.

This can be used to change the angle of a cutting knife

RAISE_ANGLE does not control the speed so it should be set equal or greater than **STOP_ANGLE**.

VALUE:

The angle to start to interact with a program through **CORNER_STATE**

EXAMPLE:

Decelerate to a slower speed when the transition is between 15 and 45 degrees. If the transition is greater than 45degrees stop so that a **CORNER_STATE** routine can run.

```
CORNER_MODE=2 + 4
DECEL_ANGLE = 15 * (PI/180)
STOP_ANGLE = 45 * (PI/180)
RAISE_ANGLE= STOP_ANGLE
```

SEE ALSO:
CORNER_MODE, CORNER_STATE, DECEL_ANGLE, STOP_ANGLE

.. (Range)

TYPE:
Reserved Keyword

RAPIDSTOP

TYPE:
Axis Command

SYNTAX:
RAPIDSTOP [(mode)]

ALTERNATE FORMAT:
RS

DESCRIPTION:
The **RAPIDSTOP** command cancels the currently executing move on ALL axes. Velocity profiled moves, for example; **FORWARD**, **REVERSE**, **MOVE**, **MOVEABS**, **MOVECIRC**, **MHELICAL**, **MOVEMODIFY**, will be ramped down at the programmed **DECEL** or **FASTDEC** rate then terminated. Other move types will be terminated immediately.

PARAMETERS:

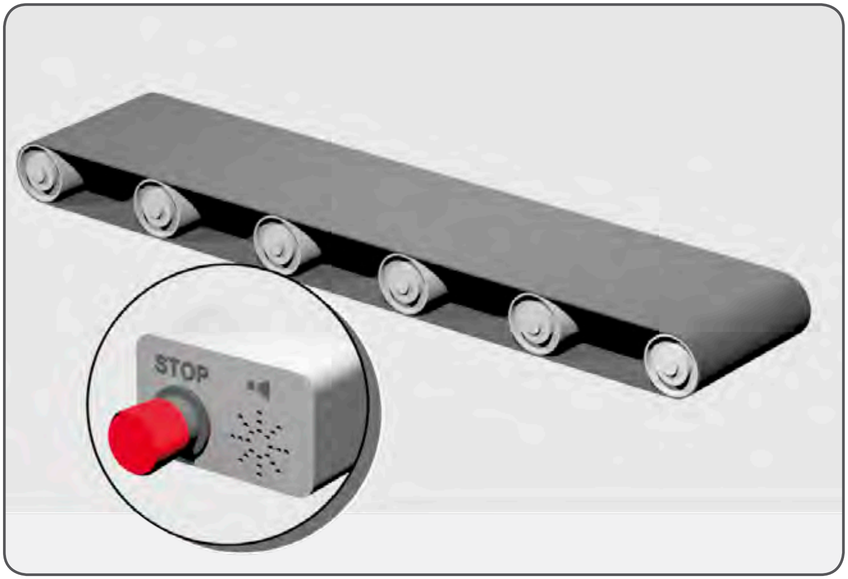
mode:	0 or none	Cancels axis commands from the **MTYPE** buffers
	1	Cancels all buffered moves on all axis (excluding the **PMOVE**)
	2	Cancels all active and buffered moves including the **PMOVE**

RAPIDSTOP will only cancel the presently executing moves. If further moves are buffered they will then be loaded and the axis will not stop.

EXAMPLES:

EXAMPLE 1:

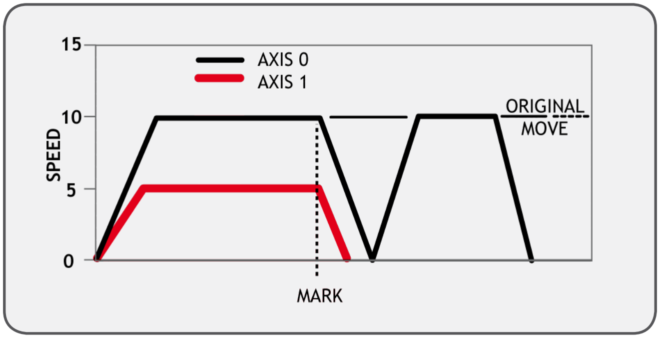
Implementing a stop override button that cuts out all motion.




```
CONNECT (1,0) AXIS(1)     'axis 1 follows axis 0
BASE(0)
REPAEAT
  MOVE(1000) AXIS (0)
  MOVE(-100000) AXIS (0)
  MOVE(100000) AXIS (0)
UNTIL IN (2)=OFF          'stop button pressed?
RAPIDSTOP(2)
```

EXAMPLE 2:

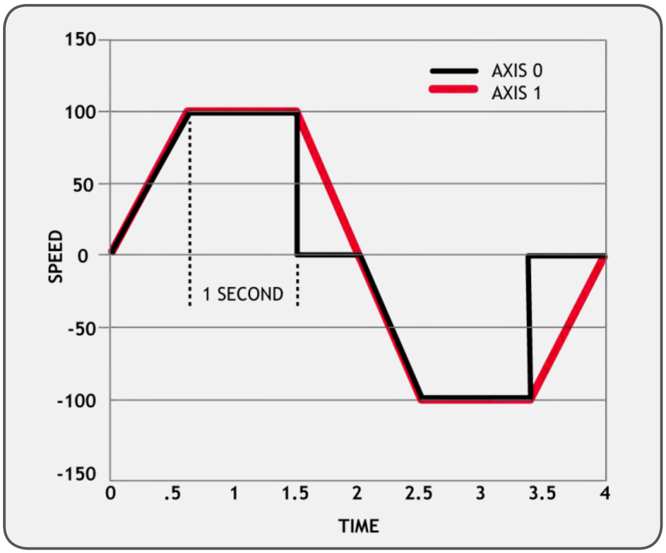
Using **RAPIDSTOP** to cancel a **MOVE** on the main axis and a **FORWARD** on the second axis. After the axes have stopped, a **MOVEABS** is applied to re-position the main axis.




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

EXAMPLE 3:

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



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

SEE ALSO:

CANCEL, FASTDEC

READ_BIT

TYPE:

Logical and Bitwise Command

SYNTAX:

READ_BIT(bit, variable)

DESCRIPTION:

READ_BIT can be used to test the value of a single bit within a **VR**() variable.

PARAMETERS:

bit:	The bit number to clear, valid range is 0 to 52
variable:	The **VR** which to operate on

EXAMPLE:

Read bit 4 of **VR**(13).

```
    Result = READ_BIT(4,13)
```

SEE ALSO:

SET_BIT, CLEAR_BIT

READ_OP

TYPE:

System Command

SYNTAX:

value = READ_OP(output [,finaloutput])

DESCRIPTION:

Returns the state of digital output logic.

If called with one parameter, it returns the state (1 or 0) of that particular output channel. If called with 2 parameters **READ_OP**() returns, in binary, the sum of the group of outputs.

READ_OP checks the state of the output logic. The output may be virtual or not powered and you will still see the logic state.

PARAMETERS:

value:	The binary pattern of the selected outputs
output:	Output to return the value of/start of output group
finaloutput:	Last output of group

The range of output to final output must not exceed 32

EXAMPLES:

EXAMPLE 1:

In this example a single output is tested:

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

EXAMPLE 2:

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

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

READPACKET

TYPE:
Command

SYNTAX:
`READPACKET(port, variable, count [,format])`

DESCRIPTION:
READPACKET is used to read in data to the **VR** variables over a serial communications port. The data is transmitted from the PC in binary format with a CRC 16bit checksum. There are four different data formats, all use the same packet structure:

Data					CRC	
Byte 0	Byte 1	Byte 2	...	Byte n	Byte 0	Byte 1

The 16bit checksum uses the generator polynomial:
$x^{16}+x^{15}+x^{2}+x^{0}$ or $8005

PARAMETERS:

port:	This value should be 0 to 2	
pariable:	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, maximum 250	
format:	The number format for the numbers being downloaded	
	0	Standard character
	1	Standard integer
	2	Standard long
	4	7bit long

Depending on the format used the data may be split over multiple bytes. It is up to the user to recombine these to get the final value.

FORMAT = 0 (STANDARD CHARACTER)
Each value is in each Byte:

Value0 = Byte 0
Value1 = Byte 1
…

FORMAT = 1 (STANDARD INTEGER)

Each value is split over 2Bytes:

Value0 = Byte1 * 256 + Byte0
Value1 = Byte3 * 256 + Byte2
…

FORMAT = 2 (STANDARD LONG)

Each value is split over 4Bytes

Value0 = ((Byte3 * 256 + Byte2) * 256 + Byte1) * 256 +Byte0
Value1 = ((Byte7 * 256 + Byte6) * 256 + Byte5) * 256 +Byte4
…

FORMAT = 4 (7BIT LONG)

Each value is split over 4Bytes, but only uses 7 bits of each byte. Only Byte 0 (including the CRC) has bit 7 set. The values sent are therefore 24bits in length.

Bits 15 and Bits 7 of the CRC are not sent and so ignored by the check.

Value0 = ((Byte3 * 128 + Byte2) * 128 + Byte1) * 128 + Byte0
Value1 = ((Byte7 * 128 + Byte6) * 128 + Byte5) * 128 + Byte4
…

EXAMPLE:

Using Standard Long (format = 2) read in the values to a sequence of **VR**'s starting at 0 from port 1. The bytes from the **READPACKET** command are stored in **VR**(100) and onwards.

```
READPACKET(1, 100, 10, 2)
FOR value = 0 to 9
  'Off set the bytes
  VR(value*4+103) = VR(value*4+103) * (2^32)
  VR(value*4+102) = VR(value*4+103) * (2^16)
  VR(value*4+101) = VR(value*4+103) * (2^8)
  VR(value)=(value*4+103)+VR(value*4+102))+VR(value*4+101))_
    +VR(value*4+100)
NEXT value
```

REG_INPUTS

TYPE:
Axis Parameter

DESCRIPTION:
Selects which of the hardware registration inputs to use for an axis. When using **REGIST** modes 3 to 17 the first input is the A channel and the second is the B.

It is recommended to use **REGIST**(20 to 22) for new projects.

On the MC464 FlexAxis the following defaults are used:

Axis	First input	Second input
0	0	4
1	1	5
2	2	6
3	3	7
4	4	0
5	5	1
6	6	2
7	7	3

VALUE:

Bits	function

3:0	Selects the first input for the axis registration	
	0000	FlexAxis Input 0
	0001	FlexAxis Input 1
	0010	FlexAxis Input 2
	0011	FlexAxis Input 3
	0100	FlexAxis Input 4
	0101	FlexAxis Input 5
	0110	FlexAxis Input 6
	0111	FlexAxis Input 7
7:4	Selects the second input for the axis registration	
	0000	FlexAxis Input 0
	0001	FlexAxis Input 1
	0010	FlexAxis Input 2
	0011	FlexAxis Input 3
	0100	FlexAxis Input 4
	0101	FlexAxis Input 5
	0110	FlexAxis Input 6
	0111	FlexAxis Input 7

EXAMPLE:

Set registration input 2 as the first inputs and 7 as the second

```
REG_INPUTS=$72
```

REG_POS

TYPE:

Axis Parameter (Read Only)

ALTERNATE FORMAT:

RPOS

DESCRIPTION:

Stores the latest position at which a registration mark was seen on each axis in user **UNITS**. This parameter is used with the first (A) hardware registration channel, or Z mark only.

VALUE:

The absolute position in user **UNITS** at which the registration event occurred.

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)

SEE ALSO:

REGIST, **REG_POSB**, **R_REGPOS**

REG_POSB

TYPE:
Axis Parameter (Read Only)

DESCRIPTION:
Stores the latest position at which a registration mark was seen on each axis in user units. This parameter is used with the second (B) hardware registration channel, or Z mark only.

VALUE:
The absolute position in user **UNITS** of where the registration event occurred.

EXAMPLE:
Detect the front and rear edges of an object on a conveyor and measure its length.

```
‘ Registration on rising edge R0 and falling edge R1
REGIST(11)
WAIT UNTIL MARK
position1 = REG_POS
WAIT UNTIL MARKB
position2 = REG_POSB

length = position2 – position1
```

SEE ALSO:
REGIST, **REG_POS**, **R_REGPOS**

REGIST

TYPE:
Axis Command

SYNTAX:
REGIST(mode [,parameters])

DESCRIPTION:
The **REGIST** command initiates a capture of an axis position when it sees a registration input or the Z mark on the encoder. Once a registration event is captured **MARK** is set and the position and speed at the event can be read back.

See the Hardware Chapter of the manual to understand which registration mode your hardware supports.

Filtering can be applied to the input as well as defining a window of where to capture.

Hardware registration captures the encoder count against the registration input in hardware

Time based registration captures the time of the registration event and interpolates the position values being sent back from the drive against it.

Although all modes are available for backwards compatibility it is recommended to use modes 20-22 for new applications. Other modes have been provided for compatibility with older products.

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

The captured registration position may be outside **REP_DIST**. You should always check the captured registration position to ensure it is within your applications usable range.

PARAMETERS:

mode:	1..4	Single channel hardware registration
	5	Reserved
	6..13	Dual channel hardware registration
	14..17	Single channel hardware registration
	20	Single channel hardware registration
	21	Single channel time based registration
	22	8 channel hardware registration
	23	Sets 2.4usec minimum pulse width
	24	Sets 0.15usec minimum pulse width (default)
	32..39	Rising edge on time based registration (use mode 21)
	64..71	Falling edge on time based registration (use mode 21)

MODE = 1..4:

SYNTAX:

REGIST(mode)

Where mode = 1..4

DESCRIPTION:

It is recommend that you use mode 20 for all new applications

Modes 1 to 4 work with the first channel or Z mark of hardware based registration.

You can add 256 or 768 to enable windowing.

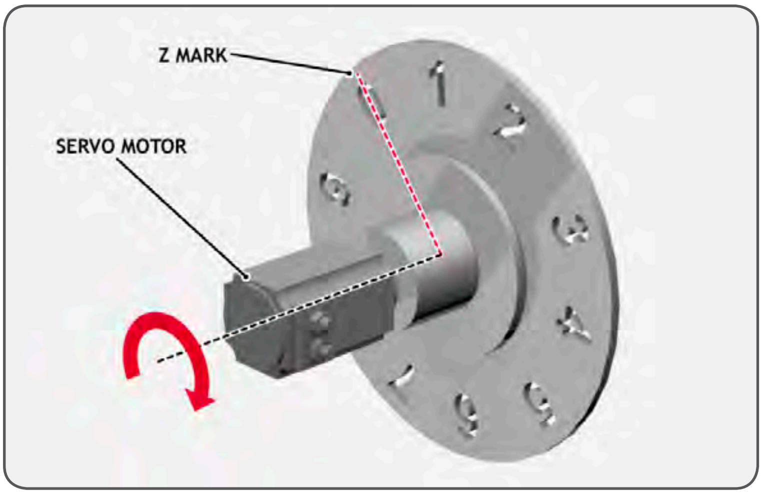
This mode works with **MARK**, **REG_POS** and **REGIST_SPEED**

PARAMETERS:

mode:	1	Z Mark rising into **REG_POS**
	2	Z Mark falling into **REG_POS**
	3	RA Input rising into **REG_POS**
	4	RA Input falling into **REG_POS**
	mode + 256	Position must be inside **OPEN_WIN**..**CLOSE_WIN**
	mode + 768	Position must be outside **OPEN_WIN**..**CLOSE_WIN**

EXAMPLE:

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



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

MODE = 6..13:

SYNTAX:

REGIST(6..13)

Where mode = 6..13

DESCRIPTION:

It is recommend that you use mode 20 for all new applications

Modes 6 to 13 work with hardware based registration but enable you to arm 2 registration registers at once.

You can add 256 or 768 to enable windowing.

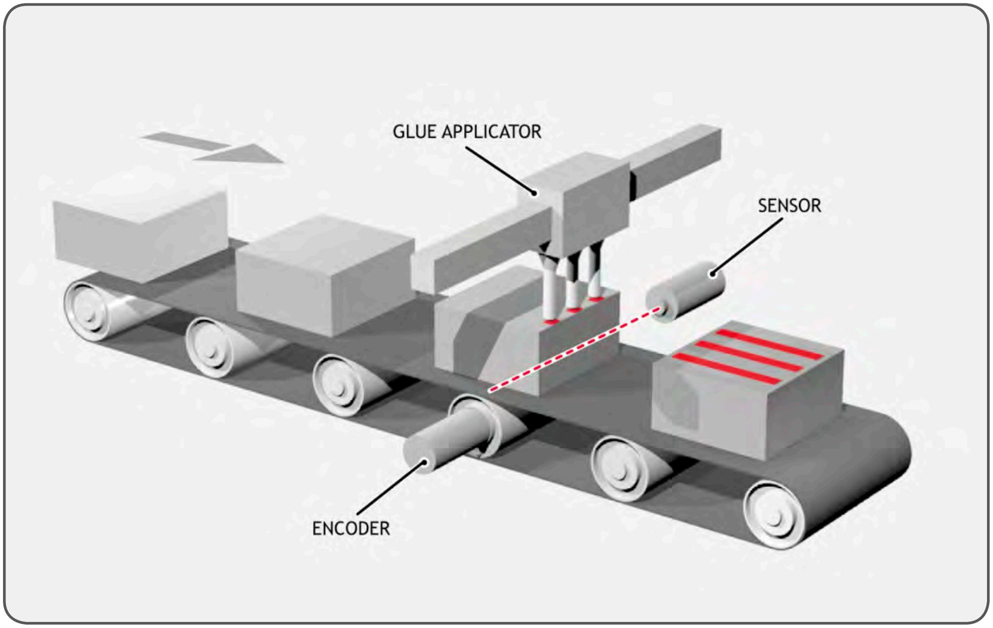
The first channel will use **MARK**, **REG_POS** and **REGIST_SPEED** and the second will use **MARKB**, **REG_POSB** and **REGIST_SPEEDB**

PARAMETERS:

mode:	6	RA Input rising into **REG_POS** & Z Mark rising into **REG_POSB**
	7	RA Input rising into **REG_POS** & Z Mark falling into **REG_POSB**
	8	RA Input falling into **REG_POS** & Z Mark rising into **REG_POSB**
	9	RA Input falling into **REG_POS** & Z Mark falling into **REG_POSB**
	10	RA Input rising into **REG_POS** & RB Input rising into **REG_POSB**
	11	RA Input rising into **REG_POS** & RB Input falling into **REG_POSB**
	12	RA Input falling into **REG_POS** & RB Input rising into **REG_POSB**
	13	RA Input falling into **REG_POS** & RB Input falling into **REG_POSB**
	mode + 256	Position must be inside **OPEN_WIN**..**CLOSE_WIN**
	mode + 768	Position must be outside **OPEN_WIN**..**CLOSE_WIN**

EXAMPLE:

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



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

MODE = 14..17:

SYNTAX:

REGIST(mode)

Where mode = 14..17

DESCRIPTION:

It is recommend that you use mode 20 for all new applications

Modes 14 to 17 work with the second channel or Z mark of hardware based registration.

You can add 256 or 768 to enable windowing.

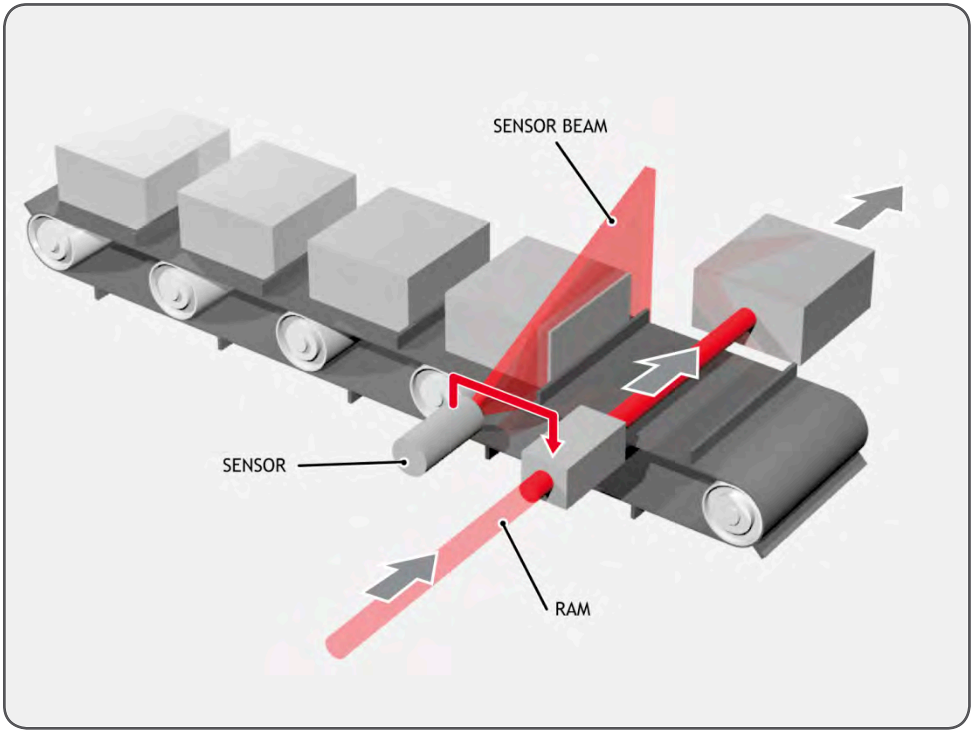
This mode works with **MARKB**, **REG_POSB** and **REGIST_SPEEDB**

PARAMETERS:

mode:	14	ZB Mark rising into **REG_POSB**
	15	ZB Mark falling into **REG_POSB**
	16	RB Input rising into **REG_POSB**
	17	RB Input falling into **REG_POSB**
	mode + 256	Position must be inside **OPEN_WIN**..**CLOSE_WIN**
	mode + 768	Position must be outside **OPEN_WIN**..**CLOSE_WIN**

EXAMPLE:

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




```
REP_DIST=120      'sets repeat distance to pitch of belt flights
REP_OPTION=ON
```

```
OPEN_WIN=30              ‘sets window open position
CLOSE_WIN=90             ‘sets window close position
REGIST(17+256)            ‘RB input registration with windowing
FORWARD                   ‘start the belt
box_seen=0
REPEAT
  WAIT UNTIL MPOS<60  ‘wait for centre point between flights
  WAIT UNTIL MPOS>60  ‘so that actuator is fired between flights
  IF box_seen=1 THEN  ‘was a box seen on the previous cycle?
    OP(8,ON)          ‘fire actuator
    WA(100)
    OP(8,OFF)         ‘retract actuator
    box_seen=0
  ENDIF
  IF MARKB THEN box_seen=1 ‘set “box seen” flag
  REGIST(17+256)
UNTIL IN(2)=OFF
CANCEL                ‘stop the belt
WAIT IDLE
```

MODE = 20:

SYNTAX:
REGIST(20, channel, source, edge, window [,quantity, table_start])

DESCRIPTION:
Mode 20 is used to set the hardware registration inputs A or B. Alternatively A or B can be replaced with the Z mark. A and B are completely independent.

When using a FlexAxis the actual input used for channel A and channel B can be selected with the **REG_INPUTS** command.

This mode can be used instead of **REGIST** modes 1..4 and 14..17

If the optional parameters quantity and table_start are used then a set of registration positions can be stored in the table. **REG_POS** and **REG_POSB** will still store the latest registration position.

PARAMETERS:

channel:	0	Selects channel A
	1	Selects channel B
	0 .. 511	Digital input selection when source set to 4
source:	0	Selects the first 24V input.
	1	Selects the Z mark.
	2	Selects the second 24V input
	3	Selects the 5V registration pin (built-in axis only)
	4	Selects any digital input as source, used on any axis
edge:	0	Rising edge
	1	Falling edge
window:	0	No windowing
	1	Position must be inside **OPEN_WIN**..**CLOSE_WIN**
	2	Position must be outside **OPEN_WIN**..**CLOSE_WIN**
quantity	1 - **TSIZE**	Quantity of registration captures to store in the **TABLE**
table_start	0 -**TSIZE**	Start position in the **TABLE** for the registration positions

If channel = 0 then *MARK*, *REG_POS* and *REGIST_SPEED* are used
If channel = 1 then *MARKB*, *REG_POSB* and *REGIST_SPEEDB* are used

If source = 4 then *MARK*, *REG_POS* and *REGIST_SPEED* are used, but only values at the nearest servo period tick are captured. (not a true hardware registration)

EXAMPLE:

Configure the windowing which will be used on channel B and then arm both channel B and the Z mark.

```
OPEN_WIN=200
CLOSE_WIN=400
REGIST(20,0,1,0,0)
REGIST(20,1,0,1,2)
```

MODE = 21:

SYNTAX:

`REGIST(21, channel, source, edge, window [,quantity, table_start])`

DESCRIPTION:

REGIST mode 21 is used to arm the time based registration.

This can be used instead of **REGIST** modes 32..39 and 64..71.

This mode operates with the parameters **R_MARK**(channel) , **R_REGPOS**(channel) and **R_REGISTSPEED**(channel).

If the optional parameters quantity and table_start are used then a set of registration positions can be stored in the table. R_ **REGPOS** will still store the latest registration position.

PARAMETERS:

channel:	This is the registration channel to be used (range 0..7)	
source:	Has no function, set to 0	
edge:	0	rising edge
	1	falling edge
window:	0	no windowing
	1	position must be inside **OPEN_WIN**..**CLOSE_WIN**
	2	position must be outside **OPEN_WIN**..**CLOSE_WIN**
quantity	1 - **TSIZE**	Quantity of registration captures to store in the **TABLE**
table_start	0 -**TSIZE**	Start position in the **TABLE** for the registration positions

MODE =22;

SYNTAX:

`REGIST(22, channel, source, edge, window [,quantity, table_start])`

DESCRIPTION:

This mode allows up to 8 hardware registration inputs to be assigned to one axis.

If this mode is used all 8 inputs are assigned to the one axis. You cannot mix **REGIST**(22) and **REGIST**(20) on one bank of inputs.

This mode operates with the parameters **R_MARK**(channel) , **R_REGPOS**(channel) and **R_**

REGISTSPEED(channel).

To use this mode **REG_INPUTS** must be set to $10 before you call the **REGIST** command.

If the optional parameters quantity and table_start are used then a set of registration positions can be stored in the table. R_ **REGPOS** will still store the latest registration position.

PARAMETERS:

channel:	This is the registration channel to be used (range 0..7)	
source:	0	Selects the 24V registration input.
	1	Selects the Z mark.
edge:	0	Rising edge
	1	falling edge
window:	0	no windowing
	1	position must be inside **OPEN_WIN**..**CLOSE_WIN**
	2	position must be outside **OPEN_WIN**..**CLOSE_WIN**
quantity	1 - **TSIZE**	Quantity of registration captures to store in the **TABLE**
table_start	0 -**TSIZE**	Start position in the **TABLE** for the registration positions

MODE = 23;

SYNTAX:
REGIST(23)

DESCRIPTION:
This mode assigns a 2.4usec minimum pulse width to the axis. This affects any **REGIST** mode that is used.

The default value is 0.15usec.

MODE = 24:

SYNTAX:
REGIST(24)

DESCRIPTION:
This mode assigns a 0.15usec minimum pulse width to the axis. This affects any **REGIST** mode that is used.

This is the default value.

SEE ALSO:
MARK, MARKB, R_MARK, REG_POS, REG_POSB, R_REGPOS, REGIST_SPEED, REGIST_SPEEDB, R_REGISTSPEED, REGIST_DELAY, REG_INPUTS

REGIST_CONTROL

TYPE:
Reserved Keyword

DESCRIPTION:
Read or set the low level bit pattern in the control register

REGIST_DELAY

TYPE:
Axis Parameter

DESCRIPTION:
The value, in milliseconds, of the total system delays between a signal appearing on the registration input and the position being available to the time-based registration algorithm. A digital system will usually transfer the actual position information with a one servo period delay. Therefore the **REGIST_DELAY** must be adjusted when the **SERVO_PERIOD** parameter is not at the default value.

In most real-world systems there are delays built into the registration circuit; the external sensor and the input opto-isolator will have some fixed response time. As machine speed increases, the fixed electrical delays will have an effect on the captured registration position. **REGIST_DELAY** can be adjusted to take account of the total delays due to the servo period and input.

VALUE:
The total registration delay in milliseconds

EXAMPLES:

EXAMPLE 1:
Compensate for fixed delay of one servo period plus 10 microseconds sensor input delay when **SERVO_PERIOD** is 1000.

```
REGIST_DELAY = -1.01
```

EXAMPLE 2:
Compensate for fixed delay of one servo period plus 15 microseconds sensor input delay when **SERVO_PERIOD** is 500.

```
REGIST_DELAY = -0.515
```

EXAMPLE 3:
Compensate for fixed delay of one servo period plus 10 microseconds sensor input delay plus one additional SLM cycle of 125 microseconds.

```
REGIST_DELAY = -1.135
```

REGIST_SPEED

TYPE:
Axis Parameter (Read Only)

DESCRIPTION:
Stores the speed of the axis when a registration mark was seen user units per milli-second. This parameter is used with the first (A) hardware registration channel, or Z mark only.

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

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

Value:

The speed of the axis in user units per milli-second at which the registration event occurred.

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

EXAMPLE:
Compensate for fixed delays in the registration circuit using **REGIST_SPEED**.

fixed_delays=0.020 ‘ circuit delays in milliseconds

```
REGIST(20, 0, 0, 0, 0)
```

```
WAIT UNTIL MARK
captured_position = REG_POS-(REGIST_SPEED*fixed_delays)
```

SEE ALSO:

REGIST, REGIST_SPEEDB, R_REGISTSPEED

REGIST_SPEEDB

TYPE:

Axis Parameter (Read Only)

DESCRIPTION:

Stores the speed of the axis when a registration mark was seen user units per milli-second. This parameter is used with the second (B) hardware registration channel, or Z mark only.

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

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

VALUE:

The speed of the axis in user units per milli-second at which the registration event occurred.

This parameter has the units of **UNITS**/msec at all **SERVO_PERIOD** settings.

SEE ALSO:

REGIST, REGIST_SPEED, R_REGISTSPEED

REMAIN

TYPE:

Axis Parameter (Read Only)

DESCRIPTION:

This is the distance, in **UNITS**, remaining to the end of the current move. It may be tested to see what amount of the move has been completed.

VALUE:
The distance remaining in user **UNITS** of the current move

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

TYPE:
System Command

SYNTAX:
REMOTE(slot)

DESCRIPTION:
Starts up the **REMOTE_PROGRAM** communication protocol as a program which communicates with PCMotion ActiveX. The **REMOTE** program will take up a user process if it is run automatically or manually. It is recommended that **REMOTE** should run on a high priority process, **REMOTE_PROC** can be set to define which process the **REMOTE_PROGRAM** runs on.

The **REMOTE** program is normally started automatically when you open a PC*Motion* connection. You can call it manually if you wish to control the starting of the process manually.

If you execute **REMOTE** manually the program it runs in will suspend at the **REMOTE** line. The **REMOTE** therefore should be the last line of the program to execute.

PARAMETERS:

slot:	0

EXAMPLE:
A program that will start the **REMOTE** program on process 20 if the project wants to run in debug mode.

```
WHILE(1)
```

```
    IF VR(debug)=TRUE THEN
      REMOTE(0)
    ELSE
      WA(100)
    ENDIF
  WEND
```

SEE ALSO:
REMOTE_PROC

REMOTE_PROC

TYPE:
System Parameter (**MC_CONFIG** / **FLASH**)

DESCRIPTION:
When the TrioPC ActiveX opens a synchronous connection to the *Motion Coordinator*, the **REMOTE_PROGRAM** is started on the highest available process. **REMOTE_PROC** can be set to specify a different process for the **REMOTE_PROGRAM**. If the defined process is in use then the next lower available process will be used.

REMOTE_PROC is stored in Flash EPROM and can also be set in the MC_**CONFIG** script file.

VALUE:

-1	Use the highest available process (default)
0 to max process	Run on defined process

EXAMPLES:

EXAMPLE1:
Set **REMOTE_PROGRAM** to start on process 19 or lower (using the command line terminal).

```
>>REMOTE_PROC=19
>>
```

EXAMPLE2:
Remove the **REMOTE_PROC** setting so that **REMOTE_PROGRAM** starts on default process (using **MC_CONFIG**).

```
'MC_CONFIG script file
REMOTE_PROC = -1  'Start on default process on connection
```

SEE ALSO:
REMOTE

RENAME

TYPE:
System Command

SYNTAX:
RENAME oldname newname

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

It is not normally used except by *Motion* Perfect.

PARAMETERS:

oldname:	The name of the program to rename.
newname:	The new name of the program.

EXAMPLE:

```
>>RENAME car voiture
OK
>>
```

REP_DIST

TYPE:
Axis Parameter

DESCRIPTION:
The repeat distance contains the allowable range of movement for an axis before the position count overflows or underflows.

When **MPOS** and **DPOS** reach **REP_DIST** they will wrap to either 0 or -**REP_DIST** depending on **REP_OPTION**. The same applies in reverse so when **MPOS** and **DPOS** reach either 0 or -**REP_DIST** they wrap to **REP_DIST**.

By default **REP_DIST** is less than the software limits. If you increase **REP_DIST** from the default value you may accidently activate **FS_LIMIT** or **RS_LIMIT**.

If a position is outside **REP_DIST** then it is adjusted by **REP_DIST** every **SERVO_PERIOD**, until the position is within **REP_DIST**. It is recommended to set the position within **REP_DIST** using **DEFPOS** or **OFFPOS** before setting **REP_DIST**.

VALUE:

The position in user units where the axis position wraps.

EXAMPLES:

EXAMPLE 1:

Units are set so that an axis units is degrees. The programmer wants to work in the range 1-360, which requires **REP_OPTION**=1.

```
REP_OPTION=1
DEFPOS(0)
REP_DIST=360
```

EXAMPLE 2:

MOVETANG requires the axis to be configures so it pi radians of the full revolution. For a 4000 count per rev encoder this means between -2000 and 2000. This can be configured as follows

```
BASE(0)
UNITS=1
DEFPOS(0)
REP_OPTION=0
REP_DIST=2000
MOVETANG(0,1)
```

SEE ALSO:

FS_LIMIT, RS_LIMIT

REP_OPTION

TYPE:

Axis Parameter

DESCRIPTION:

REP_OPTION allows different repeat options for the axis. It can be used to affect the way the position of an axis wraps or the repeating mode of **CAMBOX**, **MOVELINK** and **FLEXLINK**.

VALUE:

Bit	Description		Value
0	0	Axis position range is -**REP_DIST** to +**REP_DIST**	1
	1	Axis position range is 0 to +**REP_DIST**	
1	0	Automatic repeat option is disabled	2
	1	Disable the automatic repeat option of **CAMBOX** and **MOVELINK**	
2	0	**REP_DIST**, **DEFPOS** and **OFFPOS** will affect **MPOS** and **DPOS**	4
	1	**REP_DIST**, **DEFPOS** and **OFFPOS** will affect **MPOS** only	
3	0	**FRAME_REP_DIST** is disabled	8
	1	This mode is to be used with **FRAME** and **USER_FRAME** only and has the following functionality: **REP_DIST** is disabled **FRAME_REP_DIST** is used when **FRAME** <> 0 or **USER_FRAME** <> 0 **FRAME_REP_DIST** will only change **DPOS** and **WORLD_DPOS** **DATUM**, **DEFPOS** and **OFFPOS** only work when **FRAME** = 0 and **USER_FRAME**(0)	

Bit 2 has been included for backward compatibility, it is not recommended to use this on new applications.

EXAMPLES:

EXAMPLE 1:

An axis has 400 counts per revolution, configure **REP_DIST** and **REP_OPTION** so that it wraps from 0 to 4000.

```
REP_OPTION = 1
REP_DIST = 4000
```

EXAMPLE 2:

A program is running a continuous **MOVELINK**, when an input is triggered the link must end at the end of the next cycle. Set bit is used so not to clear any other bits that may be active.

```
MOVELINK((1, 1.6, 0.6, 0.6, 1, 4)
WAIT UNTIL IN(1) = ON
REP_OPTION = REP_OPTION AND 2
```

SEE ALSO:

CAMBOX, FRAME_REP_DIST, MOVELINK, REP_DIST

REPEAT.. UNTIL

TYPE:

Program Structure

SYNTAX:

```
REPEAT
  commands
UNTIL expression
```

DESCRIPTION:

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

The commands inside a **REPEAT**..**UNTIL** structure will always be executed at least once, if you want them to only be executed on the expression you can use a **WHILE**..**WEND**.

PARAMETERS:

expression:	Any valid TrioBASIC expression
commands:	TrioBASIC statements that you wish to execute

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.

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

RESET

TYPE:

Process Command

SYNTAX:

RESET

DESCRIPTION:
Sets the value of all the local named variables of a TrioBASIC process to 0.

EXAMPLE:
As part of an error recovery routine **RESET** can be used to clear all local variables before they are initialised again

```
WDOG=OFF
DATUM(0)  'reset error
RESET     'clear local variables
counter = 0
error_number =0
```

REV_IN

TYPE:
Axis Parameter

DESCRIPTION:
This parameter holds the input number to be used as a reverse limit input.

When the reverse limit input is active any motion on that axis is **CANCELed.**

When **REV_IN** is active **AXISSTATUS** bit 5 is set.

The input used for **REV_IN** is active low.

When the reverse limit input is active the controller will cancel the move, so the axis will decelerate at **DECEL** or **FASTDEC**.

VALUE:

-1	disable the input as **REV_IN** (default)
0-63	Input to use as the reverse input switch

Any type of input can be used, built in, Trio CAN I/O, CANopen or virtual.

EXAMPLE:
Set up inputs 8 and 9 as forward and reverse limit switches for axis 4.

```
BASE(4)
FWD_IN = 8
```

```
REV_IN = 9
```

SEE ALSO:
FWD_IN, FS_LIMIT, RS_LIMIT

REV_JOG

TYPE:
Axis Parameter

DESCRIPTION:
This parameter holds the input number to be used as a jog reverse input.

When the **REV_JOG** input is active the axis moves in reverse at **JOGSPEED**.

The input used for **REV_IN** is active low.

It is advisable to use **INVERT_IN** on the input for **REV_JOG** so that 0V at the input disables the jog.

FWD_JOG overrides **REV_JOG** if both are active

VALUE:

-1	disable the input as **REV_JOG** (default)
0-63	Input to use as datum input

EXAMPLE:
Initialise the **REV_JOG** so that it is active high on input 12

```
INVERT_IN(12,ON)
FWD_JOG=12
```

REVERSE

TYPE:
Axis Command

SYNTAX:
REVERSE

ALTERNATE FORMAT:

RE

DESCRIPTION:

Sets continuous reverse movement. The axis accelerates at the programmed **ACCEL** rate and continues moving at the **SPEED** value until either a **CANCEL** or **RAPIDSTOP** command are encountered. It then decelerates to a stop at the programmed **DECEL** rate.

If the axis reaches either the reverse limit switch or reverse soft limit, the **REVERSE** will be cancelled and the axis will decelerate to a stop.

EXAMPLES:

EXAMPLE 1:

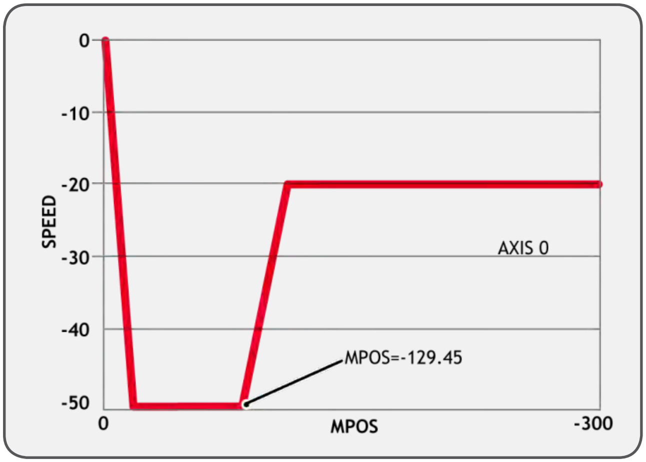
Run an axis in reverse. When an input signal is detected on input 5, stop the axis.

back:

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

EXAMPLE 2:

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

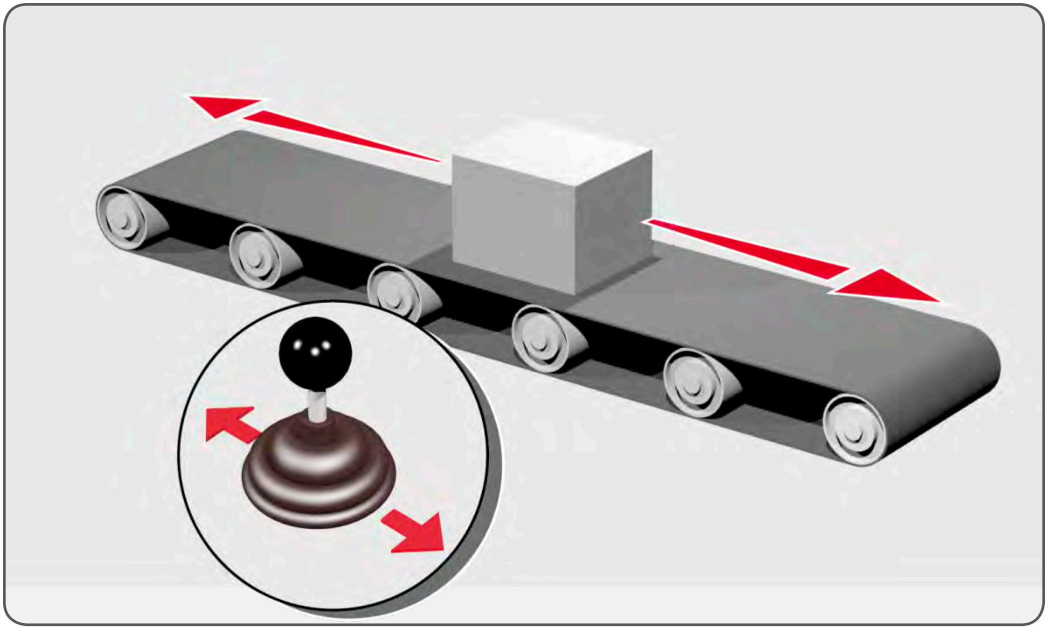



```
    DEFPOS(0)      'set starting position to zero
    REVERSE
    WAIT UNTIL MPOS<-129.45
```

```
SPEED=slow_speed
WAIT UNTIL VP_SPEED=slow_speed 'wait until the axis slows
OP(11,ON)  'turn on an output to show that speed is now slow
```

EXAMPLE 3:

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



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

SEE ALSO:

FORWARD

RIGHT

TYPE:
`STRING` Function

SYNTAX:
`RIGHT(string, length)`

DESCRIPTION:
Returns the right most section of the specified string using the length specified.

PARAMETERS:

string:	String to be used
length:	Length of string to be returned

EXAMPLES:

EXAMPLE 1:
Pre-define a variable of type string and later print its right most 10 characters:

```
DIM str1 AS STRING(32)
str1 = “TRIO MOTION TECHNOLOGY”
PRINT RIGHT(str1, 10)
```

SEE ALSO:
CHR, STR, VAL, LEN, LEFT, MID, LCASE, UCASE, INSTR

RND

TYPE:
Mathematical Function

SYNTAX:
`value = RND(<limit>)`

DESCRIPTION:
The RND function returns a random 32-bit unsigned number between 0 and (limit-1).

PARAMETERS:

limit:	Optional parameter to specify the modular math limit of the random value. The default is hex $**FFFFFFFF**
value:	The random integer number generated

EXAMPLES:

EXAMPLE 1:

Print a random 8-bit number on the command line

```
>>PRINT RND(1<<8)
173
>>PRINT RND(1<<8)
98
>>PRINT RND(1<<8)
225
>>
```

EXAMPLE 2:

Print a random number from 0 to 99 inclusive on the command line

```
>>PRINT RND(100)
61
>>PRINT RND(100)
3
>>PRINT RND(100)
40
>>
```

RS_LIMIT

TYPE:

Axis Parameter

ALTERNATE FORMAT:

RSLIMIT

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.

Bit 10 of the **AXISSTATUS** register is set when the axis position is greater than the **RS_LIMIT**.

When **DPOS** reaches **RS_LIMIT** the controller will cancel the move, so the axis will decelerate at **DECEL** or **FASTDEC**.

RS_LIMIT is disabled when it has a value greater than **REP_DIST**.

VALUE:

The absolute position of the software forward travel limit in user units. (default = 200000000000)

EXAMPLE:

After homing a machine set up the reverse software limit so that the axis will stop 10mm away from the hard stop. So if the hard limit is at -200, with a maximum speed of 400 and a **FASTDEC** of 1000 the reverse limit will be -189.6.

```
hard_limit_position = -200
max_speed = 400
FASTDEC = 1000

DATUM(3)
WAIT IDLE
RS_LIMIT= hard_limit_position + ( max_speed/FASTDEC +10 )
```

SEE ALSO:

FS_LIMIT, FWD_IN, REV_IN

RUN

TYPE:

System Command

SYNTAX:

RUN ["program" [, process]]

DESCRIPTION:

Runs a named program on the controller. Programs can be RUN from another program.

A program can be run multiple times in different processes. You can use **PROCNUMBER** to help assign values in the program.

Programs will continue to execute until there are no more lines to execute, a **HALT** is typed in the command line, a **STOP** is issued or there is a run time error.

PARAMETERS:

program:	Name of program to be run. If not present the **SELECTed** program is run
process:	Optional process number. (default highest available)

EXAMPLES:

EXAMPLE 1:

SELECT the program **STARTUP** and run it on he command line.

```
>>SELECT “STARTUP”
STARTUP selected
>>RUN%[Process 21:Program STARTUP] - Running
>>%[Process 21:Line 238] (31) - Program is stopped
>>
```

EXAMPLE 2:

From the **MAIN** program, run the **STARTUP** program on process 2 and wait for its completion:

```
RUN “STARTUP”, 2
WAIT UNTIL PROC_STATUS PROC(2) <> 0  ‘wait for program to start
WAIT UNTIL PROC_STATUS PROC(2) = 0   ‘wait for program to complete
WDOG=ON
```

EXAMPLE 3:

After **STARTUP** has completed the **MAIN** program will start other programs running in the highest available processes.

```
RUN “IO_CONTROL”
RUN “HMI”
RUN “SAUSAGE_CHOPPER”
```

SEE ALSO:

HALT , PROCNUMBER, RUN_ERROR, SELECT, STOP

RUN_ERROR

TYPE:

Process Parameter

DESCRIPTION:

Contains the number of the last run time error that stopped the program on the specified process.

RUN_ERROR = 31 is a normal completion of a program.

VALUE:

Value:	Description:
1	Command not recognized
2	Invalid transfer type
3	Error programming Flash
4	Operand expected
5	Assignment expected
6	**QUOTES** expected
7	Stack overflow
8	Too many variables
9	Divide by zero
10	Extra characters at end of line
11	] expected in **PRINT**
12	Cannot modify a special program
13	**THEN** expected in IF/**ELSEIF**
14	Error erasing Flash
15	Start of expression expected
16	) expected
17	, expected
18	Command line broken by ESC
19	Parameter out of range
20	No process available
21	Value is read only
22	Modifier not allowed
23	Remote axis is in use
24	Command is command line only
25	Command is runtime only
26	**LABEL** expected

Value:	Description:
27	Program not found
28	Duplicate Identifier
29	Program is locked
30	Program(s) running
31	Program is stopped
32	Cannot select program
33	No program selected
34	No more programs available
35	Out of memory
36	No code available to run
37	Command out of context
38	Too many nested structures
39	Structure nesting error
40	**ELSE**/**ELSEIF**/**ENDIF** without previous IF
41	**WEND** without previous **WHILE**
42	**UNTIL** without previous **REPEAT**
43	Identifier expected
44	TO expected after FOR
45	Too may nested FOR/**NEXT**
46	**NEXT** without FOR
47	**UNTIL**/**IDLE** expected after **WAIT**
48	**GOTO**/**GOSUB** expected
49	Too many nested **GOSUB**
50	**RETURN** without **GOSUB**
51	**LABEL** must be at start of line
52	Cannot nest one line IF
53	**LABEL** not found

Value:	Description:
54	**`LINE NUMBER`** cannot have decimal point
55	Cannot have multiple instances of **`REMOTE`**
56	Invalid use of $
57	VR(x) expected
58	Program already exists
59	Process already selected
60	Duplicate axes not permitted
61	PLC type is invalid
62	Evaluation error
63	Reserved keyword not available on this controller
64	**`VARIABLE`** not found
65	Table index range error
66	Features enabled do not allow **`ATYPE`** change
67	Invalid line number
68	String exceeds permitted length
69	Scope period should exceed number of Ain params
70	Value is incorrect
71	Invalid I/O channel
72	Value cannot be set. Use **`CLEAR_PARAMS`** command
73	Directory not locked
74	Directory already locked
75	Program not running on this process
76	Program not running
77	Program not paused on this process
78	Program not paused
79	Command not allowed when running *Motion* Perfect
80	Directory structure invalid

Value:	Description:
81	Directory is **LOCKED**
82	Cannot edit program
83	Too many nested **OPERANDS**
84	Cannot reset when drive servo on
85	Flash Stick Blank
86	Flash Stick not available on this controller
87	Slave error
88	Master error
89	Network timeout
90	Network protocol error
91	Global definition is different
92	Invalid program name
93	Program corrupt
94	More than one program running when trying to set **GLOBAL**/**CONSTANT**
95	Program encrypted
96	**BASIC TOKEN** definition incorrect
97	(expected
98	Number expected
99	AS expected
100	**STRING**, **VECTOR** or **ARRAY** expected
101	String expected
102	Download Abort or Timeout
103	Cannot specify program type for an existing program
104	File error: Invalid **COFF** image file
105	Variable defined outside include file
106	Command not allowed within **INCLUDE** file
107	Serial Number must be -1

Value:	Description:
108	Append block inconsistent
109	Invalid range specified
110	Too many items defined for block
111	Invalid **MSPHERICAL** input
112	Too many labels
113	Symbol table locked
114	Incorrect symbol type
115	Variables not permitted on Command Line
116	Invalid program type
117	Parameter expected
118	Firmware error: Device in use
119	Device error: Timeout waiting for device
120	Device error: Command not supported by device
121	Device error: CRC error
122	Device error: Error writing to device
123	Device error: Invalid response from device
124	Firmware error: Cannot reference data outside current block
125	Disk error: Invalid MBR
126	Disk error: Invalid boot sector
127	Disk error: Invalid sector/cluster reference
128	File error: Disk full
129	File error: File not found
130	File error: Filename already exists
131	File error: Invalid filename
132	File error: Directory full
133	Command only allowed when running *Motion* Perfect
134	# expected

Value:	Description:
135	FOR expected
136	**INPUT**/**OUTPUT**/**APPEND**/**FIFO_READ**/**FIFO_WRITE** expected
137	File not open
138	End of file
139	File already open
140	Invalid storage area
141	Numerical error: Invalid Floating-Point operation
142	Invalid System Code - wrong controller
143	IEC error: invalid variable access
144	Numerical error: Not-a-Number(NaN) used
145	Numerical error: Infinity used
146	Numerical error: Subnormal value used
147	MAC EEPROM is locked
148	Invalid mix of data types
149	Invalid startup configuration command
150	Symbol is not a variable
151	Robot Features are NOT enabled (FEC 22)
152	IEC runtime limited to 1 hour (FEC 21)
153	Command not allowed with current **ATYPE**
154	Wildcard length must be 1
155	Incompatible array dimensions
156	Matrix is singular
157	Program is not an executable type
158	Disk error: Format must be FAT32
159	Program is stopped (**HALT FORCED**)

EXAMPLE:

Use the command line to check why a program that was running on process 5 has stopped. The result of 9 indicates a divide by zero error.

```
>>? RUN_ERROR PROC(5)
9.0000
>>
```

RUNTYPE

TYPE:
System Command

SYNTAX:
RUNTYPE `“program”, mode [,process]`

DESCRIPTION:
Sets if 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. Even if they are not used.

Usually a programs **RUNTYPE** is set through *Motion* Perfect. It can be useful to set the **RUNTYPE** when loading programs from a SD card.

PARAMETERS:

program:	The program to set the power up mode.	
mode:	1	Run automatically on power up.
	0	Manual running.
process:	The process number to run the program on.	

EXAMPLE:
When loading a sequence of programs from a SD card, **MAIN** must be set to run from power up and HMI must be run on process 4 on power up. The following is from the **TRIOINIT**.bas file.

```
FILE “LOAD_PROGRAM” “MOTION”
FILE “LOAD_PROGRAM” “HMI”
FILE “LOAD_PROGRAM” “MAIN”
RUNTYPE “HMI”, 1, 4
RUNTYPE “MAIN”, 1
```

AUTORUN

S_REF

TYPE:
Axis Parameter

DESCRIPTION:
S_REF is identical to DAC.

SEE ALSO:
DAC

S_REF_OUT

TYPE:
Axis Parameter

DESCRIPTION:
S_REF_OUT is identical to **DAC_OUT**.

SEE ALSO:
DAC_OUT

SCHEDULE_OFFSET

TYPE:
System Parameter

SCHEDULE_TYPE

TYPE:
System Parameter (**MC_CONFIG** / **FLASH**)

DESCRIPTION:
This parameter changes the multi-tasking scheduling used when running programs.

Bit 0 disables the scheduling algorithm that allows another program to run while the scheduled program is in a sleep state. A sleep state can be started through a pause in the program using, for example, **WAIT** or WA.

When bit 1 is set and **SERVO_PERIOD** is 2000, the firmware doubles the number of interrupts per servo cycle. This should be used in the MC464 when **SERVO_PERIOD** is set to 2000 usec and faster communications is required. The system process can then handshake with the communications processor every millisecond.

The value is saved in Flash memory and can be included in the **MC_CONFIG** script.

VALUE:

Bit	Operation		Value
0	0	Use new scheduling algorithm to make best use of CPU time e.g. any program executing a WA command will not be available for execution again until the WA period is complete (default)	
	1	Revert to old style scheduling such that any active process will execute even when executing a WA command for example. This setting should only be used when upgrading projects from older controllers and the scheduling system causes problems with the program timings.	1
1	0	Use standard process scheduling at 2000 usec servo period.	
	1	When **SERVO_PERIOD** is set to 2000, schedule double processes. In the MC464 this enables communications like DeviceNet to run at the same rate as it does with shorter servo periods. (V2.0209 and later)	2

SCOPE

TYPE:

System Command

SYNTAX:

SCOPE(enable, [period, table_start, table_stop, p0 [,p1[,p2 [,p3 [,p4 [,p5 [,p6 [,p7]]]]]]]]])

DESCRIPTION:

The **SCOPE** command enables capture of up to 4 parameters every sample period. Samples are taken until the table range is filled. Trigger is used to start the capture.

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.

Make sure to assign the table range outside of any table data used by your programs.

It is normal to use *Motion* Perfect to assign the **SCOPE** command, but it is sometimes useful to do it manually. The table data can be read back to a PC and displayed on the *Motion* Perfect Oscilloscope, saved using *Motion* Perfect or **STICK_WRITE**.

PARAMETERS:

enable:	1 or ON	Enable software **SCOPE** (requires at least 5 parameters)
	0 or OFF	Disable **SCOPE**
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:	Second parameter to store	
p2:	Third parameter to store	
p3:	Fourth parameter to store	
p4 **p5**	Fifth parameter to store Sixth parameter to store	
p6	Seventh parameter to store	
p7	Eighth parameter to store	

EXAMPLES:

EXAMPLE 1:

This example arms the **SCOPE** to store the **MPOS** and **DPOS** on axis 5 axis 5 every 10 milliseconds (**SERVO_PERIOD** = 1000). 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.

```
SCOPE(ON,10,0,1000,MPOS AXIS(5), DPOS AXIS(5))
```

EXAMPLE 2:

Disable the **SCOPE** to prevent **TRIGGER** from starting a capture

```
SCOPE(OFF)
```

SEE ALSO:

TRIGGER

SCOPE_POS

TYPE:
System Parameter (Read Only)

DESCRIPTION:
Returns the current **TABLE** index position where the **SCOPE** function is currently storing its data.

VALUE:
The table position that is currently being used

SELECT

TYPE:
System Command

SYNTAX:
SELECT "program"

DESCRIPTION:
Makes the named program the currently selected program, if the named program does not exist then it makes a program of that name.

It is not normally used except by *Motion* Perfect.

The **SELECTed** program cannot be changed when programs are running.

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

SERCOS

TYPE:
System Function

SYNTAX:
sercos (function#,slot,{parameters})

Description:

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

Once the sercos ring is running in CP4, the standard TrioBASIC motion commands can be used.

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

The sercos command provides access to 10 separate functions:

PARAMETERS:

function:	0	Read sercos **ASIC**
	1	Write sercos **ASIC**
	2	Initialise command
	3	Link sercos drive to Axis
	4	Read parameter
	5	Write parameters
	6	Run sercos procedure command
	7	Check for dirve present
	8	Print network parameter
	9	Reserved
	10	sercos ring status
slot:	The slot number is in the range 0 to 6 and specifies the master module location.	

FUNCTION = 0:

SYNTAX:

```
sercos (0, slot, ram/reg, address)
```

DESCRIPTION:

This function reads a value from the sercos **ASIC**.

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

PARAMETERS:

slot:	The module slot in which the sercos is fitted.	
ram/reg:	0	read value from RAM
	1	read value from register.
address:	The index address in RAM or register.	

EXAMPLE:

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

FUNCTION = 1:

SYNTAX:

```
sercos (1, slot, ram/reg, address, value)
```

DESCRIPTION:

This function writes a value to the sercos **ASIC**

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

PARAMETERS:

slot:	The module slot in which the sercos is fitted.	
ram/reg:	0	write value to RAM
	1	write value to register.
address:	The index address in RAM or register.	
value:	Date to be written	

FUNCTION = 2:

SYNTAX:

`sercos (2, slot [,intensity [,baudrate [, period]]])`

DESCRIPTION:

This function initialises the parameters used for communications on the sercos ring.

PARAMETERS:

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

EXAMPLE:

```
>>SERCOS(2, 3, 4, 16, 500)
```

FUNCTION = 3:

SYNTAX:

`SERCOS(3, slot, slave_address, axis [, slave_drive_type])`

DESCRIPTION:

This function links a sercos drive (slave) to an axis.

PARAMETERS:

slot:	The module slot in which the sercos is fitted.
slave_address:	Slave address of drive to be linked to an axis.
axis:	Axis number which will be used to control this drive.

slave_drive_type:	Optional parameter to set the slave drive type. All standard sercos drives require the **GENERIC** setting. The other options below are only required when the drive is using non-standard sercos functions.	
	0	Generic Drive
	1	Sanyo-Denki
	3	Yaskawa + Trio P730
	4	PacSci
	5	Kollmorgen

EXAMPLE:

```
>> sercos (3, 1, 3, 5, 0)  'links drive at address 3 to axis 5
```

FUNCTION = 4:

SYNTAX:

```
sercos (4, slot, slave_address, parameter_ID [, parameter_size[, element_
type [, list_length_offset, [VR_start_index]]]])
```

DESCRIPTION:

This function reads a parameter value from a drive

PARAMETERS:

slot:	The module slot in which the sercos is fitted.	
slave_address:	sercos address of drive to be read.	
parameter_ID:	sercos parameter IDN	
parameter_size:	Size of parameter data expected:	
	2	2 byte parameter (default).
	4	4 byte parameter
	6	list of parameter IDs
	7	**ASCII** string

element_type:	sercos element type in the data block:	
	1	ID number
	2	Name
	3	Attribute
	4	Units
	5	Minimum Input value
	6	Maximum Input value
	7	Operational data (default)
list_length_offset:	Optional parameter to offset the list length. For drives that return 2 extra bytes, use -2.	
VR_start_index:	Beginning of **VR** array where list will be stored.	

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

EXAMPLE:

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

FUNCTION = 5:

SYNTAX:

```
sercos (5, slot , slave_address, parameter_ID, parameter_size, parameter_
value [ , parameter_value …])
```

DESCRIPTION:

This function writes one or more parameter values to a drive.

PARAMETERS:

slot:	The module slot in which the sercos is fitted.
slave_address:	sercos address of drive to be written.
parameter_ID:	sercos parameter IDN
parameter_size:	Size of parameter data to be written. 2, 4, or 6.

parameter_value:	Enter one parameter for size 2 and size 4. Enter 2 to 7 parameters for size 6 (list).

EXAMPLE:

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

FUNCTION = 6:

SYNTAX:

```
sercos (6, slot , slave_address, parameter_ID [, timeout,[command_type]])
```

DESCRIPTION:

This function runs a sercos procedure on a drive.

PARAMETERS:

<table>
<tr><td>slot:</td><td colspan="2">The communication slot in which the sercos interface is fitted.</td></tr>
<tr><td>slave_address:</td><td colspan="2">sercos address of drive.</td></tr>
<tr><td>parameter_ID:</td><td colspan="2">sercos procedure command IDN.</td></tr>
<tr><td>timeout:</td><td colspan="2">Optional time out setting (msec).</td></tr>
<tr><td rowspan="4">command_type:</td><td colspan="2">Optional parameter to define the operation:</td></tr>
<tr><td>-1</td><td>Run & cancel operation (default value)</td></tr>
<tr><td>0</td><td>Cancel command</td></tr>
<tr><td>1</td><td>Run command</td></tr>
</table>

EXAMPLE:

```
    >> sercos (6, 0, 2, 99)  'clear drive errors
```

FUNCTION = 7:

SYNTAX:

```
sercos (7 , slot , slave_address)
```

DESCRIPTION:
This function is used to detect the presence of a drive at a given sercos slave address.

PARAMETERS:

slot:	The module slot in which the sercos interface is fitted.
slave_addr:	sercos address of drive.

Returns 1 if drive detected, -1 if not detected.

EXAMPLE:

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

FUNCTION = 8:

SYNTAX:
sercos (8 , slot , required_parameter)

DESCRIPTION:
This function is used to print a sercos network parameter.

PARAMETERS:

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

EXAMPLE:

```
>>? sercos (8,0, 1 )
```

FUNCTION = 10:

SYNTAX:

```
sercos (10,<slot>)
```

DESCRIPTION:

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

PARAMETERS:

slot:	The module slot in which the sercos is fitted.

EXAMPLE:

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

SERCOS_PHASE

TYPE:

Slot Parameter

DESCRIPTION:

Sets the phase for the sercos ring in the specified slot.

VALUE:

The sercos phase, range 0-4

EXAMPLES:

EXAMPLE 1:

Set the sercos ring attached to the module in slot 0 to phase 3

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

EXAMPLE 2:

If the sercos phase is 4 in slot 2 then turn on the output

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

SERIAL_NUMBER

TYPE:
System Parameter (Read only)

DESCRIPTION:
Returns the unique Serial Number of the controller.

EXAMPLE:
For a controller with serial number 00325:

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

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 demand value dependent on the DAC parameter. When **SERVO**=ON the axis hardware will output a demand value dependant on the gain settings and the following error.

VALUE:

ON	closed loop servo control enabled
OFF	closed loop servo control disabled

EXAMPLE:
Enable axis 1 to run under closed loop control and axis 1 as open loop.

```
SERVO AXIS(0)=ON   ‘Axis 0 is under servo control
SERVO AXIS(1)=OFF  ‘Axis 1 is run open loop
```

SERVO_OFFSET

TYPE:

System Parameter (**MC_CONFIG**)

DESCRIPTION:

This parameter is a low-level scheduling parameter to allow fine tuning of when the cyclic servo activities start executing within the firmware in relation to the synchronization pulse received from controller **FPGA**.

Modification to the default settings of this parameter may be required for certain systems that require more time for data to be collected from relatively slow serial encoders for example.

SERVO_OFFSET is an **MC_CONFIG** parameter, if an entry does not exist within the **MC_CONFIG** file then default settings will be used depending upon the selected **SERVO_PERIOD** but is approximately 25% of this time period. The accepted range of values is from 0 to 75% of **SERVO_PERIOD**.

VALUE:

SERVO_OFFSET is specified in microseconds.

EXAMPLE:

```
‘ MC_CONFIG script file
SERVO_PERIOD=1000 ‘ this value is used for this cycle
SERVO_OFFSET=400  ‘ this value is used for this cycle
```

SEE ALSO:

SERVO_PERIOD

SERVO_PERIOD

TYPE:

System Parameter (**MC_CONFIG** / **FLASH**)

DESCRIPTION:

This parameter allows the controller servo period to be read or specified. This is the cycle time in which the target position updated and if applicable any positions are read and closed loop calculations performed.

SERVO_PERIOD is a flash parameter and so should be set using the **MC_CONFIG** file.

When the servo period is reduced the maximum number of axes (including virtual) is reduced as per the following table.

SERVO_PERIOD	Maximum axes
125us	8
250us	16
500us	32
1000us	64
2000us	64

VALUE:

SERVO_PERIOD is specified in microseconds. Only the values 2000, 1000, 500, 250 or 125 usec may be used and the *Motion Coordinator* must be reset before the new servo period will be applied.

The axis count will be limited as the **SERVO_PERIOD** is reduced. Normally the headline number of axes can be used when **SERVO_PERIOD** is set to 1msec.

EXAMPLES:

EXAMPLE 1:

' check controller servo_period on startup

```
IF SERVO_PERIOD<>250 THEN
  SERVO_PERIOD=250
  EX
ENDIF
```

EXAMPLE 2:

```
' MC_CONFIG script file
SERVO_PERIOD=500 ' this is the value set on power up
```

SERVO_READ

TYPE:

Axis Command

SYNTAX:

SERVO_READ(vr_start, p0[,p1[,p2[,p3[,p4[,p5[,p6[,p7]]]]]]])

DESCRIPTION:

Provides servo-synchronized access to axis/system parameters. Between 1 and 8 axis/system parameters can be read synchronously on the next servo cycle for consistent data access when required. The data read is stored in successive **VR** memory locations commencing from 'vr_start'.

The values stored are not scaled by **UNITS**.

PARAMETERS:

vr_start:	base index of **VR** memory to store data read from parameters
p0..p7:	Axis/System parameters to be read

EXAMPLE:

Read **MPOS** & FE for axes 0 & 1 and stores in **VR** locations 100,101,102 & 103.

```
SERVO_READ(100, MPOS AXIS(0), FE AXIS(0), MPOS AXIS(1), FE AXIS(1))
```

SET_BIT

TYPE:

Logical and Bitwise Command

SYNTAX:

SET_BIT(bit, variable)

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:	The bit number to set, valid range is 0 to 52
variable:	The **VR** which to operate on

EXAMPLE:

Set bit 3 of **VR**(7)

```
SET_BIT(3,7)
```

SEE ALSO:

READ_BIT, CLEAR_BIT

SET_ENCRYPTION_KEY

TYPE:
System Command

SYNTAX:
`SET_ENCRYPTION_KEY (2, fec31_password, user_security_code)`

DESCRIPTION:
SET_ENCRYPTION_KEY is used to write the user security code to the controller. The user security code is required on the controller when loading encrypted projects on that have been encrypted using the user security code method.

Motion Perfect has a tool to set the user security code

PARAMETERS:

fec31_password	The password for feature enable code 31. This can be downloaded from the E-Store or be provided by your distributor
user_security_code	Your secret user defined security code. This must be kept a secret so that other people cannot use your encrypted projects

SEE ALSO:
VALIDATE_ENCRYPTION_KEY, PROJECT_KEY

SETCOM

TYPE:
Command

SET PORT PARAMETERS:

SYNTAX:
`SETCOM(baudrate,databits,stopbits,parity,port[,mode][,variable][,timeout][,linetype])`

DESCRIPTION:
Allows the user to configure the serial port parameters and enable communication protocols.

By default the controller sets the serial ports to 38400 baud, 8 data bits, 1 stop bits and even parity.

Only one instance of Modbus **RTU** is available for the serial ports. This means that you can only run Modbus on Port 1 **OR** port 2 **NOT** both.

PARAMETERS:

baudrate:	1200, 2400, 4800, 9600, 19200, 38400 or 57600	
databits:	7 or 8	
stopbits:	1 or 2	
parity:	0	None
	1	Odd
	2	Even
port:	1, 2, 50 - 56	
mode:	0	XON/**XOFF** inactive
	1	XON/**XOFF** active
	4	**MODBUS** protocol (16 bit Integer)
	5	Hostlink Slave
	6	Hostlink Master
	7	**MODBUS** protocol (32 bit **IEEE** floating point)
	8	Reserved mode
	9	**MODBUS** protocol (32bit long word integers)
variable:	0 = Modbus uses **VR**	
	1 = Modbus uses **TABLE**	
timeout:	Communications timeout (msec). Default is 3	
linetype:	0	4 wire RS485 (Modbus only)
	1	2 wire RS485 (Modbus only)

Descriptions of the port numbers can be found under the # entry

GET PORT PARAMETERS:

SYNTAX:
`SETCOM(port)`

DESCRIPTION:
Prints the configuration of the port to the selected output channel (default terminal)

PARAMETERS:

port:	1, 2, 50 - 56

Descriptions of the port numbers can be found under the # entry

EXAMPLES:

EXAMPLE 1:
Set port 1 to 19200 baud, 7 data bits, 2 stop bits even parity and XON/**XOFF** enabled.

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

EXAMPLE 2:
Set port 2 (RS485) to 9600 baud, 8 data bits, 1 stop bit no parity and no XON/**XOFF** handshake.

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

EXAMPLE 3:
The Modbus protocol is initialised by setting the mode parameter of the **SETCOM** instruction to 4. The **ADDRESS** parameter must also be set before the Modbus protocol is activated.

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

SGN

TYPE:
Mathematical Function

SYNTAX:
`value = SGN(expression)`

DESCRIPTION:
The SGN function returns the **SIGN** of a number.

PARAMETERS:

value:	1	Positive non-zero
	0	Zero
	-1	Negative
expression:	Any valid TrioBASIC expression.	

EXAMPLE:
Detect the sign of the number -1.2 using the command line.

```
>>PRINT SGN(-1.2)
-1.0000
>>
```

<< Shift Left

TYPE:
Logical and Bitwise operator

SYNTAX:
`<expression1> << <expression2>`

DESCRIPTION:
The shift left operator, <<, can be used to logically shift left the bits in an integer variable. The value resulting from expression 1 will be shifted left by the count in expression 2. As the bits are shifted, a 0 will be inserted in the right-most bits of the value.

PARAMETERS:

Expression1:	Any valid TrioBASIC expression
Expression2:	Any valid TrioBASIC expression

EXAMPLE:
Shift the bit pattern in **VR**(23) to the left by 8, thus effecting a multiply by 256.

```
VR(23) = VR(23)<<8
```

SEE ALSO:
>>_Shift_Right

>> Shift Right

TYPE:
Logical and Bitwise operator

SYNTAX:
```
<expression1> >> <expression2>
```

DESCRIPTION:
The shift right operator, >>, can be used to logically shift right the bits in an integer variable. The value resulting from expression 1 will be shifted right by the count in expression 2. As the bits are shifted, a 0 will be inserted in the left-most bits of the value.

PARAMETERS:

Expression1:	Any valid TrioBASIC expression
Expression2:	Any valid TrioBASIC expression

EXAMPLE:
Shift the bit pattern in **AXISSTATUS** to the right by 4, thus putting the "in forward limit" bit in bit 0.
```
    result = AXISSTATUS >> 4
    in_fwd_limit = result AND 1
```

SEE ALSO:
<<_Shift_Left

SIN

TYPE:
Mathematical Function

SYNTAX:
```
value = SIN(expression)
```

DESCRIPTION:
Returns the **SINE** of an expression. This is valid for any value in expressed in radians.

PARAMETERS:

value:	The **SINE** of the expression in radians
expression:	Any valid TrioBASIC expression.

EXAMPLE:

Print the **SINE** of 0 on the command line

```
>>PRINT SIN(0)
  0.0000
>>
```

SLOT

TYPE:

Modifier

SYNTAX:

SLOT(position)

DESCRIPTION:

When expansion modules are used they are assigned a **SLOT** number depending on their position in the system. The **SLOT** modifier can be used to assign ONE command, function or slot parameter operation to a particular slot

PARAMETERS:

position:	-1	Built in feature
	0 to max_slot	Expansion module

EXAMPLE:

Check for an Anybus-CC module in the holder in slot 1

```
IF COMMSTYPE SLOT(1) = 62 THEN
  PRINT "No Anybus card present"
ENDIF
```

SEE ALSO:

COMMSPOSITION

SLOT_NUMBER

TYPE:
Axis Parameter (Read Only)

DESCRIPTION:
Returns the **SLOT** number where the axis is located. Axis numbers can be allocated to hardware in a flexible way, so the physical location of the axis cannot be found by the **AXIS** number alone. **SLOT_NUMBER** returns the value from the **BASE** axis or if the **AXIS**(number) modifier is used, it returns the **SLOT** associated with that axis.

EXAMPLE:
```
PRINT SLOT_NUMBER AXIS(12)

BASE(2)
axis2_slot = SLOT_NUMBER

IF SLOT_NUMBER AXIS(0)<>-1 THEN
  PRINT “Warning – Built-in axis configuration incorrect”
  PRINT “Axis 0 expected for this application.”
ENDIF
```

SEE ALSO:
SLOT, AXIS_OFFSET

SLOT(n)_TIME

TYPE:
Startup Parameter (**MC_CONFIG**)

DESCRIPTION:
The processor splits the time available for running system and user processes into 4 chunks. By default the system splits the available time equally into the 4 chunks, the **SLOT0_TIME, SLOT1_TIME, SLOT2_TIME** and **SLOT3_TIME** parameters allow the user to specify different percentages of the time for each slot.

Note that this is the time slots which the multitasking system uses to run the processes and nothing to do with hardware module SLOT numbers.

Out of the four slots, one is a system task only slot and so not used for user programs. The remaining are for fast and standard processes.

Slot #1: Standard task

Slot #2: Fast task

Slot #3: System process

Slot #4: Fast task

When the **SERVO_PERIOD** is 1ms or 2ms these parameters represent how the available time between consecutive servo cycles is divided into 4 slots, the total must be 100% otherwise default settings of 25% will be used.

When the **SERVO_PERIOD** is 500us SLOT0 and SLOT1 represent how the available time between consecutive servo cycles is divided into 2 slots; SLOT2 and SLOT3 represent how the available time between the next pair of consecutive servo cycles is divided into 2 slots. Both SLOT0_TIME+SLOT1_TIME and SLOT2_TIME+SLOT3_TIME must total 100% otherwise default settings of 50% will be used.

When the **SERVO_PERIOD** is less than 500us these parameters are not applicable, 100% of the available time between consecutive servo cycles is given to a single process.

Note that the minimum percentage allowed for any slot is 10%, otherwise all slots will revert to default settings.

EXAMPLES:

EXAMPLE 1 (SERVO_PERIOD=2000):

```
SLOT0_TIME=40
SLOT1_TIME=25
SLOT2_TIME=20
SLOT3_TIME=15
```

EXAMPLE 2 (SERVO_PERIOD=500):

```
SLOT0_TIME=60 'SLOT0_TIME+SLOT1_TIME=100
SLOT1_TIME=40
SLOT2_TIME=35 'SLOT2_TIME+SLOT3_TIME=100
SLOT3_TIME=65
```

EXAMPLE 3 (SERVO_PERIOD=1000):

```
SLOT0_TIME=20
SLOT1_TIME=30
SLOT2_TIME=30
SLOT3_TIME=30
```

'Invalid settings, total > 100% - default settings of 25% will be used

SEE ALSO:

SERVO_PERIOD.

SPEED

TYPE:
Axis Parameter

DESCRIPTION:
The **SPEED** axis parameter can be used to set/read back the demand speed axis parameter.

VALUE:
The axis speed in user **UNITS**

EXAMPLE:
Set the speed and then print it to the user.

```
SPEED=1000
PRINT “Speed Set=”;SPEED
```

SPEED_SIGN

TYPE:
Reserved Keyword

SPHERE_CENTRE

TYPE:
Axis Command

SYNTAX:
SPHERE_CENTRE(table_mid, table_end, table_out)

DESCRIPTION:
Returns the co-ordinates of the centre point (x, y, z) of an arc from any mid point (x, y, z) and the end point (x, y, z). X, Y and Z are returned in the **TABLE** memory area and can be printed to the terminal as required. Note that the mid and end positions are relative to the start position.

PARAMETERS:

TABLE mid:	Position in table of mid point x,y,z

TABLE end:	Position in table of end point x,y,z
TABLE out:	Position in table to store the output data: Offset 0 - X Offset 1 - Y Offset 2 - Z Offset 3 - Angle Offset 4 - Radius Offset 5 - Set to 1 if error, 0 otherwise

EXAMPLE:

```
TABLE(10,-200,400,0)
TABLE(20,-500,500,0)
SPHERE_CENTRE(10,20,30)
x = TABLE(30)
y = TABLE(31)
z = TABLE(32)
ang = TABLE(33)
rad = TABLE(34)
err = TABLE(35)
PRINT x,y,z,ang,rad,err
```

SQR

TYPE:

Mathematical Function

SYNTAX:

```
value = SQR(number)
```

DESCRIPTION:

Returns the square root of a number.

PARAMETERS:

value:	The square root of the number
number:	Any valid TrioBASIC number or variable.

EXAMPLE:

Calculate the square root of 4 using the command line.

```
>>PRINT SQR(4)
2.0000
```

```
>>
```

SRAMP

TYPE:
Axis Parameter

DESCRIPTION:
This parameter stores the s-ramp factor. It controls the amount of rounding applied to trapezoidal profiles. **SRAMP** should be set, when a move is not in progress, to a maximum of half the **ACCEL**/**DECEL** time. The setting takes a short while to be applied after changes.

VALUE:
Time between 0..250 milliseconds

SRAMP must be set before a move starts. If for example you change the **SRAMP** from 0 to 200, then start a move within 200 milliseconds the full **SRAMP** setting will not be applied.

EXAMPLE:
To provide smooth transition into the acceleration, an S-ramp is applied with a time of 50msec.

```
SPEED = 160000
ACCEL = 1600000
DECEL = 1600000
SRAMP = 50

WA(50)

MOVEABS(100000)
```

Without the S-ramp factor, the acceleration takes 100 msec to reach the set speed. With **SRAMP**=50, the acceleration takes 150 msec but the rate of change of force (torque) is controlled. i.e. Jerk is limited.

START_DIR_LAST

TYPE:
Axis Parameter (Read Only)

DESCRIPTION:
Returns the direction of the start of the last loaded interpolated motion command. **START_DIR_LAST** will be the same as **END_DIR_LAST** except in the case of circular moves.

This parameter is only available when using **SP** motion commands such as **MOVESP**, **MOVEABSSP** etc.

VALUE:
End direction, in radians between -PI and PI. Value is always positive.

EXAMPLE:
Run two moves the first starting at a direction of 45 degrees and the second 0 degrees.

```
>>MOVESP(10000,10000)
>>? START_DIR_LAST
0.7854
>>MOVESP(0,10000)
>>? START_DIR_LAST
0.0000
>>
```

SEE ALSO:
CHANGE_DIR_LAST, END_DIR_LAST

STARTMOVE_SPEED

TYPE:
Axis Parameter

DESCRIPTION:
This parameter sets the start speed for a motion command that support the advanced speed control (commands ending in SP). The **VP_SPEED** will decelerate until **STARTMOVE_SPEED** is reached for the start of the motion command.

The lowest value of **SPEED**, **ENDMOVE_SPEED**, **FORCE_SPEED** or **STARTMOVE_SPEED** will take priority.

STARTMOVE_SPEED is loaded into the buffer at the same time as the move so you can set different speeds for subsequent moves.

In general **STARTMOVE_SPEED** is only used by the **CORNER_MODE** methods. The user can program all profiles using only **FORCE_SPEED** and **ENDMOVE_SPEED**.

VALUE:

The speed at which the SP motion command will start, in user **UNITS**. (default 0)

SEE ALSO:

FORCE_SPEED, ENDMOVE_SPEED, CORNER_MODE

STEP_RATIO

TYPE:

Axis Command

SYNTAX:

STEP_RATIO(output_count, dpos_count)

DESCRIPTION:

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

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

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

PARAMETERS:

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

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

EXAMPLES:

EXAMPLE 1:

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

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

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

EXAMPLE 2:

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

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

EXAMPLE 3:

Remove the step ratio from an axis.

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

STEPLINE

TYPE:

System Command

SYNTAX:

STEPLINE ["program" ,[process]]

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.

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.

PARAMETERS:

program:	This specifies the program to be stepped.
process:	Specifies the process number.

EXAMPLE:

Start the program conveyor running in the highest available process by stepping into the first executable line.

```
>>STEPLINE “conveyor”
OK
%[Process 21:Line 19] – Paused
>>
```

STICK_READ

TYPE:

System Function

SYNTAX:

`value = STICK_READ(flash_file, table_start [,format])`

DESCRIPTION:

Read table data from the SD card to the controller.

Any existing **TABLE** data will be overwritten.

The Binary format gives the best data precision.

PARAMETERS:

value:	**TRUE** = the function was successful **FALSE** = the function was not successful
flash_file:	A number which when appended to the characters “SD” will form the data filename.
table_start:	The start point in the **TABLE** where the data values will be transferred to.
format:	0 = Binary 64bit floating point format, BIN file (default) 1 = **ASCII** comma separated values, CSV file

When storing in format=0 the data is stored in **IEEE** floating point binary format little-endian, i.e. the least significant byte first.

EXAMPLE:

Read the **ASCII** CSV file SD001984.csv from the SD card and copy the data to the table memory starting at **TABLE**(16500)

```
success = STICK_READ (1984, 16500, 1)
IF success=TRUE THEN
  PRINT #5,"SD card read OK"
ENDIF
```

SEE ALSO:

STICK_READVR

STICK_READVR

TYPE:

System Function

SYNTAX:

value = STICK_READVR(flash_file, vr_start [,format])

DESCRIPTION:

Read **VR** data from the SD card to the controller.

Any existing **VR** data will be overwritten.

The Binary format gives the best data precision.

PARAMETERS:

value:	**TRUE** = the function was successful **FALSE** = the function was not successful
flash_file:	A number which when appended to the characters "SD" will form the data filename.
vr_start:	The start point in the **VR**s where the data values will be transferred to.
format:	0 = Binary 64bit floating point format, BIN file (default) 1 = **ASCII** comma separated values, CSV file

When storing in format=0 the data is stored in **IEEE** floating point binary format little-endian, i.e. the least significant byte first.

EXAMPLE:

Read the binary file SD002012.bin from the SD card and copy the data to the **VR** memory starting at **VR**(101)

```
success = STICK_READVR(2012, 101, 0)
IF success=TRUE THEN
  PRINT #5,"SD card read OK"
ENDIF
```

SEE ALSO:

STICK_READ

STICK_WRITE

TYPE:

System Function

SYNTAX:

`value = STICK_WRITE(flash_file, table_start [,length [,format]])`

DESCRIPTION:

Used to store table data to the SD card in one of two formats.

If this file already exists, it is overwritten.

If you want to store the data without losing any precision use the Binary format.

PARAMETERS:

value:	**TRUE** = the function was successful **FALSE** = the function was not successful
flash_file:	A number which when appended to the characters "SD" will form the data filename.
table_start:	The start point in the **TABLE** where the data values will be transferred from.
length:	The number of the table values to be transferred (default 128 values)

format:	0 = Binary 64bit floating point format, BIN file (default) 1 = **ASCII** comma separated values, CSV file

When storing in format=0 the data is stored in **IEEE** floating point binary format little-endian, i.e. the least significant byte first.

EXAMPLE:

Transfer 2000 values starting at **TABLE**(1000) to the SD Card file ‘called SD1501.BIN

```
success = STICK_WRITE (1501, 1000, 2000, 0)
```

SEE ALSO:

STICK_WRITEVR

STICK_WRITEVR

TYPE:

System Function

SYNTAX:

```
value = STICK_WRITEVR(flash_file, vr_start [,length [,format]])
```

DESCRIPTION:

Used to store **VR** data to the SD card in one of two formats.

If this file already exists, it is overwritten.

If you want to store the data without losing any precision use the Binary format.

PARAMETERS:

value:	**TRUE** = the function was successful **FALSE** = the function was not successful
flash_file:	A number which when appended to the characters “SD” will form the data filename.
vr_start:	The start point in the **VR**s where the data values will be transferred from.
length:	The number of the **VR** values to be transferred (default 128 values)

format:	0 = Binary 64bit floating point format, BIN file (default) 1 = **ASCII** comma separated values, CSV file

When storing in format=0 the data is stored in **IEEE** floating point binary format little-endian, i.e. the least significant byte first.

EXAMPLE:

Transfer 2000 values starting at **VR**(1000) to the SD Card file ‘called SD1501.BIN

```
success = STICK_WRITEVR (1501, 1000, 2000, 0)
```

SEE ALSO:

STICK_WRITE

STOP

TYPE:

Command

SYNTAX:

STOP “progname”,[process_number]

DESCRIPTION:

Stops one program at its current line. A particular program name may be specified and an optional process number. The process number is required if there is more than one instance of the program running. If no name or process number is included then the selected program will be assumed.

PARAMETERS:

Progname:	name of program to be stopped.
process_number:	optional process number to be used when multiple instances of the program are running and only one is to be stopped.

EXAMPLES:

EXAMPLE 1:

Stop a program called “axis_init” from the command line. Note that quotes are optional unless the program name is also a **BASIC** keyword.

```
>>STOP axis_init
```

EXAMPLE 2:

Stop the named programs when a digital input goes off.

```
IF IN(12)=OFF THEN
  STOP “hmi_handler”
  STOP “motion1”
ENDIF
```

EXAMPLE 3:

Stop one instance of a named program and leave the other instances running.

```
proc_a = VR(45) ‘ process to be stopped is put in the VR by an HMI
STOP “test_program”,proc_a ‘ stop the required instance of test_program
```

SEE ALSO:

SELECT, RUN

STOP_ANGLE

TYPE:

Axis Parameter

DESCRIPTION:

This parameter is used with **CORNER_MODE**, it defines the maximum change in direction of a 2 axis interpolated move that will be merged at speed. When the change in direction is greater than this angle the reduced to 0.

VALUE:

The angle to reduce the speed to 0, in radians

EXAMPLE:

Reduce the speed to zero on a transition greater than 25 degrees. **DECEL_ANGLE** is set to 25 degrees as well so that there is no reduction of speed below 25 degrees.

```
CORNER_MODE=2
STOP_ANGLE=25 * (PI/180)
DECEL_ANGLE=STOP_ANGLE
```

SEE ALSO:

CORNER_MODE, DECEL_ANGLE

STORE

TYPE:
System Command

DESCRIPTION:
Used by *Motion* Perfect to load Firmware to the controller.

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

STR

TYPE:
STRING Function

SYNTAX:
STR(value[,precision[,width]])

DESCRIPTION:
Converts a numerical value to a string.

PARAMETERS:

value:	Floating-point value to be converted
precision:	Number of decimal places to be used (default=5)
width:	Width of field to be used (default=0, unlimited)

EXAMPLES:

EXAMPLE 1:
Pre-define a variable of type string and use it to store the string conversion of a **VR** variable:

```
DIM str1 AS STRING(20)
Str1 = STR(VR(100))
```

SEE ALSO:
CHR, VAL, LEN, LEFT, RIGHT, MID, LCASE, UCASE, INSTR

STRTOD

TYPE:
String Function

SYNTAX:
STRTOD(format, ...)

DESCRIPTION:
The **STRTOD** command reads a sequence of characters and converts them to a numeric value. The conversion stops at the first non-number character found in the input. The characters may be read from the **VR** array or from a TrioBASIC IO channel.

PARAMETERS:
format:

This is a bitwise field that specifies the data source and the number format.

format:	description:	value:
bit 0	Source	0 = **VR** array 1 = TrioBASIC IO channel
bit 1..2	Number format	0 = Floating point 1 = Integer. If the number is not an integer then 0 is returned. 2 = The format is auto-selected to provide the best resolution.

SOURCE = 0:

SYNTAX:
value=**STRTOD**(format, vr_start, vr_index)

DESCRIPTION:
Converts characters in the **VR** array to a number.

PARAMETERS:

Parameter:	Description:
vr_start	Position of the first character of the numeric string in the **VR** array.
vr_index	Position in the **VR** array to store the index of the first non-number character found.

SOURCE = 1:

SYNTAX:

value=**STRTOD**(format, channel, vr_length, vr_index)

DESCRIPTION:

Converts characters from the TrioBASIC channel to a number.

PARAMETERS:

Parameter:	Description:
channel	TrioBASIC IO channel to read. This can be any valid TrioBASIC IO channel: standard communications channel, **ANYBUS** channel, or file channel.
vr_length	Position in the **VR** array to store the length of the number string that was parsed.
vr_index	Position in the **VR** array to store the index of the first non-number character found.

EXAMPLE 1:

```
>>OPEN #40 AS “n” FOR OUTPUT(1)
>>PRINT #40,”123.456”
>>CLOSE #40
>>OPEN #40 AS “n” FOR INPUT
>>VR(100)=STRTOD(1,40,101,102)
>>PRINT VR(100),VR(101),VR(102)
123.4560     7.0000     13.0000
>>CLOSE #40
>>DEL “N”
```

EXAMPLE 2:

```
>>OPEN #40 AS “n” FOR OUTPUT(1)
>>PRINT #40,”123.456”
>>CLOSE #40
>>OPEN #40 AS “n” FOR INPUT
>>VR(100)=STRTOD(3,40,101,102)
>>PRINT VR(100),VR(101),VR(102)
```

```
0.0000     7.0000     13.0000
>>CLOSE #40
>>DEL “N”
```

EXAMPLE 3:

```
>>OPEN #40 AS “n” FOR OUTPUT(1)
>>PRINT #40,”123”
>>CLOSE #40
>>OPEN #40 AS “n” FOR INPUT
>>VR(100)=STRTOD(3,40,101,102)
>>PRINT VR(100),VR(101),VR(102)
123.0000     7.0000     13.0000
>>CLOSE #40
>>DEL “N”
```

- Subtract

TYPE:

Mathematical Operator

SYNTAX:

```
<expression1> - <expression2>
```

DESCRIPTION:

Subtracts expression2 from expression1

PARAMETERS:

Expression1:	Any valid TrioBASIC expression
Expression2:	Any valid TrioBASIC expression

EXAMPLE:

Evaluate 2.1 multiply by 9 and subtract the result from 10, this will then be stored in **VR** 0. Therefore **VR** 0 holds the value -8.9

```
VR(0)=10-(2.1*9)
```

SYNC

TYPE:

Axis command

DESCRIPTION:

The **SYNC** command is used to synchronise one axis with a moving position on another axis. It does this by linking the **DPOS** of the slave axis to the **MPOS** of the master. So both axes must be programed in the same scale (for example mm). This can be used to synchronise a robot to a point on a conveyor. The user can define a time to synchronise and de-synchronise.

The synchronising movement on the base axis is the sum of two parts:

- The conveyor movement from the 'sync_pos', this is the movement of the demand point along the conveyor.
- The movement to 'pos1', this is the position in the current **USER_F**RAM**E** where the sync_pos was captured on the slave axis.

When the axis is synchronised it will follow the movements on the 'sync_axis'. As the **SYNC** does not fill the **MTYPE** buffer you can perform movements while synchronised.

To synchronise to a new **USER_FRAME** using **SYNC**(20) requires the kinematic runtime **FEC**

As **SYNC** does not get loaded in to the move buffer it is not cancelled by **CANCEL** or **RAPIDSTOP**, you have to perform **SYNC**(4). When a software or hardware limit is reached the **SYNC** is immediately stopped with no deceleration.

Typically you can use the captured position for example **REG_POS**, or a position from a vision system for the 'sync_position'. The pos1, pos2 and pos3 are typically the position of the sensor/ vision system in the current **USER_FRAME**.

SYNTAX:

SYNC(control, sync_time, [sync_position, sync_axis, pos1[, pos2 [,pos3]]])

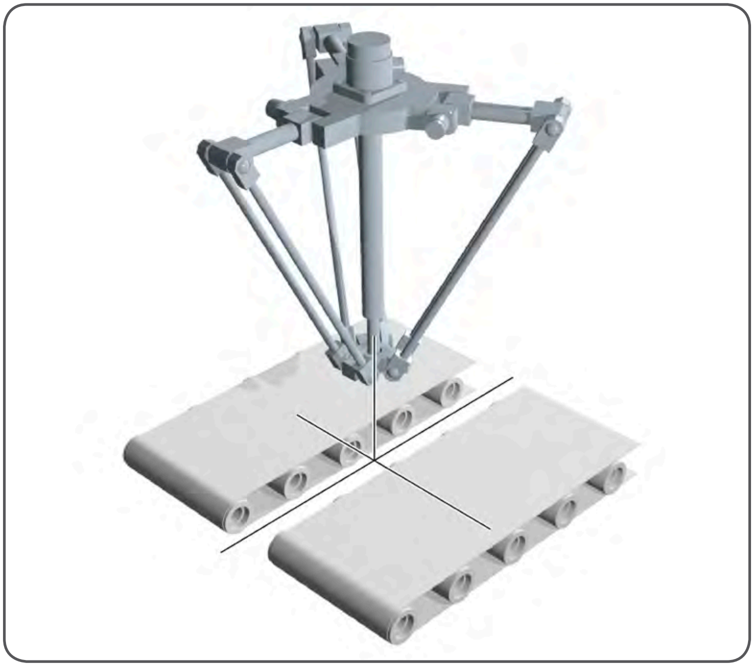
PARAMETERS:

Parameter	Description

control:	1 = Start synchronisation, requires minimum first 5 parameters
	4 = Stop synchronisation, requires minimum first 2 parameters
	10 = Re-synchronise to another axis, requires minimum first 5 parameters
	20 = Re-synchronise to **USER_FRAMEB**, requires minimum first 5 parameters
sync_time:	Time to complete the synchronisation movement in milliseconds
sync_position:	The captured position on the sync_axis.
sync_axis:	The axis to synchronise with.
pos1:	Absolute position on the first axis on the base array
pos2:	Absolute position on the second axis on the base array
pos3:	Absolute position on the third axis on the base array

EXAMPLE:

The robot must pick up the components from one conveyor and place them at 100mm pitch on the second. The registration sensor is at 385mm from the robots origin and the start of the second conveyor is 400mm from the robots origin.



```
'axis(0) - robot axis x
'axis(1) - robot axis y
'axis(2) - robot axis z
'axis(3) - robot wrist rotate
'These are the actual robot axis, FRAME=14 can be applied to these

'axis(10) - conveyor axis
'axis(11) - conveyor axis
'These are the real conveyors that you wish to link to

  'Sensor and conveyor offsets
  sen_xpos = 385
  conv1_yoff = 200
  conv2_yoff = -250
  conv2_xoff = 40
  place_pos = 0

  BASE(0,1)
  'Move to home position.
  MOVEABS(200,50)
  'start conveyors
  DEFPOS(0) AXIS(11) ' reset conveyor position for place
  FORWARD AXIS(10)
  FORWARD AXIS(11)
  WAIT IDLE

  WHILE(running)
    REGIST(20,0,0,0,0) AXIS(10)
    WAIT UNTIL MARK AXIS(10)

    SYNC(1, 1000, REG_POS, 10, sen_xpos , conv1_yoff)
    WAIT UNTIL SYNC_CONTROL AXIS(0)=3
    'Now synchronised
    GOSUB pick

    SYNC(10, 1000, place_pos, 11, conv2_xoff, conv2_yoff)
    WAIT UNTIL SYNC_CONTROL AXIS(0)=3
    'Now synchronised
    GOSUB place

    SYNC(4, 500)
    place_pos = place_pos + 100
  WEND
```

SEE ALSO:
SYNC_CONTROL, SYNC_TIMER, USER_FRAME, USER_FRAMEB

SYNC_CONTROL

TYPE:
Axis parameter (Read Only)

DESCRIPTION:
SYNC_CONTROL returns the current **SYNC** state of the axis

VALUE:

0	No synchronisation
1	Starting synchronisation
2	Performing synchronisation movement
3	Synchronised
4	Stopping synchronisation
5	Starting interpolated movement on second or third axis
6	Performing interpolated movement on second or third axis
10	Starting re- synchronisation
11	Performing re- synchronisation
20	Starting re-synchronisation to a different **USER_FRAME**
21	Performing re-synchronisation to a different **USER_FRAME**

EXAMPLE:
Synchronise to a conveyor linking to a position defined from registration, then wait until synchronisation before picking a part

```
'Set up start position and link to conveyor
  SYNC(10, 500, REG_POS AXIS(5), 5) AXIS(0)
  WAIT UNTIL SYNC_CONTROL AXIS(0)= 3
  GOSUB pick_part
```

SEE ALSO:
SYNC

SYNC_TIMER

TYPE:
Axis parameter (Read Only)

DESCRIPTION:
SYNC_TIMER returns the elapsed time of the synchronisation or re-synchronisation phase of **SYNC**. Once the synchronisation is complete the **SYNC_TIMER** will return the completed synchronisation time.

VALUE:
The elapsed time of the synchronisation phase in milliseconds

EXAMPLE:
Synchronise to a conveyor linking to a position defined from registration, then wait until synchronisation before picking a part

```
'Set up start position and link to conveyor
  SYNC(10, 500, REG_POS AXIS(5), 5) AXIS(0)
  WAIT UNTIL SYNC_TIMER AXIS(0)= 500
  GOSUB pick_part
```

SEE ALSO:
SYNC

SYSTEM_ERROR

TYPE:
System Parameter

DESCRIPTION:

The system errors are in blocks based on the following byte masks:	
System errors	0x0000ff
Configuration errors	0x00ff00
Unit errors	0xff0000
The following are system errors:	
Ram error	0x000001
Battery error	0x000002

Invalid module error	0x000004
VR/TABLE corrupt entry	0x000008
The following are configuration errors:	
Unit error	0x000100
Station error	0x000200
IO Configuration error	0x000400
Axes Configuration error	0x000800
The following are Unit errors:	
Unit Lost	0x010000
Unit Terminator Lost	0x020000
Unit Station Lost	0x040000
Invalid Unit error	0x080000
Unit Station Error	0x100000

SYSTEM_LOAD

TYPE:
System parameter (Read Only)

DESCRIPTION:
SYSTEM_LOAD returns the amount of time that is used by the system and motion software. The value is expressed as a percentage of the current servo period. The remaining time, that is 100 minus **SYSTEM_LOAD** percent, is therefore available to the application programs.

When setting **SERVO_PERIOD** appropriate to the number of axes running, the value of **SYSTEM_LOAD** should normally not be more than 55%.

VALUE:
The percentage of the servo period time that is used for system and motion processing.

EXAMPLE:
From the terminal 0 command line, read the percentage of servo time being used by the system firmware.

```
>>?SYSTEM_LOAD
23.1390
>>
```

The remaining processing time, 76.8610% is available for the multi-tasking **BASIC** or IEC61131-3 programs.

SEE ALSO:
SYSTEM_LOAD_MAX

SYSTEM_LOAD_MAX

TYPE:
System parameter

DESCRIPTION:
SYSTEM_LOAD_MAX returns the maximum value of **SYSTEM_LOAD** since power-up, or since **SYSTEM_LOAD_MAX** was last set to 0. If **SYSTEM_LOAD_MAX** is greater than 100 then at some point the firmware system and motion processing has overflowed the servo period. The number of axes should be reduced or the **SERVO_PERIOD** set to a higher value.

VALUE:
The maximum percentage of servo period time that is used for system and motion processing.

EXAMPLE 1:
From the terminal 0 command line, read the max percentage of servo time being used by the system firmware.

```
>>?SYSTEM_LOAD_MAX
56.9780
>>
```

EXAMPLE 2:
Reset the **SYSTEM_LOAD_MAX** value so that it can record a new maximum value since reset.

```
>>SYSTEM_LOAD_MAX = 0
>>
```

SEE ALSO:
SYSTEM_LOAD

T_REF

TYPE:
Axis Parameter

DESCRIPTION:
T_REF is identical to DAC.

SEE ALSO:
DAC_OUT

T_REF_OUT

TYPE:
Axis Parameter

DESCRIPTION:
T_REF_OUT is identical to **DAC_OUT**.

SEE ALSO:
DAC_OUT

TABLE

TYPE:
System Command

SYNTAX:
```
value = TABLE(address [, data0..data35])
```

DESCRIPTION:
The **TABLE** command can be used to load and read back the internal **TABLE** values. As the table can be written to and read from, it may be used to hold information as an alternative to variables.

The table values are floating point and can therefore be fractional.

You can clear the **TABLE** using **NEW "TABLE"**

PARAMETERS:

value:	returns the value stored at the address or -1 if used as part of a write
address:	The address of the first point of a write, or the address to read
data0:	The data written to the address
data1:	The data written to address+1
data2:	The data written to address+2
…	
data35	The data written to address+35

EXAMPLES:

EXAMPLE 1:

This loads the **TABLE** with the following values, starting at address 100:

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

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

EXAMPLE 2:

Use the command line to read the value stored in address 1000

```
>>PRINT TABLE(1000)
1234.0000
>>
```

SEE ALSO:

FLASHVR, NEW, TSIZE

TABLE_POINTER

TYPE:

Axis Parameter (Read Only)

DESCRIPTION:

Using the **TABLE_POINTER** command it is possible to determine which **TABLE** memory location is currently being used by the CAM or **CAMBOX**.

TABLE_POINTER returns the current table location that the CAM function is using. The returned number contains the table location and divides up the interpolated distance between the current and next **TABLE** location to indicate exact location.

The user can load new **CAM** data into previously processed **TABLE** location ready for the next **CAM** cycle. This is ideal for allowing a technician to finely tune a complex process, or changing recipes on the fly whilst running.

VALUE:

The value is returned of type X.Y where X is the current **TABLE** location and Y represents the interpolated distance between the start and end location of the current **TABLE** location.

EXAMPLE:

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

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

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

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

```
    FORWARD

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

SEE ALSO:

CAM, CAMBOX, TABLE

TABLEVALUES

TYPE:

System Command

SYNTAX:

TABLEVALUES(first, last [,format])

DESCRIPTION:

Returns a list of table values starting at the table address specified. The output is a comma delimited list of values.

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

PARAMETERS:

first:	First **TABLE** address to be returned
last:	Last **TABLE** address to be returned
format:	Format for the list.
	0 = Uncompressed comma delimited text (default) 1 = Compressed comma delimited text, repeated values are compressed using a repeat count before the value (k7,0.0000 representing 7 successive values of 0.0000). Single values do not have the repeat count;

EXAMPLE:

For a controller containing the values 0.0, 0.1, 0.1, 0.1, 0.2, 0.2, 0.0 in addresses 1 to 7:-

```
>>TABLEVALUES(1,7,0)
0.0000,0.1000,0.1000,0.1000,0.2000,0.2000,0.0000
>>
>>TABLEVALUES(1,7,1)
0.0000,k3,0.1000,k2 0.2000,0.0000
>>
```

TAN

TYPE:

Mathematical Function

SYNTAX:

`value = TAN(expression)`

DESCRIPTION:

Returns the **TANGENT** of an expression. This is valid for any value expressed in radians.

PARAMETERS:

value:	The **TANGENT** of the expression
expression:	Any valid TrioBASIC expression.

EXAMPLE:

Print the tangent of 0.5 using the command line.

```
>>PRINT TAN(0.5)
 0.5463
>>
```

TANG_DIRECTION

TYPE:
Axis Parameter

DESCRIPTION:
When used with a 2 axis X-Y system, this parameter returns the angle in radians that represents the vector direction of the interpolated axes.

VALUE:
The value returned is between -PI and +PI and is determined by the directions of the interpolated axes.

value	X	Y
0	0	1
PI/2	1	0
PI/2 (+PI or -PI)	0	-1
-PI/2	-1	0

EXAMPLES:

EXAMPLE1:
Note scale_factor_x **MUST** be the same as scale_factor_y

```
UNITS AXIS(4)=scale_factor_x
UNITS AXIS(5)=scale_factor_y
BASE(4,5)
MOVE(100,50)
angle = TANG_DIRECTION
```

EXAMPLE2:

```
BASE(0,1)
angle_deg = 180 * TANG_DIRECTION / PI
```

TEXT_FILE_LOADER

TYPE:
Command

SYNTAX:

TEXT_FILE_LOADER[(function [, parameter[,value]])]

DESCRIPTION:

The **TEXT_FILE_LOADER** command controls the **TEXT_FILE_LOADER_PROGRAM** on the controller. This function allows the **TEXT_FILE_LOADER** to be controlled and configured from the **BASIC**. **TEXT_FILE_LOADER_PROC** can be set to define which process the **TECT_FILE_LOADER_PROGRAM** runs on.

The **TEXT_FILE_LOADER_PROGRAM** is the controller end of the fast file transfer process that communicates with the file loading functionality of PCMotion.

If no parameters are used then the function is 0.

PARAMETERS:

function:	description:
0	Run the **TEXT_FILE_LOADER** program
1	Read a **TEXT_FILE_LOADER** parameter
2	Write a **TEXT_FILE_LOADER** parameter

FUNCTION = 0:

SYNTAX:

TEXT_FILE_LOADER

TEXT_FILE_LOADER (0)

DESCRIPTION:

Starts up the **TEXT_FILE_LOADER** communication protocol as a program. The **TEXT_FILE_LOADER** program will take up a user process if it is run automatically or manually.

The **TEXT_FILE_LOADER** program is normally started automatically when you open a file load connection. You can call it manually if you wish to specify which process it should run on.

If you execute **TEXT_FILE_LOADER** manually the program it runs in will suspend at the **TEXT_FILE_LOADER** line. The **TEXT_FILE_LOADER** therefore should be the last line of the program to execute.

FUNCTION = 1 AND FUNCTION = 2:

SYNTAX:

value = **TEXT_FILE_LOADER** (function, parameter [,value])

DESCRIPTION:
Functions 1 and 2 are used to (1) read and (2) write parameters from the **TEXT_FILE_LOADER_PROGRAM**.

The default destination for transparent protocol transfers should be set before any transfers occur.

PARAMETERS:

Parameter:	Description:	Values:
0	Transfer status parameter (read only)	0 = no transfer active 1 = transfer active
1	Default destination for transparent transfers	0 = **TEMP** file 1 = **FIFO** file 2 = **SDCARD**

EXAMPLES:

EXAMPLE 1:
Wait for a transfer to start then process the characters as they arrive at on the controller.

```
‘ wait for a file transfer to start
WAIT UNTIL TEXT_FILE_LOADER(1,0) = 1

‘ process this file
WHILE KEY#fifo_channel
    GET#fifo_channel,k
    PRINT #echo_channel,CHR(k);
    IF k=13 THEN PRINT #echo_channel, CHR(10);

    IF k>=65 AND k<=90 THEN ‘A to Z
        ltflag=0
        spflag=0
        value=0
        GOTO command_pro
    ENDIF
WEND
```

EXAMPLE 2:
Load a file into a **FIFO** then configure the **FILE** to be read back into the **BASIC**.

```
‘Set the FIFO as default file location for transparent protocol
TEXT_FILE_LOADER(2,1,1)
‘ initialise fifo
OPEN #fifo_channel AS “TRANSFER_FILE” FOR FIFO_WRITE(fifo_size)
CLOSE #fifo_channel
```

```
' open fifo to read
OPEN #fifo_channel AS "TRANSFER_FILE" FOR FIFO_READ

' run
WHILE running
    ' wait for a file transfer to start
    WAIT UNTIL TEXT_FILE_LOADER(1,0)
    WHILE KEY#fifo_channel
        GET#fifo_channel,char
        PRINT#5, CHR(char)
    WEND
WEND
```

SEE ALSO:

TEXT_FILE_LOADER_PROC

TEXT_FILE_LOADER_PROC

TYPE:

System Parameter (**MC_CONFIG**)

DESCRIPTION:

When the TrioPC ActiveX starts a text file transfer to the *Motion Coordinator*, the **TEXT_FILE_LOADER_PROGRAM** is started on the highest available process. **TEXT_FILE_LOADER _PROC** can be set to specify a different process for the **TEXT_FILE_LOADER _PROGRAM.** If the defined process is in use then the next lower available process will be used.

TEXT_FILE_LOADER _PROC can be set in the MC_**CONFIG** script file.

VALUE:

-1	Use the highest available process (default)
0 to max process	Run on defined process

EXAMPLES:

EXAMPLE1:

Set **TEXT_FILE_LOADER _PROGRAM** to start on process 19 or lower (using the command line terminal).

```
>> TEXT_FILE_LOADER_PROC=19
>>
```

EXAMPLE2:

Remove the **TEXT_FILE_LOADER _PROC** setting so that **TEXT_FILE_LOADER _PROGRAM** starts on default process (using **MC_CONFIG**).

```
'MC_CONFIG script file
TEXT_FILE_LOADER_PROC = -1  'Start on default process on connection
```

SEE ALSO:

TEXT_FILE_LOADER

TICKS

TYPE:

Process Parameter

DESCRIPTION:

The current count of the process clock ticks is stored in this parameter. The process parameter is a 64 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.

As **TICKS** is a process parameter each process will have its own counter.

VALUE:

The value of the 64bit counter

EXAMPLE:

With **SERVO_PERIOD** set to 1000 use **TICKS** for a 3 second delay

```
delay:
 TICKS=3000
 OP(9,ON)
test:
 IF TICKS<=0 THEN OP(9,OFF) ELSE GOTO test
```

TIME$

TYPE:

System Parameter

DESCRIPTION:

TIME$ is used as part of a **PRINT** statement or a **STRING** variable to write the current time from the real time clock. The date is printed in the format Hour:Minute:Second.

The **TIME$** is set through the **TIME** command

PARAMETERS:

None.

EXAMPLES

EXAMPLE 1:

Print the current time from the real time clock to the command line.

```
>>print time$
15:51:06
>>
```

EXAMPLE 2:

Create an error message to print later in the program

```
DIM string1 AS STRING(30)
string1 = “Error occurred at “ + TIME$
```

SEE ALSO:

PRINT, STRING, TIME

TIME

TYPE:

System Parameter

DESCRIPTION:

Allows the user to set and read the time from the real time clock.

VALUE:

Read = the number of seconds since midnight (24:00 hours)

Write = the time in 24hour format hh:mm:ss	

EXAMPLES:

EXAMPLE 1:

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

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

EXAMPLE 2:

Calculate elapsed time in seconds

```
time1 = TIME
'wait for event
time2 = TIME
timeelapsed = time1-time2
```

SEE ALSO:

TIME$

TIMER

TYPE:

Command

SYNTAX:

TIMER(switch, output, pattern, time[,option])

DESCRIPTION:

The **TIMER** command allows an output or a selection of outputs to be set or cleared for a predefined period of time. There are 64 timer slots available, each can be assigned to any outputs. The timer can be configured to turn the output ON or OFF.

PARAMETERS:

switch:	The timer number in the range 0-63
output:	Selects the physical output or first output in a group. Range 0-31.
pattern:	1 = for a single output. Number = If set to a number this represents a binary array of outputs to be turned on. Range 0-65535.

time:	The period of operation in milliseconds
option:	Inverts the output, set to 1 to turn OFF at start and ON at end.

EXAMPLES:

EXAMPLE1:

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

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

EXAMPLE2:

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

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

EXAMPLE3:

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

```
WHILE feed=ON
 MOVELINK(30,60,60,0,1)
 MOVELINK(70,100,0,60,1)
 WAIT LOADED ‘Wait until the tracking phase starts
 TIMER(42,10,1,250) ‘Fire the output during the tracking phase
 MOVELINK(-100,200,50,50,1)
WEND
```

TIMER

TYPE:

Command

SYNTAX:

TIMER(switch, output, pattern, time[,option])

DESCRIPTION:

The **TIMER** command allows an output or a selection of outputs to be set or cleared for a predefined period of time. There are 64 timer slots available, each can be assigned to any outputs. The timer can be configured to turn the output ON or OFF.

PARAMETERS:

switch:	The timer number in the range 0-63
output:	Selects the physical output or first output in a group. Range 0-31.
pattern:	1 = for a single output. Number = If set to a number this represents a binary array of outputs to be turned on. Range 0-65535.
time:	The period of operation in milliseconds
option:	Inverts the output, set to 1 to turn OFF at start and ON at end.

EXAMPLES:

EXAMPLE1:

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

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

EXAMPLE2:

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

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

EXAMPLE3:

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

```
WHILE feed=ON
```

```
    MOVELINK(30,60,60,0,1)
    MOVELINK(70,100,0,60,1)
    WAIT LOADED 'Wait until the tracking phase starts
    TIMER(42,10,1,250) 'Fire the output during the tracking phase
    MOVELINK(-100,200,50,50,1)
  WEND
```

TOOL_OFFSET

TYPE:
Axis Command

SYNTAX
TOOL_OFFSET(identity, x_offset, y_offset, z_offset)

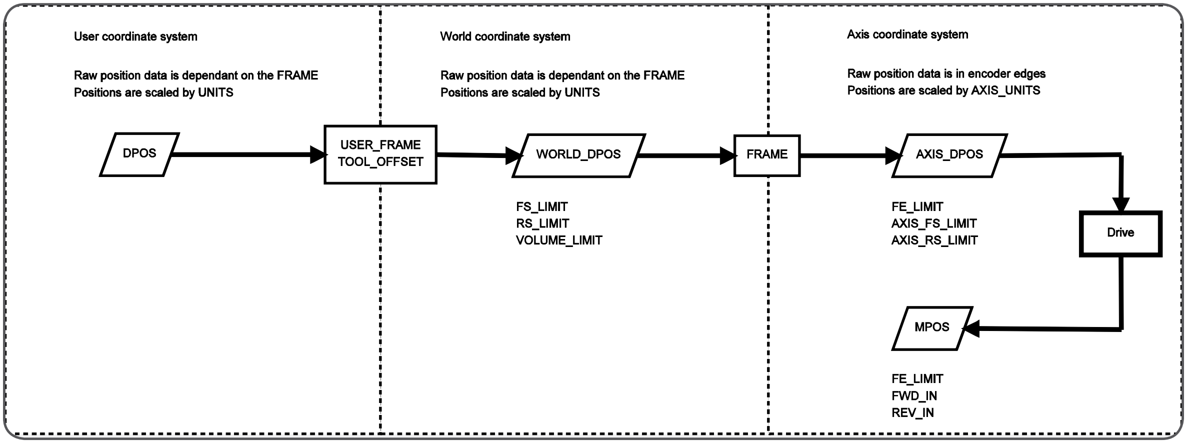
DESCRIPTION:
TOOL_OFFSET is used to adjust the programming point on a system. This is achieved by offsetting **DPOS** from the programming point. For example a wrist of the robot is the programming point and the tool offset can be used to adjust the programming point to the end of a tool on the wrist. Multiple tool points can be assigned and the user can switch between points on the fly.

TOOL_OFFSET requires the kinematic runtime **FEC**

The default **TOOL_OFFSET** has the identity 0 and is equal to the world coordinate system origin, this cannot be modified. If you wish to disable the **TOOL_OFFSET** select **TOOL_OFFSET**(0).

TOOL_OFFSETs are applied on the axis **FRAME_GROUP**. If no **FRAME_GROUP** is defined then a runtime error will be generated. **TOOL_OFFSET** supports a **FRAME_GROUP** containing 2-6 axes.

Movements are loaded with the selected **TOOL_OFFSET**. This means that you can buffer a sequence of movements on different tools. The active **TOOL_OFFSET** is the one associated with the movement in the **MTYPE**. If the **FRAME_GROUP** is **IDLE** then the active **TOOL_OFFSET** is the selected **TOOL_OFFSET**.



If you wish to check which **USER_FRAME**, **TOOL_OFFSET** and **VOLUME_LIMIT** are active you can print the details using **FRAME_GROUP**(group).

PARAMETERS

identity:	0 = default group which is set to the world coordinate system
	1 to 31 = Identification number for the user defined tool offset.
x_offset:	Offset in the x axis from the world origin to the user origin.
y_offset:	Offset in the y axis from the world origin to the user origin.
z_offset:	Offset in the z axis from the world origin to the user origin.

EXAMPLE

A tool is rotated 45degrees about the y axis and has an offset of 20mm in the x direction, 30mm in the y direction and 300mm in the z direction. The programmer wants to move the tool forward on its axis so a **TOOL_OFFSET** is applied to adjust the position to the tool tip, then a USER_FRAMEis applied to allow programming about the tool axis.

```
'Configure USER_FRAME and TOOL_OFFSET
FRAME_GROUP(0,0,0,1,2)
USER_FRAME(1, 20, 30, 300, 0, PI/4, 0)
TOOL_OFFSET(1, 20, 30, 300)
'Select tool and frame and start motion.
USER_FRAME(1)
TOOL_OFFSET(1)
BASE(2)
FORWARD
```

TRIGGER

TYPE:
System Command

DESCRIPTION:
Starts a previously set up **SCOPE** command. This allows you to start the scope capture at a specific part of your program.

EXAMPLE:
The *Motion* Perfect oscilloscope is set to record **MPOS** and **DPOS** of axis 0. The settings allow for program trigger and a repeat trigger. This loop can then be used as part of a PID tuning routine.

```
WHILE IN(tuning)=ON
DEFPOS(0)
TRIGGER
 WA(5) 'Allow the scope to start
 MOVE(100)
 WAIT IDLE
 WA(100)
 MOVE(-100)
 WA(100)
WEND
```

TRIOPCTESTVARIAB

TYPE:
Reserved Keyword

TROFF

TYPE:
System Command

SYNTAX:
TROFF ["program"]

DESCRIPTION:
The trace off command resumes execution of the **SELECTed** or specified program. The command can be

included in a program to resume the execution of that program.

For de-bugging the *Motion* Perfect breakpoint tool should be used.

PARAMETERS:

program:	The name of the program which you wish to resume

EXAMPLE:

Resume execution of a program names **TEST**

```
>>TROFF "TEST"
OK
>>%[Process 21:Program TEST] – Released
```

SEE ALSO:

HALT, STOP, STEPLINE, TRON

TRON

TYPE:

System Command

SYNTAX:

TRON ["program"]

DESCRIPTION:

The trace on command pauses the **SELECTed** or specified program. The command can be included in a program to pause the execution of that program. The program can then be stepped through a single line, run or halted.

PARAMETERS:

program:	The name of the program which you wish to step

Motion Perfect highlights lines containing **TRON** in its editor and debugger. For de-bugging the *Motion* Perfect breakpoint tool should be used.

EXAMPLES:

EXAMPLE 1:

Use suspend a program by including **TRON**. Another program will then use **STEPLINE** to step through until the **TRON**.

```
TRON
```

```
MOVE(0,10)
MOVE(10,0)
TROFF
MOVE(0,-10)
MOVE(-10,0)
```

EXAMPLE 2:

Start a program by stepping into the first line, then stepping through. The line that is stepped to is displayed

```
>>SELECT "STARTUP"
STARTUP selected
>>TRON
OK
>>%[Process 20:Line 3] - Paused
TABLE(0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0)

STEPLINE
OK
>>%[Process 20:Line 4] - Paused
TABLE(10,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0)

STEPLINE
OK
>>%[Process 20:Line 5] - Paused
TABLE(20,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0)
```

EXAMPLE 3:

Pause a program called test that is currently running:

```
TRON "TEST"
OK
>>%[Process 21:Line 6] - Paused
WA(4)
```

SEE ALSO:

HALT, **STOP**, **STEPLINE**, **TROFF**

TRUE

TYPE:

Constant

DESCRIPTION:

The constant **TRUE** takes the numerical value of -1.

EXAMPLE:

Checks that the logical result of input 0 and 2 is true

```
t=IN(0)=ON AND IN(2)=ON
IF t=TRUE THEN
 PRINT "Inputs are on"
ENDIF
```

TSIZE

TYPE:

System Parameter (Read Only)

DESCRIPTION:

Returns the size of the **TABLE**.

Not all table positions are battery backed, see your controller information for exact values.

VALUE:

The size of the **TABLE**

EXAMPLE:

Check the size of the table and write to the last position in the table (remember the table starts at position 0).

```
>>?tsize
500000.0000
>>table(499999,123)
>>
```

UCASE

TYPE:
STRING Function

SYNTAX:
UCASE(string)

DESCRIPTION:
Returns a new string with the input string converted to all upper case.

PARAMETERS:

string:	String to be used

EXAMPLES:

EXAMPLE 1:
Pre-define a variable of type string and later print it in all upper case characters:

```
DIM str1 AS STRING(32)
str1 = “Trio Motion Technology”
PRINT UCASE(str1)
```

SEE ALSO:
CHR, STR, VAL, LEFT, RIGHT, MID, LEN, LCASE, INSTR

UNIT_CLEAR

TYPE:
System command

DESCRIPTION:
Clears all the bits in the **UNIT_ERROR** system parameter.

VALUE:
This command takes no values

EXAMPLE:
Clear the **UNIT_ERROR** bits and then check for which module or modules may be in error.

```
UNIT_CLEAR
```

```
WA(10)
PRINT UNIT_ERROR[0]
```

SEE ALSO:
SLOT, SYSTEM_ERROR, UNIT_ERROR

UNIT_DISPLAY

TYPE:
System Parameter

DESCRIPTION:
Reserved Keyword

UNIT_ERROR

TYPE:
System Parameter (read only)

DESCRIPTION:
The **UNIT_ERROR** provides a simple single indicator that at least one module is in error and can indicate multiple modules that have an error. The value returns details which SLOTs are in error.

VALUE:
A binary sum of the module **SLOT** numbers for the modules which are in error.

Bit	Value	Slot
0	1	0
1	2	1
2	4	2
3	8	3
...		

EXAMPLE:
Test for the module in slot 1 having an error which is a ‘Unit station error’. This could indicate a problem with a drive on the network in slot 1.

```
IF UNIT_ERROR=2 AND SYSTEM_ERROR=1048576 THEN
  ‘Handle Unit station error for slot 1
```

```
    ...
ENDIF
```

SEE ALSO:

SLOT, SYSTEM_ERROR, UNIT_CLEAR

UNIT_SW_VERSION

TYPE:

Reserved Keyword

UNITS

TYPE:

Axis Parameter

DESCRIPTION:

UNITS is a conversion factor that allows the user to scale the edges/ stepper pulses to a more convenient scale. The motion commands to set speeds, acceleration and moves use the **UNITS** scalar to allow values to be entered in more convenient units e.g.: mm for a move or mm/sec for a speed.

Units may be any positive value but it is recommended to design systems with an integer number of encoder pulses/user unit. If you need to use a non integer number you should use **ENCODER_RATIO**. **STEP_RATIO** can be used for non integer conversion on a stepper axis.

VALUE:

The number of counts per required units.

EXAMPLES:

EXAMPLE 1:

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 in the controller. One rev is equal to 5mm therefore there are 4000/5=800 edges/mm.

```
>>UNITS=1000*4/5
```

EXAMPLE 2:

A stepper motor has 180 pulses/rev. There is a built in 16 multiplier so the controller will use 180*16 counts per revolution.

To program in revolutions the unit conversion factor will be:

```
>>UNITS=180*16
```

SEE ALSO:

ENCODER_RATIO, STEP_RATIO

UNLOCK

TYPE:

System Command (command line only)

SYNTAX:

UNLOCK(code)

DESCRIPTION:

Unlocks a *Motion Coordinator* which has previously been locked using the **LOCK** command.

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

You should use *Motion* Perfect to **LOCK** and **UNLOCK** your controller.

If you forget the security code number which was used to lock the *Motion Coordinator*, it may have to be returned to your supplier to be unlocked.

PARAMETERS:

code:	Any 7 digit integer number

SEE ALSO:

LOCK

USER_FRAME

TYPE:

Axis Command

SYNTAX

USER_FRAME(identity [, x_offset, y_offset, z_offset [, x_rotation [, y_rotation [, z_rotation]]]])

DESCRIPTION:

The **USER_FRAME** allows the user to program in a different coordinate system. The **USER_FRAME** can be defined up to a 3-axis translation and rotation from the world coordinate origin. The rotations are applied using the Euler ZYX convention. This means that the z rotation is applied first, then the y is applied on the new coordinate system and finally the x is applied. The coordinate system is defined using the 'right hand rule' and the rotation of the origin is defined using the 'right hand turn'.

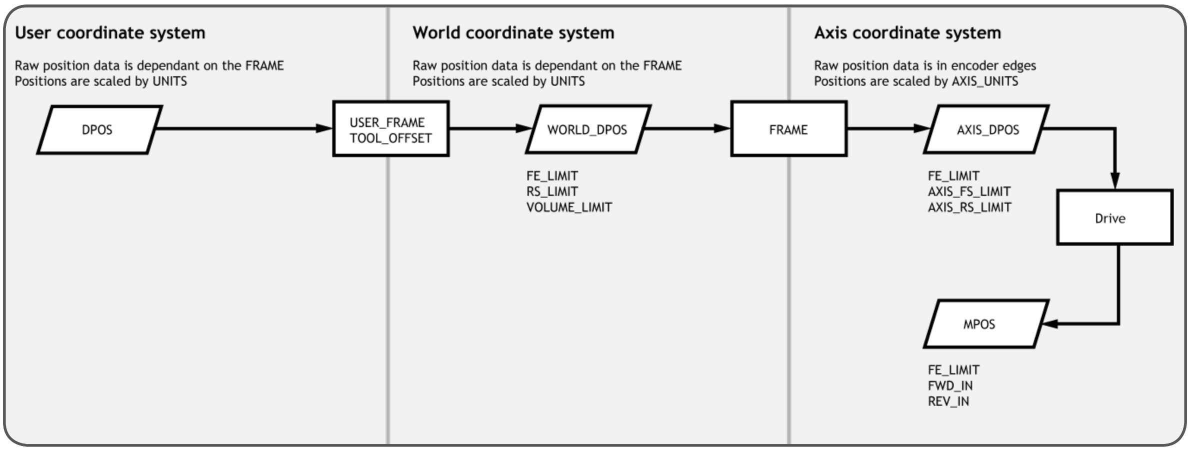
USER_FRAME requires the kinematic runtime **FEC**

The default coordinate system has the identity 0 and is equal to the world coordinate system, this cannot be modified. If you wish to disable the **USER_FRAME** select **USER_FRAME**(0).

USER_FRAMEs are applied on the axis **FRAME_GROUP**. If no **FRAME_GROUP** is defined then a runtime error will be generated.

Movements are loaded with the selected **USER_FRAME**. This means that you can buffer a sequence of movements on different **USER_FRAMES**. The active **USER_FRAME** is the one associated with the movement in the **MTYPE**. If the **FRAME_GROUP** is **IDLE** then the active **USER_FRAME** is the selected **USER_FRAME**.

The **USER_FRAME** is applied to all the axes in the **FRAME_GROUP**. This can be the same group as used by **FRAME**. The **FRAME_GROUP** does not have to be 3 axis, however the **USER_FRAME** will only process position for the axes in the **FRAME_GROUP**. It can be useful in a 2 axes **FRAME_GROUP** to perform a **USER_FRAME** rotation about the third axis.



If you wish to check which **USER_FRAME**, **TOOL_OFFSET** and **VOLUME_LIMIT** are active you can print the details using **FRAME_GROUP**(group).

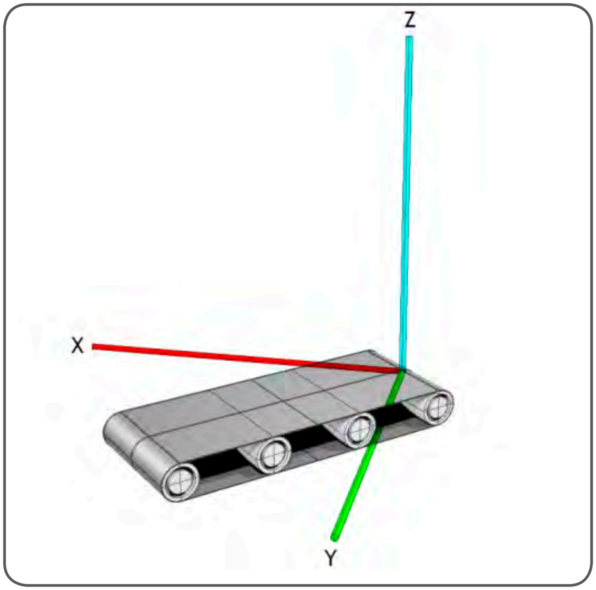
PARAMETERS

identity:	0 = default group which is set to the world coordinate system
	1 to 31 = Identification number for the user defined frame.
x_offset:	Offset in the x axis from the world origin to the user origin.
y_offset:	Offset in the y axis from the world origin to the user origin.
z_offset:	Offset in the z axis from the world origin to the user origin.
x_rot:	Rotation about the items x axis in radians.
y_rot:	Rotation about the items y axis in radians.
z_rot:	Rotation about the items z axis in radians.

EXAMPLES:

EXAMPLE 1:

A conveyors origin is at 45degrees to the world coordinate (robots) origin, as shown in the image. To ease programming a **USER_FRAME** is assigned to align the x axis with the conveyor so that it is possible to program in the conveyor coordinate system.




```
FRAME_GROUP(0,0,0,1,2)
USER_FRAME(1,0,0,0,PI/4)
```

EXAMPLE 2

Initialise a user coordinate system then perform a movement on the world coordinate system before starting a **FORWARD** on the first user coordinate system.

```
FRAME_GROUP(0,0,0,1,2)
BASE(0,1,2)
DEFPOS(10,20,30)
USER_FRAME(1,10,20,30,PI/2)
USER_FRAME(0)
MOVEABS(100,100,50)
WAIT IDLE
USER_FRAME(1)
FORWARD
```

USER_FRAME_TRANS

TYPE:

Mathematical Function

SYNTAX:

USER_FRAME_TRANS(user_frame_in, user_frame_out, tool_offset_in, tool_offset_out, table_in, table_out, [scale])

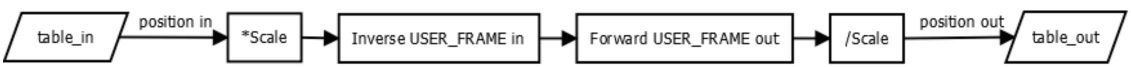
DESCRIPTION:

This function enables you to transform a set of positions from one frame to another. This could be used to take a set of positions from a vision system and transform them so that they are a set of positions relative to a conveyor.

USER_FRAME_TRANS requires the kinematic runtime **FEC**

It is required to set-up a **FRAME_GROUP** and **USER_FRAME** to use this function. If you do not wish to set up a **FRAME_GROUP** with real axis you can use virtual.

The **USER_FRAME** calculations are performed on raw position data which are integers. The table data is scaled by the scale parameter, for optimal resolution scale should be set to the **UNITS** of the robot.



As all the **USER_FRAME** transformations use the same coordinate scale it does not matter if the positions are supplied as raw positions or scaled by **UNITS**.

PARAMETERS:

user_frame_in:	The **USER_FRAME** identity that the points are supplied in
user_frame_out:	The **USER_FRAME** identity that the points are transformed to
tool_offset_in:	The **TOOL_OFFSET** identity that the points are supplied in
tool_offset_out:	The **TOOL_OFFSET** identity that the points are transformed to
table_in:	The start of the input positions
table_out:	The start of the generated positions
scale:	This parameter allows you to scale the table values (default 1000)

The table_in requires 12 values. Any that are not required should be set to zero for position and 1 for scale.

table_in	First axis position
table_in +1	Second axis position
table_in +2	Third axis position
table_in +3	Fourth axis position
table_in +4	Fifth axis position
table_in +5	Sixth axis position
table_in +6	First axis **FRAME_SCALE**
table_in +7	Second axis **FRAME_SCALE**
table_in +8	Third axis **FRAME_SCALE**
table_in +9	Fourth axis **FRAME_SCALE**
table_in +10	Fifth axis **FRAME_SCALE**
table_in +11	Sixth axis **FRAME_SCALE**

EXAMPLE:

USER_FRAME(vision) has been configured to the vision system relative to the robot origin. The conveyor has been configures in **USER_FRAME**(conveyor). To use the vision system positions on the conveyor **USER_FRAME** they must be transformed through **USER_FRAME_TRANS**.

```
USER_FRAME_TRANS(vision, conveyor, 0, 0, 200,300)
```

USER_FRAMEB

TYPE:

Axis Command

SYNTAX

USER_FRAMEB(identity)

DESCRIPTION:

USER_FRAMEB is only used with **SYNC**. It defines the new **USER_FRAME** to resynchronise to when performing the **SYNC**(20) operation. When the resynchronisation is complete **USER_FRAMEB** is the active **USER_FRAME**. **USER_FRAMEB** selects one of the defined USER_FRAMEs.

EXAMPLE:

The robot must pick up the components from one conveyor and place them on a second conveyor which is in a different **USER_FRAME**.

```
WHILE(running)
  USER_FRAMEB(conv1)
  REGIST(20,0,0,0,0) AXIS(10)
  WAIT UNTIL MARK AXIS(10)

  SYNC(1, 1000, REG_POS, 10, sen_xpos , conv1_yoff)
  WAIT UNTIL SYNC_CONTROL AXIS(0)=3
  'Now synchronised
  GOSUB pick

  USER_FRAMEB(conv2)
  SYNC(20, 1000, place_pos, 11, conv2_xoff, conv2_yoff)
  WAIT UNTIL SYNC_CONTROL AXIS(0)=3
  'Now synchronised
  GOSUB place

  SYNC(4, 500)
  place_pos = place_pos + 100
WEND
```

SEE ALSO:

SYNC, USER_FRAME

VAL

TYPE:
STRING Function

SYNTAX:
VAL(string)

DESCRIPTION:
Converts a string to a numerical value. If the string is not a numerical value then VAL returns 0.

PARAMETERS:

string:	String to be converted

EXAMPLES:

EXAMPLE 1:
Pre-define a variable of type string and then later, convert its current value to a numerical value stored in a **VR**. The resulting number in the **VR** is -132.456:

```
DIM str1 AS STRING(20)
str1 = “-123.456”
VR(100)=VAL(str1)
```

EXAMPLE 2:
Pre-define a variable of type string and then later, convert its current value to an integer numerical value stored in a local variable. The resulting number in the local variable is 1110:

```
DIM str2 AS STRING(10)
DIM number AS INTEGER
str2 = “987”
number = INT(VAL(str2)) + 123
```

SEE ALSO:
CHR, STR, LEN, LEFT, RIGHT, MID, LCASE, UCASE, INSTR

VALIDATE_ENCRYPTION_KEY

TYPE:
System Command

SYNTAX:

`VALIDATE_KEY (security_code_type, validation_key)`

DESCRIPTION:

VALIDATE_ENCRYPTION_KEY is used to check that the controller has the correct user or OEM security code programmed. If the correct security code is not programmed then **VALIDATE_ENCRYPTION_KEY** will produce a runtime error (parameter out of range) and so stop the program from functioning.

Motion Perfect has a tool to generate the validation keys

Do not put the user or **OEM** security code in the program as these must be kept secret.

PARAMETERS:

security_code_type	1	OEM security code
	2	User security code
validation_key	A string which is a validation keys that has been generated by *Motion* Perfect	

EXAMPLE:

Test that the user security code is valid before running the main program

```
'Validate the user security code
VALIDATE_ENCRYPTION_KEY(2,"1Wgltam0wzrbCVJwUgEnGU")
RUN "MAIN_PROGRAM"
```

SEE ALSO:

SET_ENCRYPTION_KEY, PROJECT_KEY

VECTOR_BUFFERED

TYPE:

Axis Parameter (Read only)

DESCRIPTION:

This holds the total vector length of the buffered moves. It is effectively the amount the VPU can assume is available for deceleration. It should be executed with respect to the first axis in the group.

VALUE:

The vector length of buffered moves on the axis group.

EXAMPLE:

Return the total vector length for the current buffered moves whose axis group begins with axis(0).

```
>>BASE(0,1,2)
>>? VECTOR_BUFFERED AXIS(0)
1245.0000
>>
```

VERIFY

TYPE:

Reserved Keyword

VERSION

TYPE:

System Parameter (read only)

DESCRIPTION:

Returns the version number of the firmware installed on the *Motion Coordinator*.

You can use *Motion* Perfect to check the firmware version when looking at the controller configuration.

VALUE:

Controllers' firmware version number.

EXAMPLE:

Check the version of the firmware using the command line

```
>>? VERSION
2.0100
>>
```

VFF_GAIN

TYPE:

Axis Parameter

DESCRIPTION:

The velocity feed forward gain is a constant which is multiplied by the change in demand position. Velocity feed forward gain can be used to decreases the following error during constant speed by increasing the output proportionally with the speed. For a velocity feed forward Kvff and change in position ΔPd, the contribution to the output signal is:

Ovff = Kvff x ΔPd

VALUE:

Velocity feed forward constant (default =0)

EXAMPLE:

Set the **VFF_GAIN** on axis 15 to 12

```
BASE(15)
VFF_GAIN=12
```

VIEW

TYPE:

Reserved Keyword

VOLUME_LIMIT

TYPE:

Axis Function

SYNTAX:

VOLUME_LIMIT(mode, [,table_offset])

DESCRIPTION:

VOLUME_LIMIT enables a software limit that restricts the motion into a defined three dimensional shape. The calculations are performed on **DPOS** and so it can be used in addition to a **FRAME**. The limit applies to axes defined in a **FRAME_GROUP**.

VOLUME_LIMIT requires the kinematic runtime **FEC**

If no **FRAME_GROUP** is defined then a 'parameter out of range' run time error will be returned when **VOLUME_LIMIT** is called.

All axes in the **FRAME_GROUP** must have the same **UNITS**

When the limit is active moves on all axes in the **FRAME_GROUP** are cancelled and so will stop with the programmed **DECEL** or **FAST_DEC**. Any active **SYNC** is also stopped. **AXISSTATUS** bit 15 is also set. This means you should set your **VOLUME_LIMIT** smaller than the absolute operating limits of the robot.

PARAMETERS:

mode:	0	**VOLUME_LIMIT** is disabled
	1	Cylinder with cone base volume

MODE = 1 CYLINDER WITH CONE BASE VOLUME

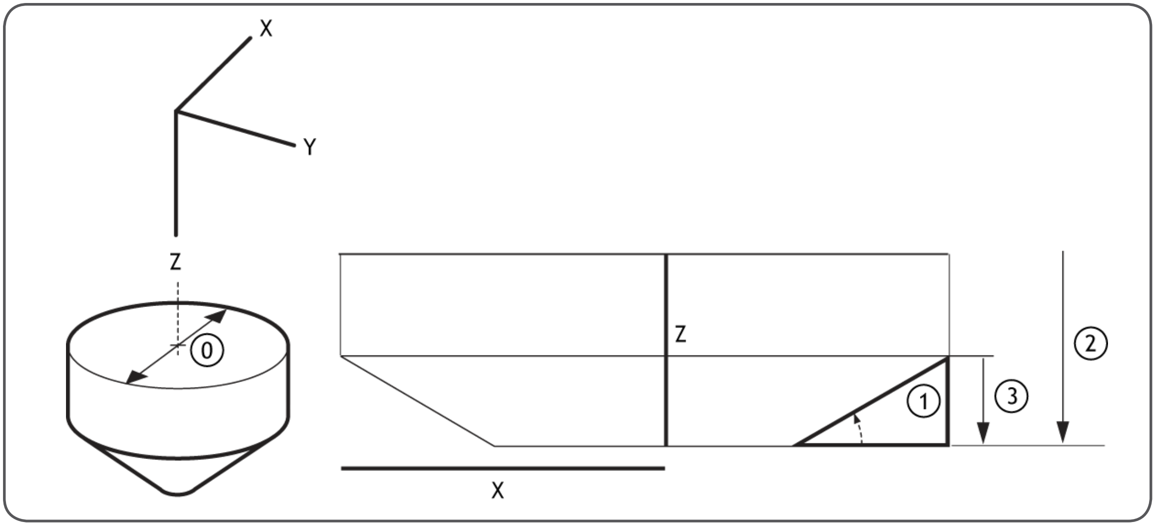
SYNTAX:

VOLUME_LIMIT(1, [,table_offset])

DESCRIPTION:

Mode 1 enables a cylinder with a cone base, this is a typical working volume for a delta robot.

The origin for the shape is the centre top . It is possible to align this with your coordinate system using the X,Y and Z offsets



If you wish to check which **USER_FRAME**, **TOOL_OFFSET** and **VOLUME_LIMIT** are active you can print the details using **FRAME_GROUP**(group).

PARAMETERS:

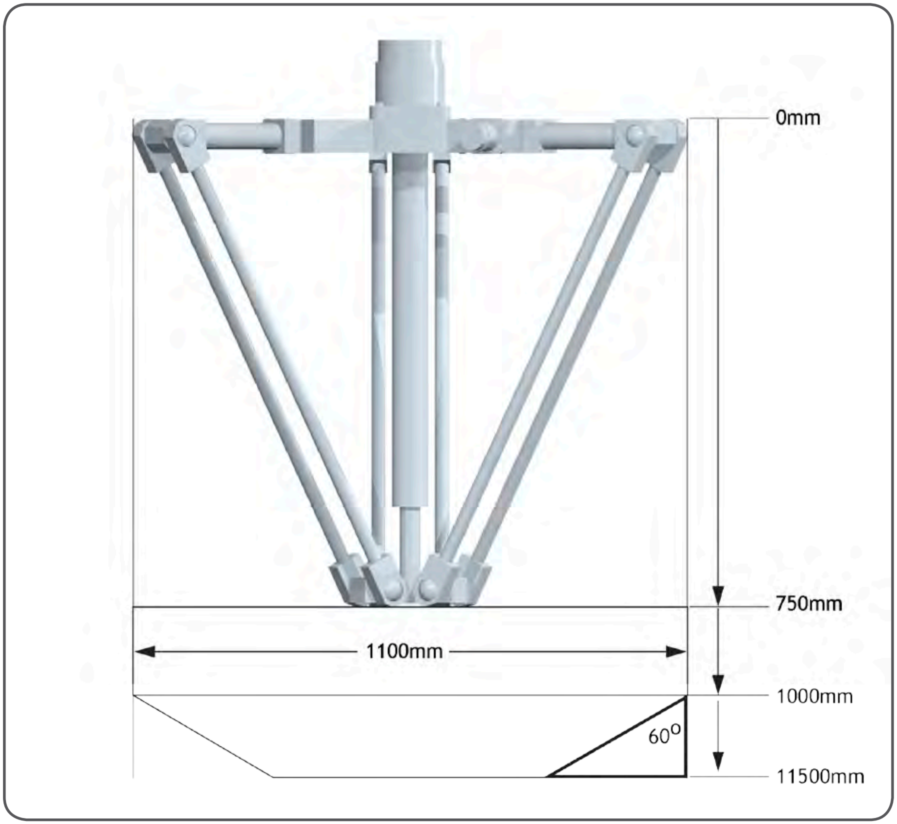
mode:	0	**VOLUME_LIMIT** is disabled
	1	Cylinder with cone base volume
table_offset:	The start position in the table to store the **VOLUME_LIMIT** configuration	

Mode 0 table values, all length values use **UNITS** from the first axis in the **FRAME_GROUP**.

0	Cylinder Diameter
1	Cone angle in radians
2	Total height
3	Cone height
4	X offset
5	Y offset
6	Z offset

EXAMPLE:

The cylinder with a flat base is typically used with delta robots (**FRAME**=14), the following example configures the **VOLUME_LIMIT** with this configuration.




```
TABLE(100,1100)' Cylinder diameter
TABLE(101,(60/360)* 2* PI)' Cone angle
TABLE(102,400)' Total height
TABLE(103,150)' Cone height
TABLE(104,0)' X offset
TABLE(105,0)' Y offset
TABLE(106,750)' Z offset

VOLUME_LIMIT(1,100)
```

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.

VALUE:
The velocity profile speed in user **UNITS**/second.

EXAMPLE:
Wait until command speed is achieved:

```
MOVE(100)
WAIT UNTIL SPEED=VP_SPEED
```

VR

TYPE:
System Command

SYNTAX:
value = 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.

VR can also be used to hold **ASCII** representations of **STRINGS** and can be assigned with a string value. To read the string value back you must use **VRSTRING**.

The numbered variables are globally shared between programs and can be used for communication between programs. Be careful when multiple programs write to the same **VR**.

PARAMETERS:

value:	The value written to or read from the **VR**
expression:	Any valid TrioBASIC expression that produces an integer

EXAMPLES:

EXAMPLE 1:

Put value 1.2555 into **VR**() variable 15. Note local variable 'val' used to give name to global variable:

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

EXAMPLE 2:

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

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

EXAMPLE 3:

Assign **VR**(65) with the value **VR**(0) multiplied by Axis 1 measured position

```
VR(65)=VR(0)*MPOS AXIS(1)
PRINT VR(65)
```

EXAMPLE 4:

Write a string into a sequence of **VR**'s starting at index 10

```
VR(10)="Hello World"
PRINT VR(10) 'Prints 72, ASCII for H
PRINT VRSTRING(10) 'Prints Hello World
```

VRSTRING

TYPE:
String Function

SYNTAX:
VRSTRING(variable)

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

PARAMETERS:

variable:	Number of first **VR**() in the character array.

EXAMPLES:

EXAMPLE1:
Print a sequence of characters stored in the **VR**'s starting at position 100.

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

EXAMPLE2:
Store the characters saved in the **VR**'s into one **STRING** variable.

```
DIM string2 AS STRING(11)
string2 = VRSTRING(0)
```

WA

TYPE:
Program Structure

SYNTAX:
WA(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:
Turn output 17 off 2 seconds after switching output 11 off.

```
OP(11,OFF)
WA(2000)
OP(17,ON)
```

WAIT

TYPE:
Command

SYNTAX:
WAIT UNTIL expression

DESCRIPTION:
Suspends program execution until the expression is **TRUE**.

It is very common to use onlyWAIT **IDLE** and **WAIT LOADED** as the expression. In this situation the **UNTIL** is optional. When **IDLE** and **LOADED** are part of an expression **UNTIL** is required.

PARAMETERS:

condition:	Any valid TrioBASIC expression

EXAMPLES:

EXAMPLE 1:

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

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

EXAMPLE 2:

Start a move and then suspend program execution until the move has finished. Note: This does not necessarily imply that the axis is stationary in a servo motor system.

```
MOVE(100)
WAIT IDLE
PRINT “Move Done”
```

EXAMPLE 3:

Switch output 45 ON at start of **MOVE**(350) and OFF at the end of that move.

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

EXAMPLE 4:

Force the program to wait until either the current move has finished or an input goes ON.

As the expression contains **UNTIL** and **IN**(12) the **UNTIL** is required.

```
MOVELINK(distance, link_dist, acceldist, deceldist, linkaxis)
WAIT UNTIL IDLE OR IN(12)=ON
```

WDOG

TYPE:

System Parameter

DESCRIPTION:

Controls the **WDOG** relay contact used for enabling external drives. The **WDOG**=ON command **MUST** be issued in a program prior to executing moves. It may then be switched ON and OFF under program control. If however a following error condition exists on any axis the system software will override the **WDOG** setting and turn watchdog contact OFF. When **WDOG**=OFF, the relay is opened, the analogue outputs are set to 0V, the step/direction outputs and any digital axis enable functions are disabled.

EXAMPLE:

```
WDOG=ON
```

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

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

WHILE .. WEND

TYPE:

Program Structure

SYNTAX:

```
WHILE condition
  Commands
WEND
```

DESCRIPTION:

The commands contained in the **WHILE**..**WEND** loop are continuously executed until the condition becomes **FALSE**. Execution then continues after the **WEND**. If the condition is false when the **WHILE** is first executed then the loop will be skipped.

PARAMETERS:

condition:	Any valid logical TrioBASIC expression
commands:	TrioBASIC statements that you wish to execute

EXAMPLE:

While input 12 is off, move the base axis and flash an LED on output 10

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

```
    WAIT IDLE
    OP(10,ON)
  WEND
```

WORLD_DPOS

TYPE:

Axis Parameter (Read Only)

DESCRIPTION:

The **WORLD_DPOS** is the demand position in the **FRAME** coordinate system. It sits between the **DPOS** and **AXIS_DPOS**.

With no **USER_FRAME** or **TOOL_OFFSET**, **WORLD_DPOS** is equal to **DPOS**. With no **FRAME**, **WORLD_DPOS** is equal to **AXIS_DPOS**. 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. In this situation when **FRAME** is not zero **WORLD_DPOS** returns the demand position for the programming point of the **FRAME**.

WORLD_DPOS can be scaled by **UNITS**

VALUE:

Demand position in user units of the **FRAME** programming point.

EXAMPLE:

Read the world demand position for axis 10 in user units

```
>>PRINT WORLD_DPOS AXIS(10)
5432
>>
```

SEE ALSO:

AXIS_DPOS, DPOS, FRAME, TOOL_OFFSET, USER_FRAME

XOR

TYPE:

Logical and Bitwise operator

SYNTAX:

```
<expression1> XOR <expression2>
```

DESCRIPTION:

This performs and exclusive or function between corresponding bits of the integer part of two valid TrioBASIC expressions. It may therefore be used as either a bitwise or logical condition.

The XOR function between two values is defined as follows:

XOR	0	1
0	0	1
1	1	0

PARAMETERS:

expression1:	Any valid TrioBASIC expression
expression2:	Any valid TrioBASIC expression

EXAMPLE:

```
    a = 10 XOR (2.1*9)
```

TrioBASIC 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.

ZIP_READ

TYPE:

Command

SYNTAX:

```
ZIP_READ(function ,...)
```

DESCRIPTION:

This function will read a compressed file into RAM on the *Motion Coordinator* and then decompress it in blocks.

The file must be transferred to the SD card on the *Motion Coordinator* using the TextFileLoader (executable or ActiveX) with compression enabled and decompression disabled, that way the file will be stored in compressed format.

Internally we handle two buffer areas: compressed buffer and decompressed buffer. The compressed buffer is filled from the file, the decompressed buffer is filled from the compressed buffer. The data is transferred between the buffers when required.

PARAMETERS:

function:	description:
0	Initialise the **ZIP_READ** resources.
1	Release all the **ZIP_READ** resources.
2	Transfer a block of data from the decompressed buffer to **VR** or **TABLE** memory.
3	Skip a number of bytes in the decompressed buffer.
4	Read buffer indices.
5	Move to a position in the decompressed buffer.
6	Decompress the next buffer.

FUNCTION = 0:

SYNTAX:

value = **ZIP_READ**(0,"filename"[,decompress_block_size[,decompress_block_count]])

DESCRIPTION:

This function initialises the **ZIP_READ** resources.

Due to the size of the internal decompression data structures both the **TEXT_FILE_LOADER** and the **ZIP_READ** commands share the same data structure. This means that if the **TEXT_FILE_LOADER** is decompressing data then the **ZIP_READ** function will fail, and vice versa the **TEXT_FILE_LOADER** decompression will fail if the **ZIP_READ** function is running. This should not be a problem as the **TEXT_FILE_LOADER** must not decompress files that will be processed by the **ZIP_READ** command.

The file is decompressed in blocks. By default there is one 32 KB block. This decompress_block_size parameter allows the block size to be reduced. The block size will be rounded down to the nearest power of 2.

If decompress_block_count is greater than 1 then the **ZIP_READ** will perform double buffering. This means that one process may be decompressing the file whilst another process is using the decompressed data. The total amount of decompressed data is limited to 32 KB so the number of available decompression blocks is limited by the decompress_block_size

PARAMETERS:

value:	0	The initialisation failed
	1	The initialization succeeded but the complete compressed file could not be loaded into memory. This means that at some point another buffer will need to be read. This buffer read can take an appreciable time so a double buffering scheme might be required.
	2	The initialisation succeed and the complete compressed file was loaded into memory.
Filename:	Name of the file on the SD card to be opened.	
decompress_block_size:	2 – 32768 (default =32768)	
decompress_block_count:	1 - (32768 / decompress_block_size)	

EXAMPLE:

```
IF ZIP_READ(0,”myfile.tfl”,2048)=0 THEN
    PRINT “Error initialising reader”
    STOP
ENDIF
```

FUNCTION = 1:

SYNTAX:

`ZIP_READ(1)`

DESCRIPTION:

Frees all the resources held by the **ZIP_READ** command.

EXAMPLE:

```
IF ZIP_READ(0,”myfile.tfl”,2048)=0 THEN
    PRINT “Error initialising reader”
    STOP
ENDIF
ZIP_READ(1)
```

FUNCTION = 2:

SYNTAX:

```
value=ZIP_READ(2,format,destination,start,length)
```

DESCRIPTION:

This function reads a block of data from the decompressed buffer into **VR** or **TABLE** memory. If there is not enough decompressed data available then more data will be decompressed.

PARAMETERS:

value:	Number of values stored	
format:	0	8 bit integer (**ASCII** character data)
	1	16 bit integer (little endian)
	2	16 bit integer (big endian)
	3	32 bit integer (little endian)
	4	32 bit integer (big endian)
	5	64 bit integer (little endian)
	6	64 bit integer (big endian)
	7	32 bit float (little endian)
	8	32 bit float (big endian)
	9	64 bit float (little endian)
	10	64 bit float (big endian)
destination:	0	Store data in **TABLE**
	1	Store data in **VR**
start:	0 ≤start	Position in the destination memory area at which to start storing the data.
length:	Number of values to store. The number of bytes processed will depend on the format, for example if format is 7 then a length of 100 will process 400 bytes	

If the return value this is less than the length parameter then we have reached the end of the file and any further reads will cause a TrioBASIC error.

EXAMPLE:

```
    IF ZIP_READ(0,"myfile.tfl",2048)=0 THEN
        PRINT "Error initialising reader"
        STOP
    ENDIF
    REPEAT
```

```
        c=ZIP_READ(2,0,0,1000,50)
    UNTIL c<50
    ZIP_READ(1)
```

FUNCTION = 3:

SYNTAX:

`value=ZIP_READ(3,length)`

DESCRIPTION:

This function skips a number of bytes in the decompressed buffer.

PARAMETERS:

value:	The number of bytes skipped
length:	The number of bytes to skip.

If the return value this is less than the length parameter then we have reached the end of the file and any further reads will cause a TrioBASIC error.

EXAMPLE:

```
    IF ZIP_READ(0,"myfile.tfl",2048)=0 THEN
        PRINT "Error initialising reader"
        STOP
    ENDIF
    ZIP_READ(3,23)
    REPEAT
        c=ZIP_READ(2,0,0,1000,50)
    UNTIL c<50
    ZIP_READ(1)
```

FUNCTION = 4:

SYNTAX:

`value=ZIP_READ(4,index)`

DESCRIPTION:

This function returns the value of the internal buffer indices.

PARAMETERS:

value:	The value of the specified index.	
index:	0	compressed buffer offset
	1	compressed buffer length
	2	compressed file offset
	3	uncompressed buffer offset
	4	uncompressed buffer length
	5	uncompressed file offset

EXAMPLE:

```
IF ZIP_READ(0,"myfile.tfl",2048)=0 THEN
    PRINT "Error initialising reader"
    STOP
ENDIF
ZIP_READ(3,23)
REPEAT
    c=ZIP_READ(2,0,0,1000,50)
    PRINT "Compressed file indices: ";
    PRINT ZIP_READ(4,0),ZIP_READ(4,1),ZIP_READ(4,2)
    PRINT "Decompressed file indices: ";
    PRINT ZIP_READ(4,3),ZIP_READ(4,4),ZIP_READ(4,5)
UNTIL c<50
ZIP_READ(1)
```

FUNCTION = 5:

SYNTAX:

`value=ZIP_READ(5[,position])`

DESCRIPTION:

This function sets the absolute decompressed file position. If the optional position parameter is not specified then the default value of 0 is used.

PARAMETERS:

value:	The absolute position of the decompressed file or -1 if there is an error
position:	The absolute position in the decompressed file.

If the return value this is less than the length parameter then we have reached the end of the file and any further reads will cause a TrioBASIC error.

EXAMPLE:

```
IF ZIP_READ(0,"myfile.tfl",2048)=0 THEN
    PRINT "Error initialising reader"
    STOP
ENDIF
ZIP_READ(3,23)
VR(100)=-1
REPEAT
    IF VR(100)>=0 THEN ZIP_READ(5,VR(100)):VR(100)=-1
    c=ZIP_READ(2,0,0,1000,50)
    PRINT "Compressed file indices: ";
    PRINT ZIP_READ(4,0),ZIP_READ(4,1),ZIP_READ(4,2)
    PRINT "Decompressed file indices: ";
    PRINT ZIP_READ(4,3),ZIP_READ(4,4),ZIP_READ(4,5)
UNTIL c<50
ZIP_READ(1)
```

FUNCTION = 6:

SYNTAX:

`value=ZIP_READ(6)`

DESCRIPTION:

This function decompresses the next buffer. This is only applicable when the decompress_buffer_count is greater than 1.

PARAMETERS:

value:	The absolute position of the decompressed file or -1 if there is an error.
position	The absolute position in the decompressed file.

If the return value this is less than the length parameter then we have reached the end of the file and

any further reads will cause a TrioBASIC error.

EXAMPLE:

```
IF ZIP_READ(0,"myfile.tfl",2048,2)=0 THEN
    PRINT "Error initialising reader"
    STOP
```

```
ENDIF
ZIP_READ(3,23)
VR(100)=-1
REPEAT
    IF VR(100)>=0 THEN ZIP_READ(5,VR(100)):VR(100)=-1
    IF 2048-ZIP_READ(4,3)<50 THEN ZIP_READ(6)
    c=ZIP_READ(2,0,0,1000,50)
    PRINT “Compressed file indices: “;
    PRINT ZIP_READ(4,0),ZIP_READ(4,1),ZIP_READ(4,2)
    PRINT “Decompressed file indices: “;
    PRINT ZIP_READ(4,3),ZIP_READ(4,4),ZIP_READ(4,5)
UNTIL c<50
ZIP_READ(1)
```

TRIO IEC 61131-3 MOTION LIBRARY

3

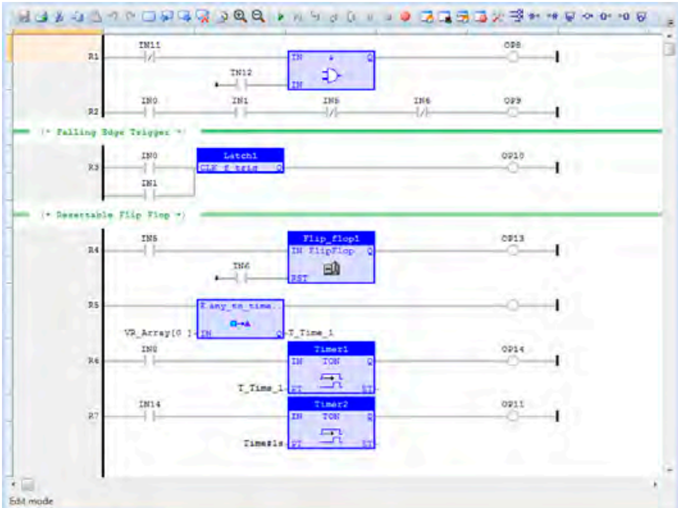
Contents

Introduction to the Trio IEC Motion Library

MC4xx IEC 61131-3 overview

In addition to the well-established Trio **BASIC** programming language, the MC4xx range introduces the possibility to design programs using the international standard IEC 61131-3 language for industrial controls.

Motion Perfect version 3 comes complete with editors for the 4 methods supported; Ladder (LD), Structured Text (ST), Function Block Diagram (FBD) and Sequential Function Chart (SFC). The use of the Motion Perfect v3 editor is covered in the Motion Perfect section of the manual. Motion Perfect v3 compiles the IEC 61131-3 programs and loads the compiled code into the *Motion Coordinator*. The code is run in the MC4xx by run-time execution software which operates in parallel to the Trio **BASIC** run-time environment. Therefore, both programming systems can be used together within the same project, on the same *Motion Coordinator*.



The main functions of the IEC 61131-3 languages follow the standard. So a programmer already familiar with IEC 61131-3 will be able to start creating programs with ease. The only new features a programmer needs to learn is how to work within the Motion Perfect v3 environment. The IEC 61131-3 editor and toolbox allows for rapid development of standard programs. Inputs, Ouputs, VRs and **TABLE** can all be bound to named IEC 61131-3 named variables, giving access from any programming method to the MC4xx IO space.

IEC 61131-3 Motion Library

The motion functions provided in the MC4xx range are the many functions which have been developed over years of putting *Motion Coordinator*s into service on machines of all types. They cover the whole range of motion from simple point-to-point moves, through multi-axis interpolated motion, gearing and linked moves, to sophisticated robotics. Application areas include cutting, gluing, packaging machines, printing machines, pick and place, and production lines of all kinds.

The MC4xx motion library will be immediately recognised by programmers who have used Trio's **BASIC** language. Although it is not a strict match for the PLC Open-Motion part of IEC 61131-3, it does have many parallel move types which can be used in place of the standard functions. What is more, the MC4xx motion library has the full set of Trio motion functions which have been proven to enable complex axis synchronisation to be achieved in a very straight-forward way. Setting up complex, repeatable motion in a very short time is now available in the IEC 61131-3 languages.

FUNCTION BLOCKS

Each Trio Motion function is available as a function block. The function blocks can be added to any of the 4 supported programming methods, including Ladder (LD). Function blocks run either when an enable input is set to **TRUE**, or are triggered by a rising edge on the Execute input. For example, a TC_MOVE1 function block may be set up with the axis number set on one input and the move distance set on the second input. The move only starts when the Execute input changes from **FALSE** to **TRUE**.

In the IEC 61131-3 programming system, the program is continuously scanned. Therefore it is not possible to have the equivalent of a **WAIT IDLE** that is commonly used in **BASIC**. Each function block therefore has a number of outputs which can be used to determine whether the move is buffered, running, completed or if there was an error. The common outputs are:

BUSY:

This **BOOL** output is **TRUE** after the Execute input has triggered the function. It goes back to **FALSE** once the motion function has completed.

DONE:

This **BOOL** output goes **TRUE** after the motion function has been completed normally.

BUFFERED:

This **BOOL** output is **TRUE** to show that the motion command is waiting in **NTYPE** buffer.

ACTIVE:

This **BOOL** output is **TRUE** when the motion command is running. i.e. in **MTYPE**.

ABORTED:

This **BOOL** output goes **TRUE** if the motion is terminated due to a **CANCEL** or reaching an end-limit. It indicates that the motion did not run to completion.

ERROR:

This **BOOL** output is set **TRUE** if a program error is detected. For example if an input value is out of range.

ERRORID:

An **UINT** value which gives the error number. This value is available when the Error output is **TRUE**. The meaning of the ErrorID value is the same as a Trio **BASIC** run-time error value.

FUNCTION BLOCK DESCRIPTIONS

Each function block is described in the usual format for IEC 61131-3 library components. The details are limited to those required in order to add the function block to a program. For a full description of the associated motion command, see the Trio **BASIC** commands in chapter 2. Function block **TC_MOVELINK**, for example, has the same operation as the Trio **BASIC MOVELINK** command, and the entry in chapter 2 includes examples of how it may be used.

TC_ADDAX

TYPE:
Motion Function.

FUNCTION:
Applies a new **ADDAX** request for the axis specified by 'AxisNo'.

INPUTS:

EN : **BOOL**;	Set **TRUE** to enable the function
AxisNo : **USINT**;	Axis number
AxisToAdd : **USINT**;	Axis number of the axis to add to AxisNo

OUTPUTS:

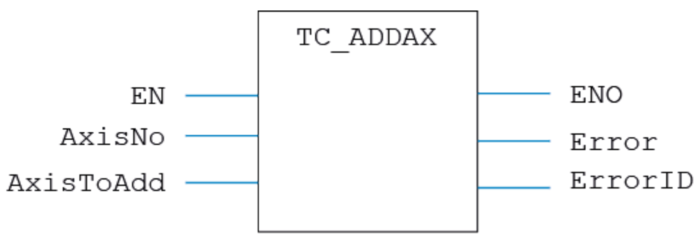
ENO : **BOOL**;	**TRUE** if function is enabled
Error : **BOOL**;	**TRUE** if a program error is detected
ErrorID : **UINT**;	Returned error number

DESCRIPTION:
When the EN input is **TRUE**, the function block applies the **ADDAX** command to the axis indicated by AxisNo. The axis number of the axis to add is taken from the AxisToAdd input. If the AxisToAdd is -1, then the Addax axis connection is terminated.

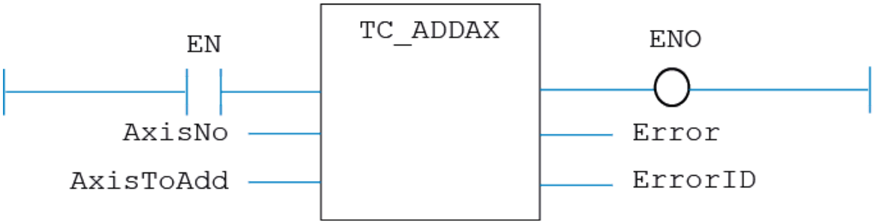
A programming error, such as parameter out of range, will set the Error output and return an error ID number. For the Error ID reference, see the Trio Programming error list.

ST LANGUAGE:
TC_ADDAX(EN, AxisNo, AxisToAdd, ENO, Error, ErrorID);

FBD LANGUAGE:



LD LANGUAGE:



IL LANGUAGE:
Not available.

TC_ADDDAC

TYPE:
Motion Function.

FUNCTION:
Applies a new **ADDDAC** request for the axis specified by ‘AxisNo’.

INPUTS:

EN : **BOOL**;	Set **TRUE** to enable the function
AxisNo : **USINT**;	Axis number
AxisToAdd : **USINT**;	Axis number of the axis to add to AxisNo

OUTPUTS:

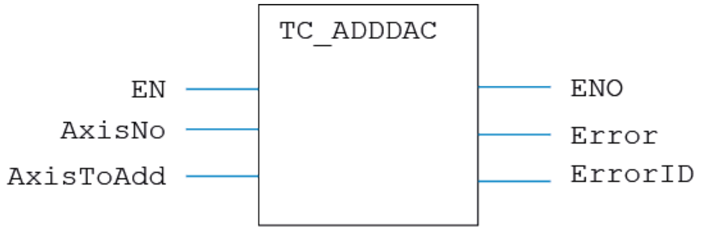
ENO : **BOOL**;	**TRUE** if function is enabled
Error : **BOOL**;	**TRUE** if a program error is detected
ErrorID : **UINT**;	Returned error number

DESCRIPTION:
When the EN input is **TRUE**, the function block applies the **ADDDAC** command to the axis indicated by AxisNo. The axis number of the axis to add is taken from the AxisToAdd input. If the AxisToAdd is -1, then the AddDAC axis connection is terminated.

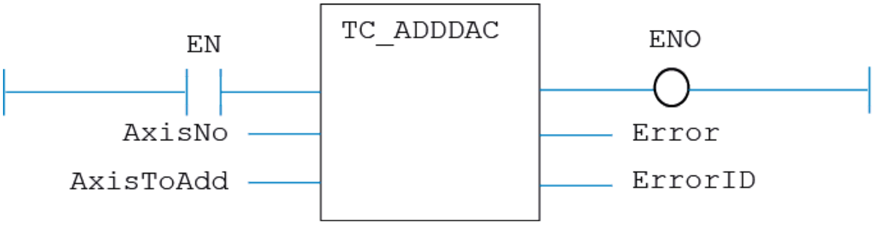
A programming error, such as parameter out of range, will set the Error output and return an error ID number. For the Error ID reference, see the Trio Programming error list.

ST LANGUAGE:
```
TC_ADDDAC(EN, AxisNo, AxisToAdd, ENO, Error, ErrorID);
```

FBD LANGUAGE:



LD LANGUAGE:



IL LANGUAGE:
Not available.

TC_BACKLASH

TYPE:
Motion Function.

FUNCTION:
Issues a new **BACKLASH** motion request for the axis specified by 'AxisNo'.

INPUTS:

Execute : **BOOL**;	Rising edge requests execution
AxisNo : **USINT**;	Axis number

Enable : **BOOL**;	Set **TRUE** to enable the backlash function
Distance : **LINT**;	Backlash distance to apply on direction change
Speed : **LREAL**;	Speed of backlash correction in Units per Second
Accel : **LREAL**;	Acceleration of backlash correction in Units s^-2

OUTPUTS:

Done : **BOOL**;	**TRUE** when function has completed normally
Error : **BOOL**;	**TRUE** if a program error is detected
ErrorID : **UINT**;	Returned error number

DESCRIPTION:

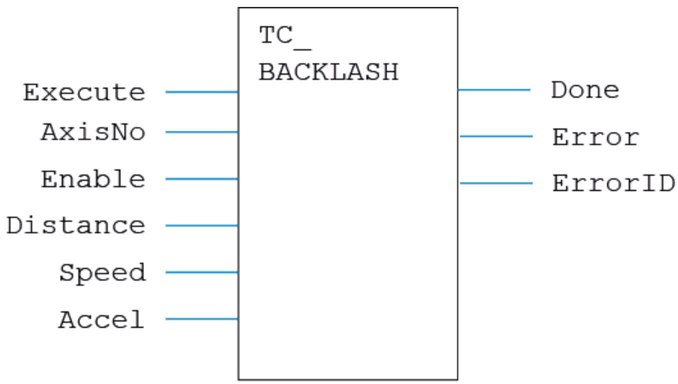
When the execute input changes from **FALSE** to **TRUE** (rising edge), the function block issues the command for execution in the velocity profile software. If the Enable is **TRUE**, the function sets up the Backlash operation using the parameters given. If the Enable is **FALSE** then the Backlash operation is cancelled on the axis defined by AxisNo.

A programming error, such as parameter out of range, will set the Error output and return an error ID number. For the Error ID reference, see the Trio Programming error list.

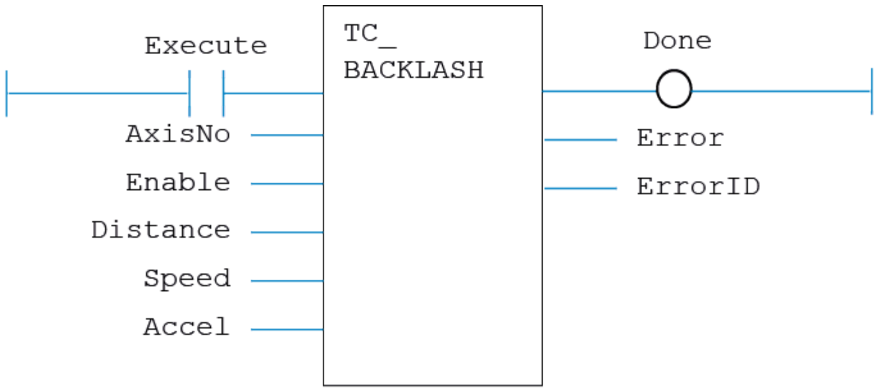
ST LANGUAGE:

```
TC_BACKLASH(Execute, AxisNo, Enable, Distance, Speed, Accel, Done, Error, ErrorID);
```

FBD LANGUAGE:



LD LANGUAGE:



IL LANGUAGE:
Not available.

TC_BASE

TYPE:
Motion Function.

FUNCTION:
Applies a new **BASE** request for the axis or axes specified by 'AxisNo[]'.

INPUTS:

Execute : **BOOL**;	Rising edge requests execution
Count : **USINT**;	Number of axes specified in the AxisNo array
AxisNo[] : **USINT**[];	Axis number(s) of the axes to use in move commands

OUTPUTS:

Done : **BOOL**;	**TRUE** when function has completed normally
Error : **BOOL**;	**TRUE** if a program error is detected
ErrorID : **UINT**;	Returned error number

DESCRIPTION:

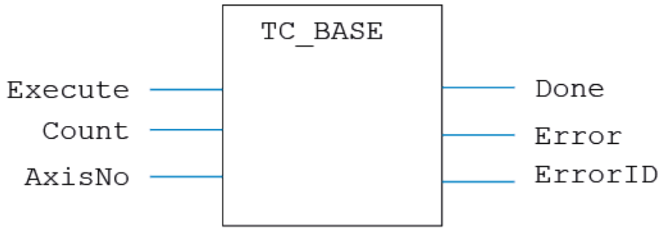
When the Execute input changes from **FALSE** to **TRUE** (rising edge), the function block issues the command for execution in the velocity profile software. The axis numbers in the array AxisNo become the axes to be moved in any profiled move that is executed after the **TC_BASE**.

A programming error, such as parameter out of range, will set the Error output and return an error ID number. For the Error ID reference, see the Trio Programming error list.

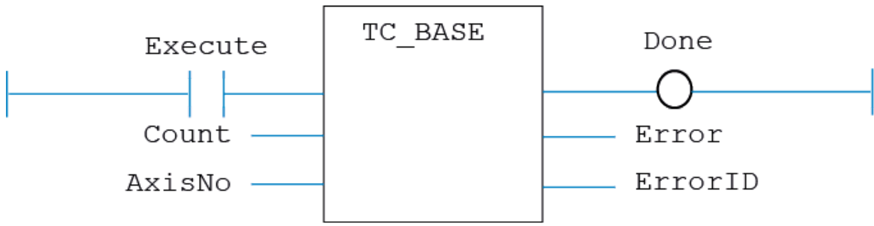
ST LANGUAGE:

TC_BASE(Execute, Count, AxisNo[], Done, Error, ErrorID);

FBD LANGUAGE:



LD LANGUAGE:



IL LANGUAGE:

Not available.

TC_CAM

TYPE:

Motion Function.

FUNCTION:
Issues a new CAM motion request for the axis specified by 'AxisNo'.

INPUTS:

Execute : **BOOL**;	Rising edge requests execution
AxisNo : **USINT**;	Axis number
Start : **LINT**;	Table index for start of Cam data
Stop : **LINT**;	Table index for end of Cam data
Multiplier : **LREAL**;	Output position multiplier
Distance : **LREAL**;	Distance parameter for CAM command

OUTPUTS:

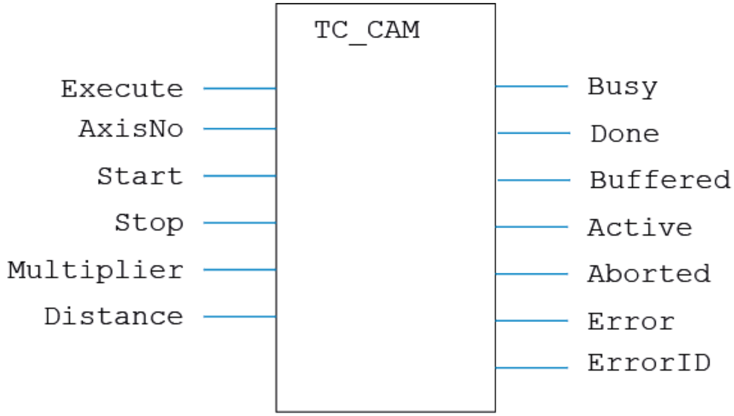
Busy : **BOOL**;	**TRUE** if function is running
Done : **BOOL**;	**TRUE** when function has completed normally
Buffered : **BOOL**;	**TRUE** when motion command is in **NTYPE** buffer
Active : **BOOL**;	**TRUE** when motion command is in **MTYPE** buffer
Aborted : **BOOL**;	**TRUE** if function terminates due to **CANCEL**
Error : **BOOL**;	**TRUE** if a program error is detected
ErrorID : **UINT**;	Returned error number

DESCRIPTION:
When the execute input changes from **FALSE** to **TRUE** (rising edge), the function block attempts to load the motion command into the required axis buffer. If the buffer is unavailable, the function re-tries on each PLC scan. Once the motion command has been loaded, the appropriate outputs will indicate the state of the motion; in **NTYPE**, **MTYPE**, aborted (Cancelled) or done.

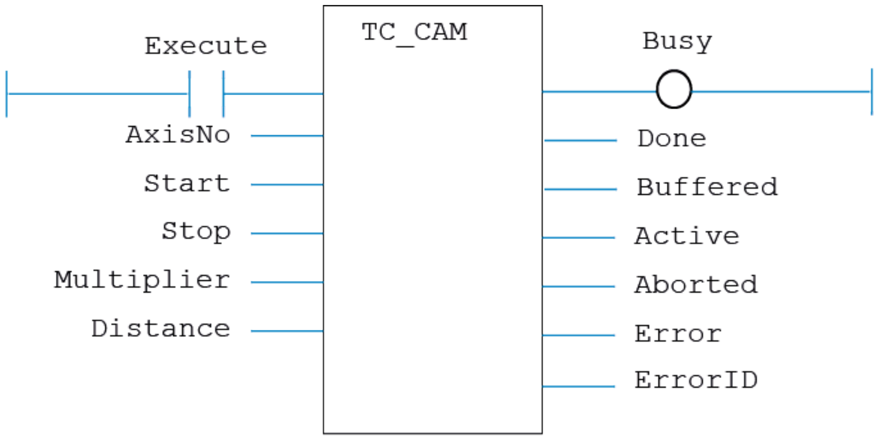
A programming error, such as parameter out of range, will set the Error output and return an error ID number. For the Error ID reference, see the Trio Programming error list.

ST LANGUAGE:
```
TC_CAM(Execute, AxisNo, Start, Stop, Multiplier, Distance, Busy, Done, Buffered,
Active, Aborted, Error, ErrorID);
```

FBD LANGUAGE:



LD LANGUAGE:



IL LANGUAGE:

Not available.

TC_CAMBOX

TYPE:
Motion Function.

FUNCTION:
Issues a new **CAMBOX** motion request for the axis specified by ‘AxisNo’.

INPUTS:

Execute : **BOOL**;	Rising edge requests execution
AxisNo : **USINT**;	Axis number
Start : **LINT**;	Table index for start of Cam data
Stop : **LINT**;	Table index for end of Cam data
Multiplier : **LREAL**;	Output position multiplier
LinkAxis : **USINT**;	Link axis number
LinkDistance : **LREAL**;	Link distance
LinkOptions : **DINT**;	Link options, set to 0 for none
LinkPosition : **LREAL**;	Link Position, set to 0 if unused
LinkOffset : **LREAL**;	Link Offset, set to 0 if unused

OUTPUTS:

Busy : **BOOL**;	**TRUE** if function is running
Done : **BOOL**;	**TRUE** when function has completed normally
Buffered : **BOOL**;	**TRUE** when motion command is in **NTYPE** buffer
Active : **BOOL**;	**TRUE** when motion command is in **MTYPE** buffer
Aborted : **BOOL**;	**TRUE** if function terminates due to **CANCEL**
Error : **BOOL**;	**TRUE** if a program error is detected
ErrorID : **UINT**;	Returned error number

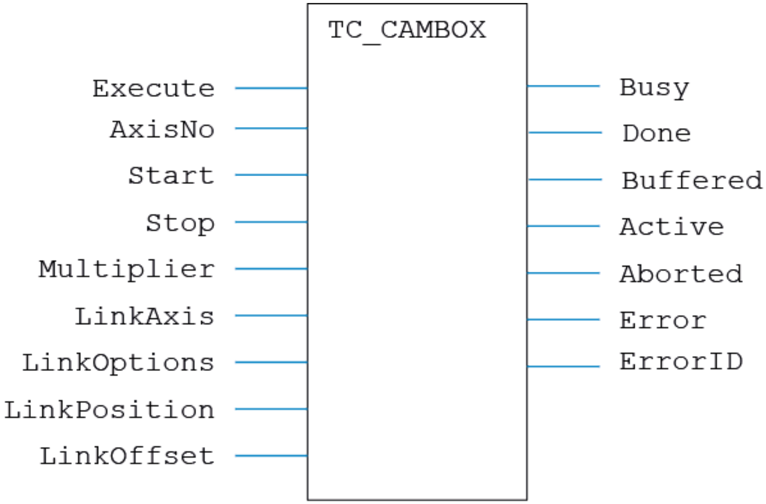
DESCRIPTION:
When the execute input changes from **FALSE** to **TRUE** (rising edge), the function block attempts to load the motion command into the required axis buffer. If the buffer is unavailable, the function re-tries on each PLC scan. Once the motion command has been loaded, the appropriate outputs will indicate the state of the motion; in **NTYPE**, **MTYPE**, aborted (Cancelled) or done.

A programming error, such as parameter out of range, will set the Error output and return an error ID number. For the Error ID reference, see the Trio Programming error list.

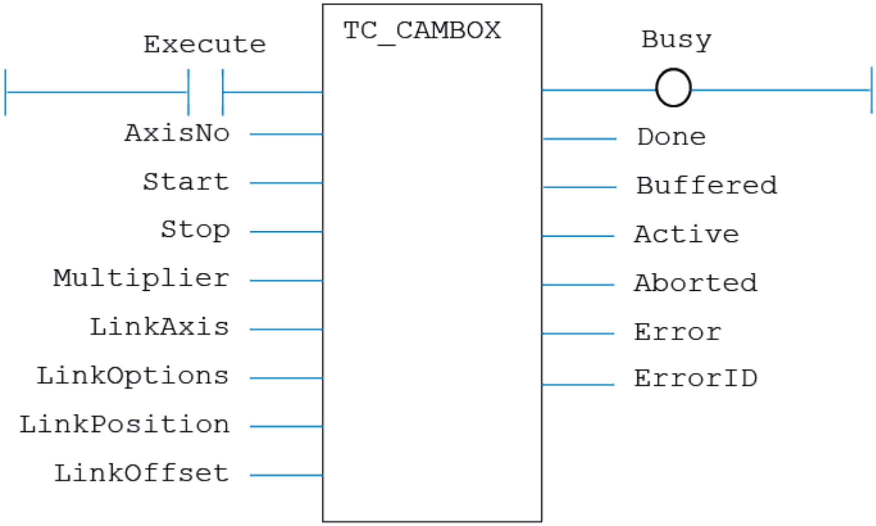
ST LANGUAGE:

```
TC_CAMBOX(Execute, AxisNo, Start, Stop, Multiplier, LinkAxis,  LinkDistance,
LinkOptions, LinkPosition, LinkOffset, Busy, Done, Buffered, Active, Aborted,
Error, ErrorID);
```

FBD LANGUAGE:



LD LANGUAGE:



IL LANGUAGE:
Not available.

TC_CANCEL

TYPE:
Motion Function.

FUNCTION:
Issues a new **CANCEL** motion request for the axis specified by 'AxisNo'.

INPUTS:

Execute : **BOOL**;	Rising edge requests execution
AxisNo : **USINT**;	Axis number
Mode : **BOOL**;	**CANCEL** mode

OUTPUTS:

Busy : **BOOL**;	**TRUE** if function is running

Done : **BOOL**;	**TRUE** when function has completed normally
Buffered : **BOOL**;	**TRUE** when motion command is in **NTYPE** buffer
Active : **BOOL**;	**TRUE** when motion command is in **MTYPE** buffer
Aborted : **BOOL**;	**TRUE** if function terminates due to **CANCEL**
Error : **BOOL**;	**TRUE** if a program error is detected
ErrorID : **UINT**;	Returned error number

DESCRIPTION:

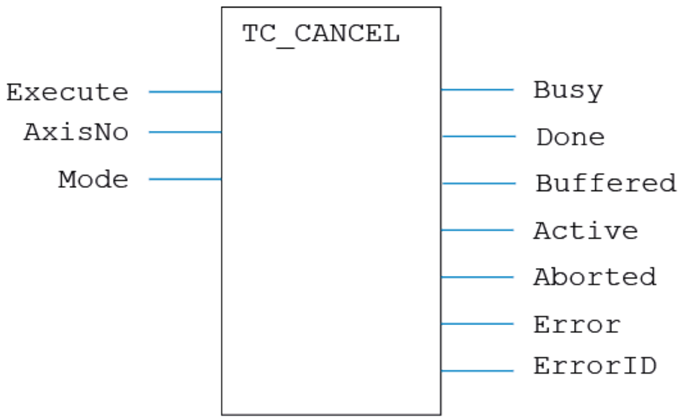
When the execute input changes from **FALSE** to **TRUE** (rising edge), the function block attempts to load the motion command into the required axis buffer. If the buffer is unavailable, the function re-tries on each PLC scan. Once the motion command has been loaded, the appropriate outputs will indicate the state of the motion; in **NTYPE**, **MTYPE**, aborted (Cancelled) or done.

A programming error, such as parameter out of range, will set the Error output and return an error ID number. For the Error ID reference, see the Trio Programming error list.

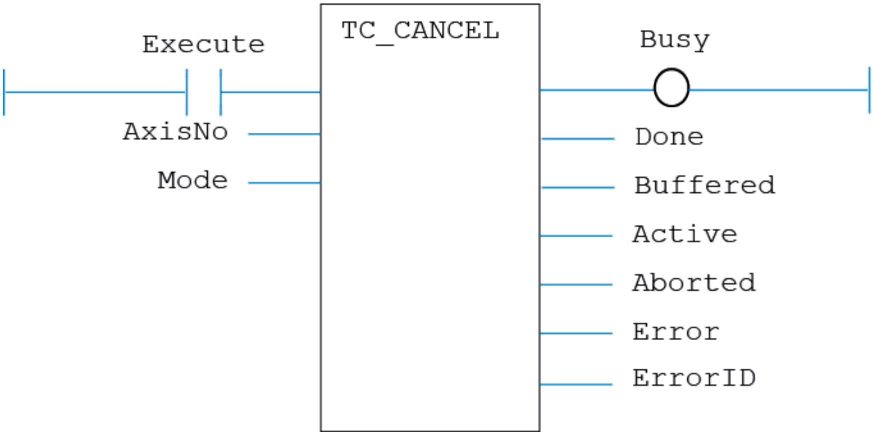
ST LANGUAGE:

TC_CANCEL(Execute, AxisNo, Mode, Busy, Done, Buffered, Active, Aborted, Error, ErrorID);

FBD LANGUAGE:



LD LANGUAGE:



IL LANGUAGE:
Not available.

TC_CONNECT

TYPE:
Motion Function.

FUNCTION:
Issues a new **CONNECT** motion request for the axis specified by 'AxisNo'.

INPUTS:

Execute : **BOOL**;	Rising edge requests execution
AxisNo : **USINT**;	Axis number
LinkAxis : **USINT**;	Link axis number
Ratio : **LREAL**;	Connect ratio: axis_counts/linkaxis_counts

OUTPUTS:

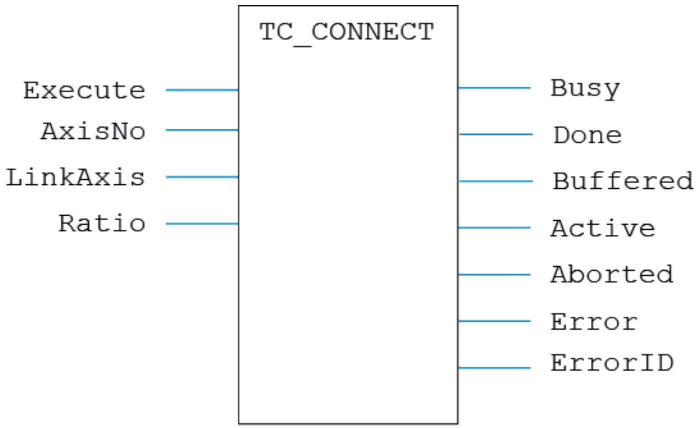
Busy : **BOOL**;	**TRUE** if function is running
Done : **BOOL**;	**TRUE** when function has completed normally
Buffered : **BOOL**;	**TRUE** when motion command is in **NTYPE** buffer
Active : **BOOL**;	**TRUE** when motion command is in **MTYPE** buffer

Aborted : **BOOL**;	**TRUE** if function terminates due to **CANCEL**
Error : **BOOL**;	**TRUE** if a program error is detected
ErrorID : **UINT**;	Returned error number

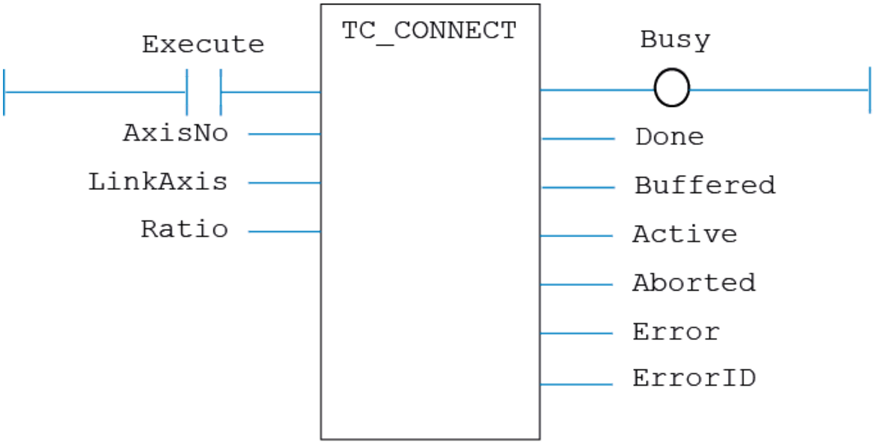
DESCRIPTION:
When the execute input changes from **FALSE** to **TRUE** (rising edge), the function block attempts to load the motion command into the required axis buffer. If the buffer is unavailable, the function re-tries on each PLC scan. Once the motion command has been loaded, the appropriate outputs will indicate the state of the motion; in **NTYPE**, **MTYPE**, aborted (Cancelled) or done.

A programming error, such as parameter out of range, will set the Error output and return an error ID number. For the Error ID reference, see the Trio Programming error list.

ST LANGUAGE:
```
TC_CONNECT(Execute, AxisNo, LinkAxis, Ratio, Busy, Done, Buffered, Active, Aborted,
Error, ErrorID);
```

FBD LANGUAGE:


LD LANGUAGE:



IL LANGUAGE:
Not available.

TC_DATUM

TYPE:
Motion Function.

FUNCTION:
Issues a new **DATUM** motion request for the axis specified by 'AxisNo'.

INPUTS:

Execute : **BOOL**;	Rising edge requests execution
AxisNo : **USINT**;	Axis number
Mode : **DINT**;	Datum sequence number

OUTPUTS:

Busy : **BOOL**;	**TRUE** if function is running
Done : **BOOL**;	**TRUE** when function has completed normally
Buffered : **BOOL**;	**TRUE** when motion command is in **NTYPE** buffer

Active : **BOOL**;	**TRUE** when motion command is in **MTYPE** buffer
Aborted : **BOOL**;	**TRUE** if function terminates due to **CANCEL**
Error : **BOOL**;	**TRUE** if a program error is detected
ErrorID : **UINT**;	Returned error number

DESCRIPTION:

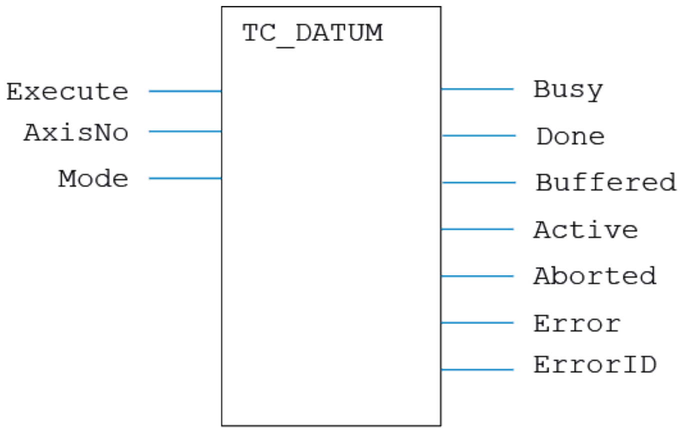
When the execute input changes from **FALSE** to **TRUE** (rising edge), the function block attempts to load the motion command into the required axis buffer. If the buffer is unavailable, the function re-tries on each PLC scan. Once the motion command has been loaded, the appropriate outputs will indicate the state of the motion; in **NTYPE**, **MTYPE**, aborted (Cancelled) or done.

A programming error, such as parameter out of range, will set the Error output and return an error ID number. For the Error ID reference, see the Trio Programming error list.

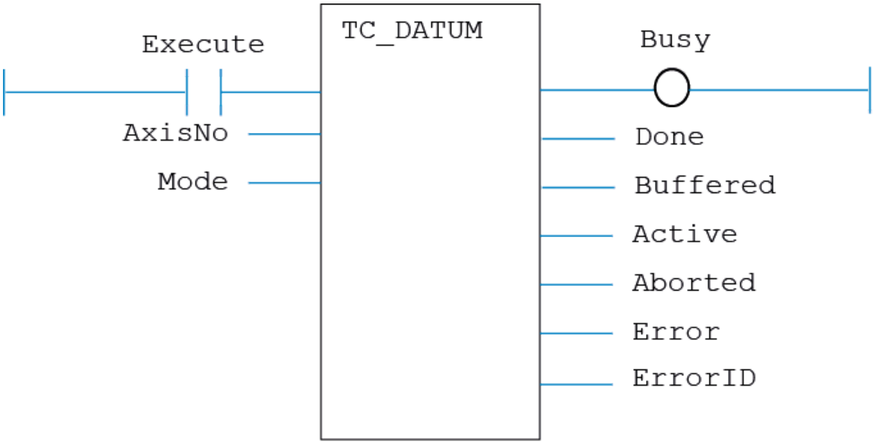
ST LANGUAGE:

```
TC_DATUM(Execute, AxisNo, Mode, Busy, Done, Buffered, Active, Aborted, Error, ErrorID);
```

FBD LANGUAGE:



LD LANGUAGE:



IL LANGUAGE:
Not available.

TC_DEFINETOOLOFFSET

TYPE:
Motion Function.

FUNCTION:
Issues a new **TOOL_OFFSET** definition request for the identity specified by 'ID'.

INPUTS:

EN : **BOOL**;	**TRUE** enables the function
ID : **USINT**;	Identification number for the defined tool offset (0 - 31)
XOFF : **LREAL**;	Offset in the x axis from the world origin to the user origin
YOFF : **LREAL**;	Offset in the y axis from the world origin to the user origin
ZOFF : **LREAL**;	Offset in the z axis from the world origin to the user origin

OUTPUTS:

ENO : **BOOL**;	**TRUE** if function is enabled
Error : **BOOL**;	**TRUE** if a program error is detected
ErrorID : **UINT**;	Returned error number

DESCRIPTION:

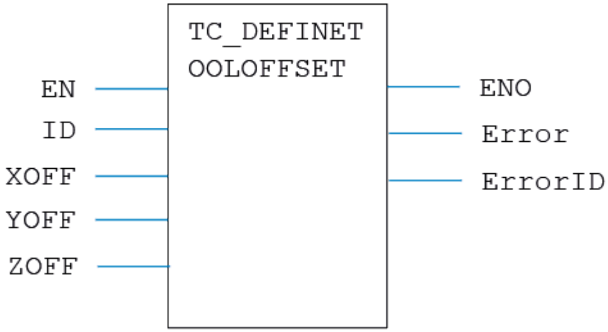
When the EN input is **TRUE**, the function block applies the **TOOL_OFFSET** command to the identity indicated by ID. The offsets are applied to the identity, but are not selected until the **TC_SELECTTOOLOFFSET** is executed.

A programming error, such as parameter out of range, will set the Error output and return an error ID number. For the Error ID reference, see the Trio Programming error list.

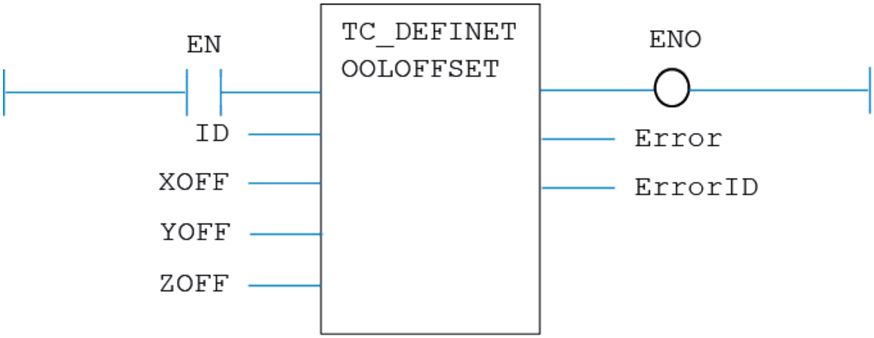
ST LANGUAGE:

TC_DEFINETOOLOFFSET(EN, ID, XOFF, YOFF, ZOFF, ENO, Error, ErrorID);

FBD LANGUAGE:



LD LANGUAGE:



IL LANGUAGE:
Not available.

TC_DEFINEUSERFRAME

TYPE:
Motion Function.

FUNCTION:
Issues a new **USER_FRAME** definition request for the identity specified by 'ID'.

INPUTS:

EN : **BOOL**;	**TRUE** enables the function
ID : **USINT**;	Identification number for the defined tool offset (0 - 31)
XOFF : **LREAL**;	Offset in the x axis from the world origin to the user origin
YOFF : **LREAL**;	Offset in the y axis from the world origin to the user origin
ZOFF : **LREAL**;	Offset in the z axis from the world origin to the user origin
XROT : **LREAL**;	Rotation about the items x axis in radians
YROT : **LREAL**;	Rotation about the items y axis in radians
ZROT : **LREAL**;	Rotation about the items z axis in radians

OUTPUTS:

ENO : **BOOL**;	**TRUE** if function is enabled

Error : **BOOL**;	**TRUE** if a program error is detected
ErrorID : **UINT**;	Returned error number

DESCRIPTION:

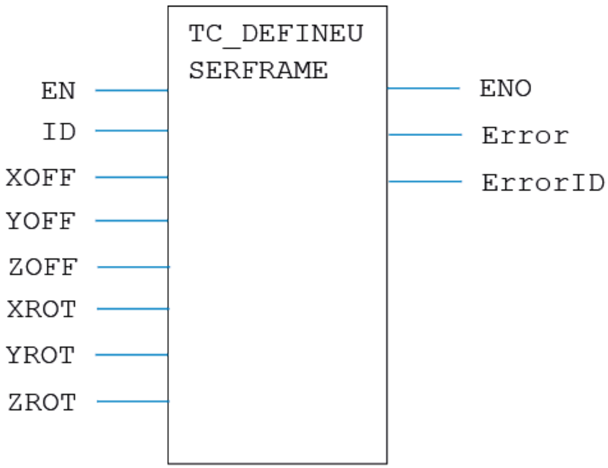
When the EN input is **TRUE**, the function block applies the **USER_OFFSET** command to the identity indicated by ID. The user frame parameters are applied to the identity, but are not selected until the **TC_SELECTUSERFRAME** is executed.

A programming error, such as parameter out of range, will set the Error output and return an error ID number. For the Error ID reference, see the Trio Programming error list.

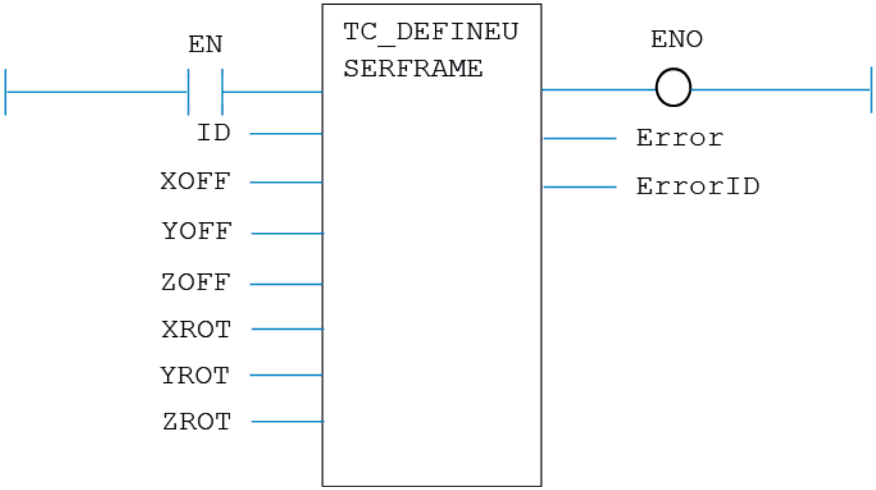
ST LANGUAGE:

```
TC_DEFINUSERFRAME(EN, ID, XOFF, YOFF, ZOFF, XROT, YROT, ZROT,
ENO, Error, ErrorID);
```

FBD LANGUAGE:



LD LANGUAGE:



IL LANGUAGE:
Not available.

TC_DEFPOS

TYPE:
Motion Function.

FUNCTION:
Applies a new **DEFPOS** request for the axis or axes specified by 'AxisNo'.

INPUTS:

Execute : **BOOL**;	Rising edge requests execution
AxisNo : **USINT**;	Axis number of the base axis
Count : **USINT**;	Number of values specified in the Positions array
Positions[] : **LREAL**[];	Array containing the position values to be applied

OUTPUTS:

Done : **BOOL**;	**TRUE** when function has completed normally

Error : **BOOL**;	**TRUE** if a program error is detected
ErrorID : **UINT**;	Returned error number

DESCRIPTION:

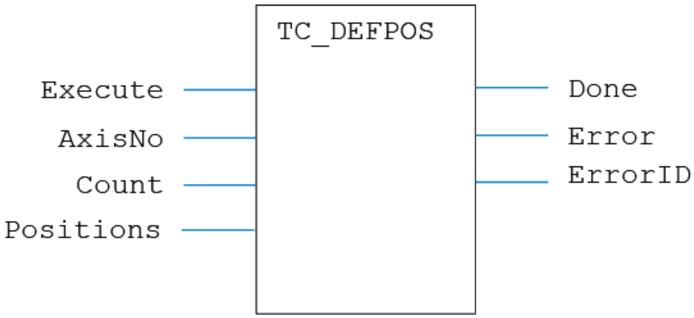
When the Execute input changes from **FALSE** to **TRUE** (rising edge), the function block issues the command for execution in the velocity profile software. The values in the array Positions are applied to Count axes, starting at axis AxisNo.

A programming error, such as parameter out of range, will set the Error output and return an error ID number. For the Error ID reference, see the Trio Programming error list.

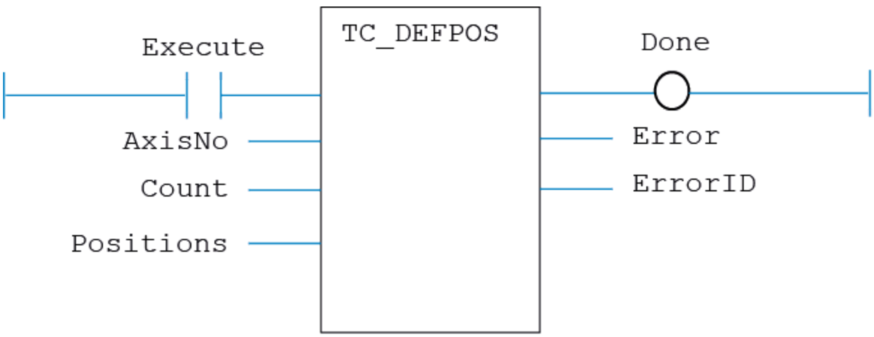
ST LANGUAGE:

```
TC_DEFPOS(Execute, AxisNo, Count, Positions[], Done, Error, ErrorID);
```

FBD LANGUAGE:



LD LANGUAGE:



IL LANGUAGE:
Not available.

TC_DEFPOS1

TYPE:
Motion Function.

FUNCTION:
Applies a new **DEFPOS** request for one axis specified by 'AxisNo'.

INPUTS:

Execute : **BOOL**;	Rising edge requests execution
AxisNo : **USINT**;	Axis number
Pos : **LREAL**;	Position value to be applied

OUTPUTS:

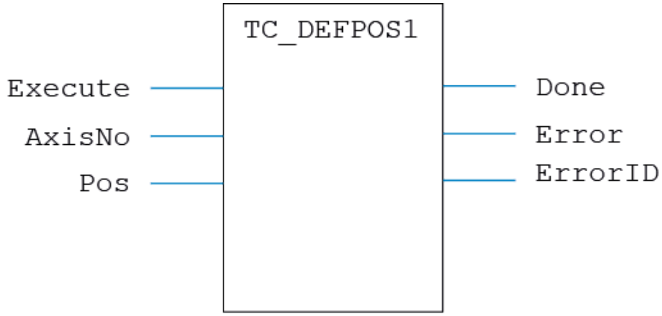
Done : **BOOL**;	**TRUE** when function has completed normally
Error : **BOOL**;	**TRUE** if a program error is detected
ErrorID : **UINT**;	Returned error number

DESCRIPTION:
When the Execute input changes from **FALSE** to **TRUE** (rising edge), the function block issues the command for execution in the velocity profile software. The value in Position is applied to the axis given by AxisNo.

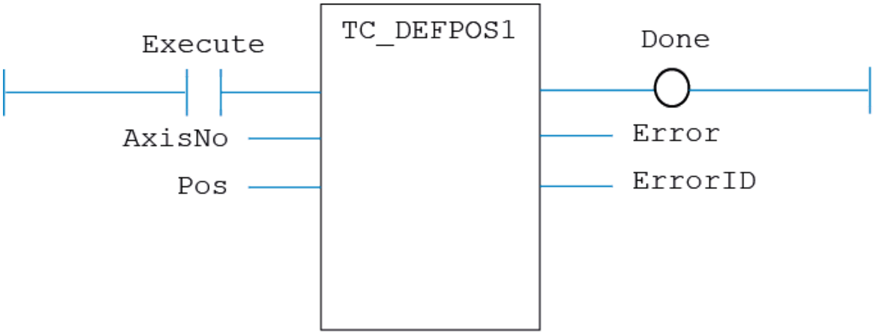
A programming error, such as parameter out of range, will set the Error output and return an error ID number. For the Error ID reference, see the Trio Programming error list.

ST LANGUAGE:
```
TC _ DEFPOS1(Execute, AxisNo, Pos, Done, Error, ErrorID);
```

FBD LANGUAGE:



LD LANGUAGE:



IL LANGUAGE:
Not available.

TC_DEFPOS2

TYPE:
Motion Function.

FUNCTION:
Applies a new **DEFPOS** request for two axes specified by ‘AxisNo’.

INPUTS:

Execute : **BOOL**;	Rising edge requests execution
AxisNo : **USINT**;	Axis number
Pos1 : **LREAL**;	Position value to be applied to first axis
Pos2 : **LREAL**;	Position value to be applied to second axis

OUTPUTS:

Done : **BOOL**;	**TRUE** when function has completed normally
Error : **BOOL**;	**TRUE** if a program error is detected
ErrorID : **UINT**;	Returned error number

DESCRIPTION:

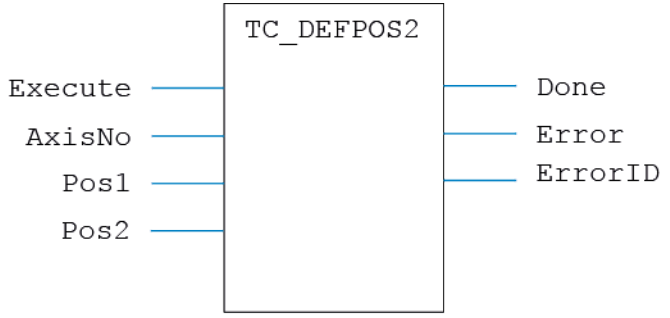
When the Execute input changes from **FALSE** to **TRUE** (rising edge), the function block issues the command for execution in the velocity profile software. The values in Pos1 and Pos2 are applied to the axes starting at AxisNo.

A programming error, such as parameter out of range, will set the Error output and return an error ID number. For the Error ID reference, see the Trio Programming error list.

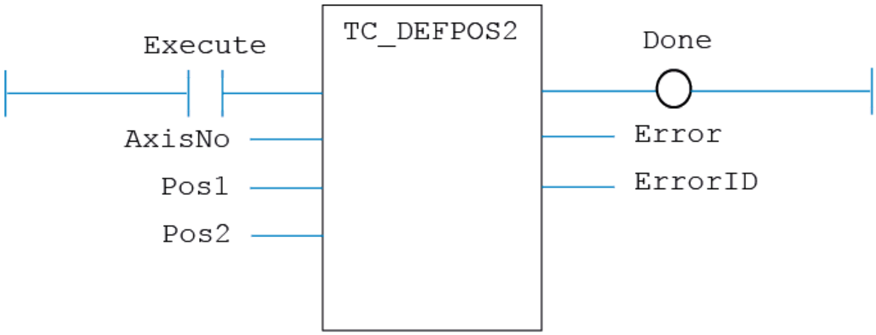
ST LANGUAGE:

```
TC _ DEFPOS2(Execute, AxisNo, Pos1, Pos2, Done, Error, ErrorID);
```

FBD LANGUAGE:



LD LANGUAGE:



IL LANGUAGE:
Not available.

TC_DEFPOS3

TYPE:
Motion Function.

FUNCTION:
Applies a new **DEFPOS** request for three axes specified by 'AxisNo'.

INPUTS:

Execute : **BOOL**;	Rising edge requests execution
AxisNo : **USINT**;	Axis number
Pos1 : **LREAL**;	Position value to be applied to first axis
Pos2 : **LREAL**;	Position value to be applied to second axis
Pos3 : **LREAL**;	Position value to be applied to third axis

OUTPUTS:

Done : **BOOL**;	**TRUE** when function has completed normally
Error : **BOOL**;	**TRUE** if a program error is detected
ErrorID : **UINT**;	Returned error number

DESCRIPTION:

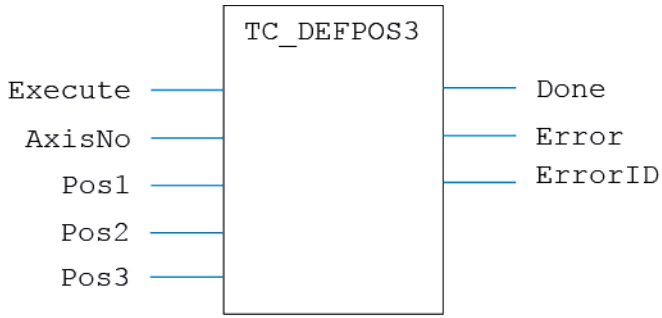
When the Execute input changes from **FALSE** to **TRUE** (rising edge), the function block issues the command for execution in the velocity profile software. The values in Pos1, Pos2 and Pos3 are applied to the axes starting at AxisNo.

A programming error, such as parameter out of range, will set the Error output and return an error ID number. For the Error ID reference, see the Trio Programming error list.

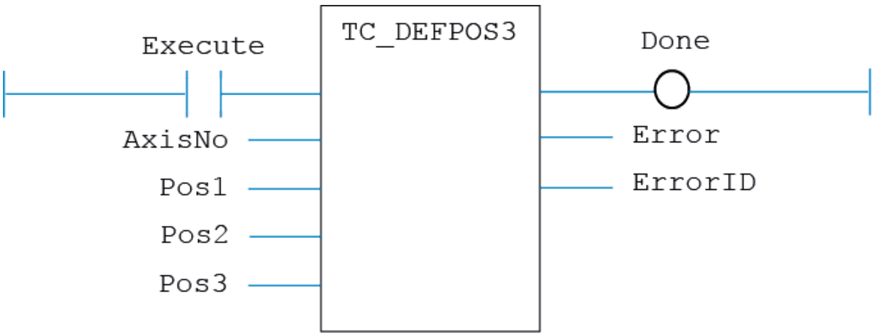
ST LANGUAGE:

`TC _ DEFPOS3(Execute, AxisNo, Pos1, Pos2, Pos3, Done, Error, ErrorID);`

FBD LANGUAGE:



LD LANGUAGE:



IL LANGUAGE:

Not available.

TC_DISABLEGROUP

TYPE:
Motion Function.

FUNCTION:
Applies a new **DISABLE_GROUP** request for the axis or axes specified by 'AxisNo[]'.

INPUTS:

EN : **BOOL**;	**TRUE** to enable the function
AxisCount : **USINT**;	Number of axes specified in the Axes array
Axes[] : **USINT**[];	Axis numbers of the axes to put in the Disable Group

OUTPUTS:

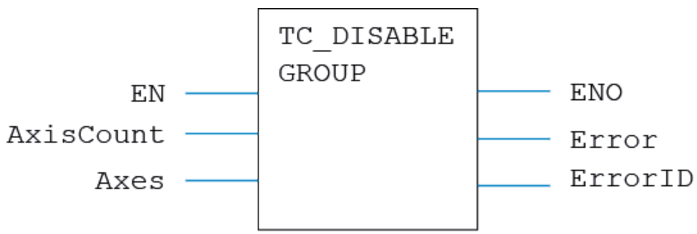
ENO : **BOOL**;	**TRUE** when function is Enabled
Error : **BOOL**;	**TRUE** if a program error is detected
ErrorID : **UINT**;	Returned error number

DESCRIPTION:
When the EN input is **TRUE**, the function block applies the command with the axes indicated.

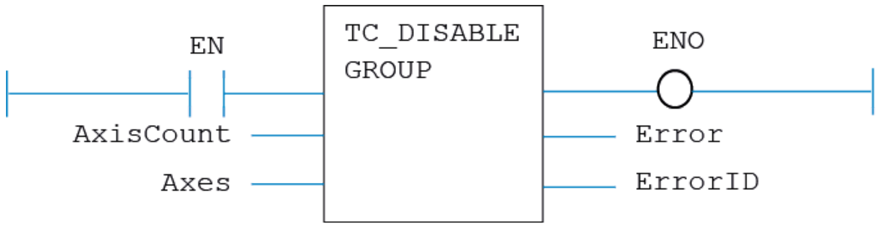
A programming error, such as parameter out of range, will set the Error output and return an error ID number. For the Error ID reference, see the Trio Programming error list.

ST LANGUAGE:
TC_DISABLEGROUP(EN, AxisCount, Axes[], ENO, Error, ErrorID);

FBD LANGUAGE:



LD LANGUAGE:



IL LANGUAGE:

Not available.

TC_ENCODERRATIO

TYPE:

Motion Function.

FUNCTION:

Issues a new **ENCODER_RATIO** motion request for the axis specified by 'AxisNo'.

INPUTS:

EN : **BOOL**;	**TRUE** enables the function
AxisNo **: USINT**;	Axis number
Numerator: **LINT**;	The **MPOS** count (output of the function)
Denominator: **LINT**;	The input count

OUTPUTS:

ENO : **BOOL**;	**TRUE** when function is enabled
Error : **BOOL**;	**TRUE** if a program error is detected
ErrorID : **UINT**;	Returned error number

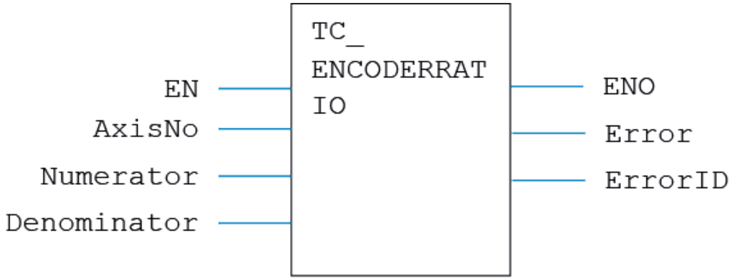
DESCRIPTION:

When the EN input is **TRUE**, the function block applies the command to the axis indicated.

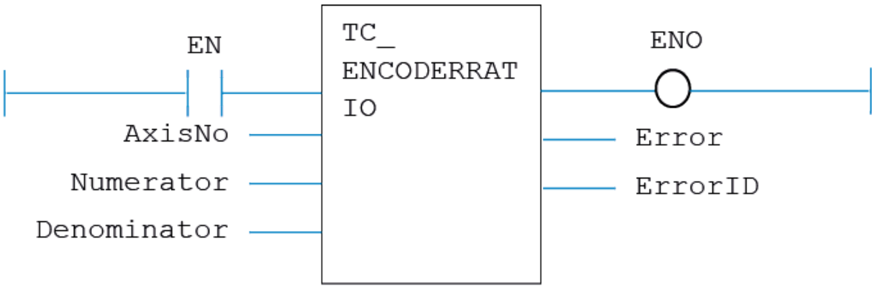
A programming error, such as parameter out of range, will set the Error output and return an error ID number. For the Error ID reference, see the Trio Programming error list.

ST LANGUAGE:
TC_ENCODERRATIO(EN, AxisNo, Numerator, Denominator, ENO, Error, ErrorID);

FBD LANGUAGE:



LD LANGUAGE:



IL LANGUAGE:
Not available.

TC_FORWARD

TYPE:
Motion Function.

FUNCTION:
Issues a new **FORWARD** motion request for the axis specified by 'AxisNo'.

INPUTS:

Execute : **BOOL**;	Rising edge requests execution
AxisNo : **USINT**;	Axis number

OUTPUTS:

Busy : **BOOL**;	**TRUE** if function is running
Done : **BOOL**;	**TRUE** when function has completed normally
Buffered : **BOOL**;	**TRUE** when motion command is in **NTYPE** buffer
Active : **BOOL**;	**TRUE** when motion command is in **MTYPE** buffer
Aborted : **BOOL**;	**TRUE** if function terminates due to **CANCEL**
Error : **BOOL**;	**TRUE** if a program error is detected
ErrorID : **UINT**;	Returned error number

DESCRIPTION:

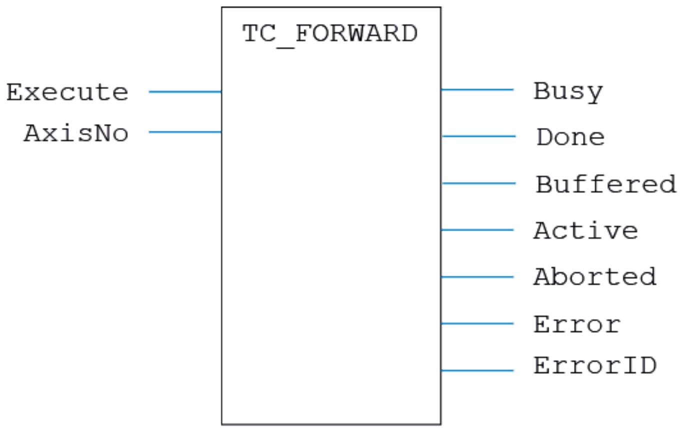
When the execute input changes from **FALSE** to **TRUE** (rising edge), the function block attempts to load the motion command into the required axis buffer. If the buffer is unavailable, the function re-tries on each PLC scan. Once the motion command has been loaded, the appropriate outputs will indicate the state of the motion; in **NTYPE**, **MTYPE**, aborted (Cancelled) or done.

A programming error, such as parameter out of range, will set the Error output and return an error ID number. For the Error ID reference, see the Trio Programming error list.

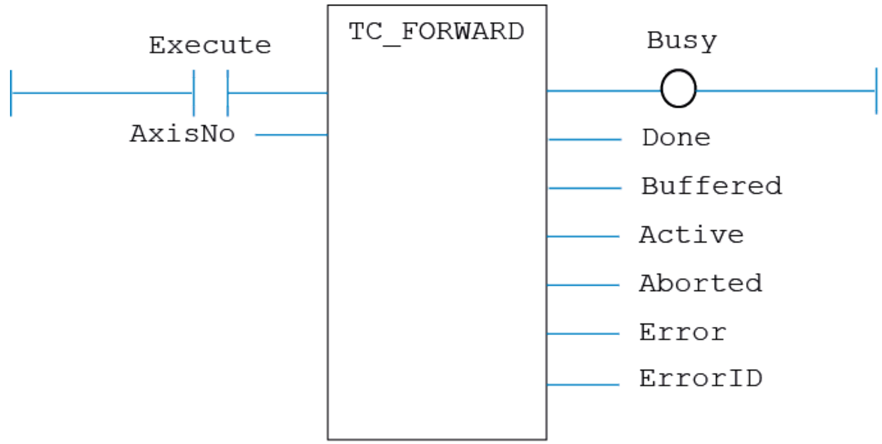
ST LANGUAGE:

```
TC_FORWARD(Execute, AxisNo, Busy, Done, Buffered, Active, Aborted, Error, ErrorID);
```

FBD LANGUAGE:



LD LANGUAGE:



IL LANGUAGE:
Not available.

TC_FRAMEGROUP

TYPE:
Motion Function.

FUNCTION:
Issues a new **FRAME_GROUP** motion request for the axis specified by 'AxisNo'.

INPUTS:

EN : **BOOL**;	**TRUE** to enable the function
ID : **USINT**;	Frame Group Identity number
TableIndex : **DINT**;	Table index points to frame parameters
AxisCount : **USINT**;	Number of axes in Frame Group
Axes[] : **USINT**[];	Array containing the axis numbers

OUTPUTS:

ENO : **BOOL**;	**TRUE** when function is enabled

Error : **BOOL**;	**TRUE** if a program error is detected
ErrorID : **UINT**;	Returned error number

DESCRIPTION:

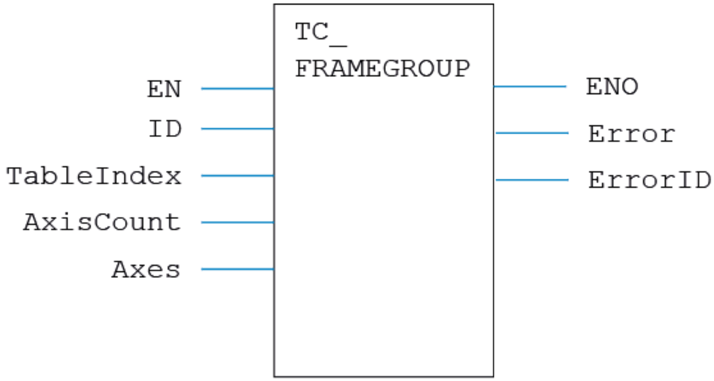
When the EN input is **TRUE**, the function block applies the command to the axis group indicated by AxisNo.

A programming error, such as parameter out of range, will set the Error output and return an error ID number. For the Error ID reference, see the Trio Programming error list.

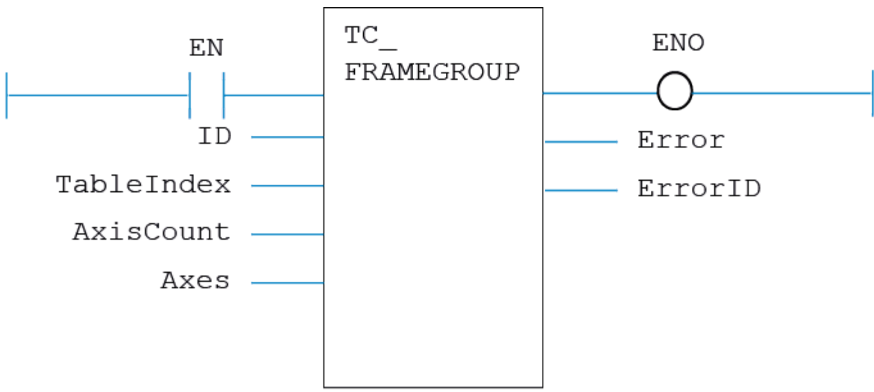
ST LANGUAGE:

`TC_FRAMEGROUP(EN, ID, TableIndex, AxisCount, Axes, ENO, Error, ErrorID);`

FBD LANGUAGE:



LD LANGUAGE:



IL LANGUAGE:
Not available.

TC_FRAMETRANS

TYPE:
Motion Function.

FUNCTION:
Issues a new **FRAME_TRANS** motion request for the axis specified by 'AxisNo'.

INPUTS:

EN : **BOOL**;	**TRUE** to enable the function
Frame : **DINT**;	The **FRAME** number to run
DataIn : **DINT**;	The start position in the **TABLE** of the input positions
DataOut : **DINT**;	The start position in the **TABLE** of the generated positions
Option : **DINT**;	1 = **AXIS_DPOS** to **DPOS** (Forward Kinematics) 0 = **DPOS** to **AXIS_DPOS** (Inverse Kinematics)
TableData : **DINT**;	The first position in the table where the frame configuration is located.

OUTPUTS:

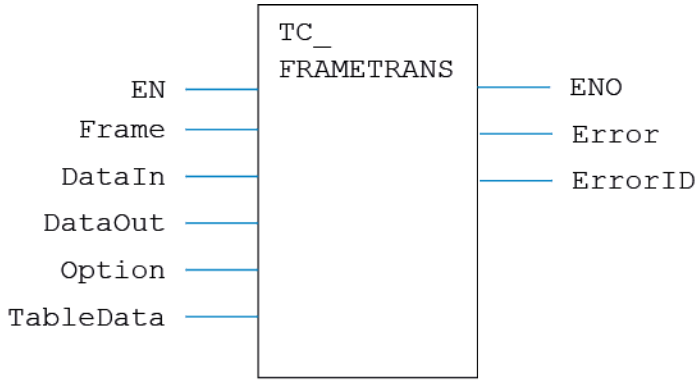
ENO : **BOOL**;	**TRUE** when function is enabled
Error : **BOOL**;	**TRUE** if a program error is detected
ErrorID : **UINT**;	Returned error number

DESCRIPTION:
When the EN input is **TRUE**, the function block applies the command using the frame number indicated by Frame.

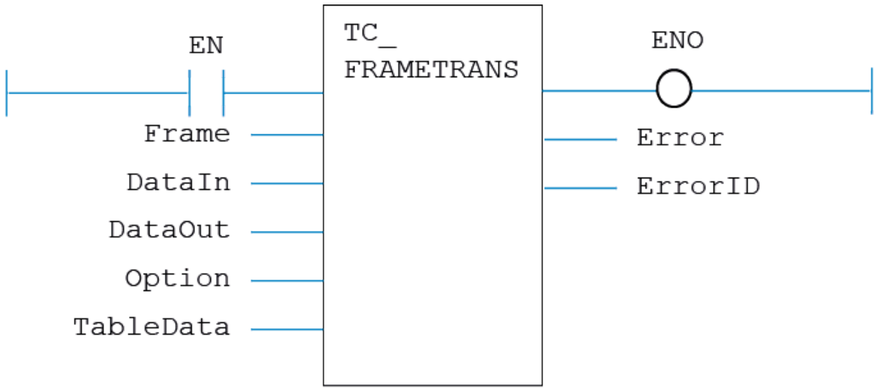
A programming error, such as parameter out of range, will set the Error output and return an error ID number. For the Error ID reference, see the Trio Programming error list.

ST LANGUAGE:
TC_FRAMETRANS(EN, Frame, DataIn, DataOut, Option, TableData, ENO, Error, ErrorID);

FBD LANGUAGE:



LD LANGUAGE:



IL LANGUAGE:
Not available.

TC_GetFRAME

TYPE:
Motion Function.

FUNCTION:
Fetches the currently active **FRAME**.

INPUTS:

EN : **BOOL**;	Set **TRUE** to enable the function
AxisNo : **USINT**;	Axis number

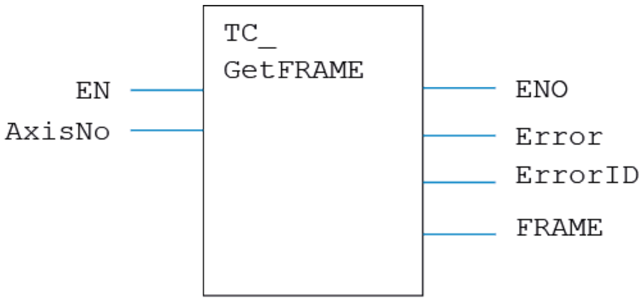
OUTPUTS:

ENO : **BOOL**;	**TRUE** if function is enabled
Error : **BOOL**;	**TRUE** if a program error is detected
ErrorID : **UINT**;	Returned error number
FRAME: **DINT**	The active Frame

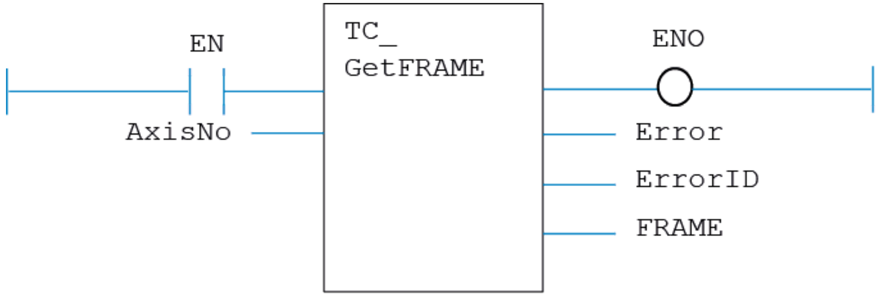
DESCRIPTION:
When the EN input is **TRUE**, the function block applies the command to the axis indicated by AxisNo.

A programming error, such as parameter out of range, will set the Error output and return an error ID number. For the Error ID reference, see the Trio Programming error list.

ST LANGUAGE:
TC _ GetFRAME(EN, AxisNo, ENO, Error, ErrorID, FRAME);

FBD LANGUAGE:


LD LANGUAGE:



IL LANGUAGE:
Not available.

TC_IDLE

TYPE:
Motion Function.

FUNCTION:
Evalues whether the axis is **IDLE** or not.

INPUTS:

EN : **BOOL**;	Set **TRUE** to enable the function
AxisNo : **USINT**;	Axis number

OUTPUTS:

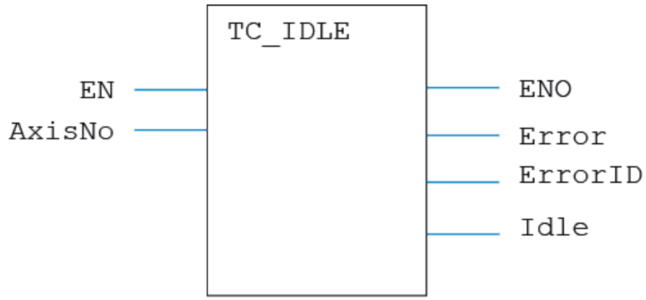
ENO : **BOOL**;	**TRUE** if function is enabled
Error : **BOOL**;	**TRUE** if a program error is detected
ErrorID : **UINT**;	Returned error number
Idle: **BOOL**	**TRUE** when axis is Idle, **FALSE** if motion in progress

DESCRIPTION:
When the EN input is **TRUE**, the function block applies the command to the axis indicated by AxisNo.

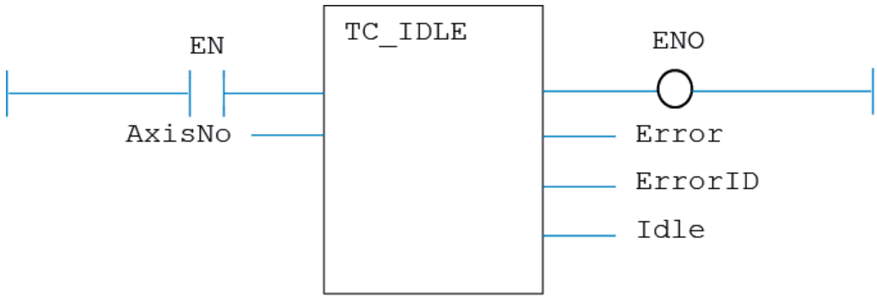
A programming error, such as parameter out of range, will set the Error output and return an error ID number. For the Error ID reference, see the Trio Programming error list.

ST LANGUAGE:
`TC_IDLE(EN, AxisNo, ENO, Error, ErrorID, Idle);`

FBD LANGUAGE:



LD LANGUAGE:



IL LANGUAGE:
Not available.

TC_MOVE

TYPE:
Motion Function.

FUNCTION:
Issues a new **MOVE** motion request for the axis specified by 'AxisNo'.

INPUTS:

Execute : **BOOL**;	Rising edge requests execution
AxisNo : **USINT**;	Axis number of base axis
Count : **USINT**;	Number of axes to be interpolated together
Distances[] : **LREAL**;	Array containing the distances to be moved, one per axis

OUTPUTS:

Busy : **BOOL**;	**TRUE** if function is running
Done : **BOOL**;	**TRUE** when function has completed normally
Buffered : **BOOL**;	**TRUE** when motion command is in **NTYPE** buffer
Active : **BOOL**;	**TRUE** when motion command is in **MTYPE** buffer
Aborted : **BOOL**;	**TRUE** if function terminates due to **CANCEL**
Error : **BOOL**;	**TRUE** if a program error is detected
ErrorID : **UINT**;	Returned error number

DESCRIPTION:

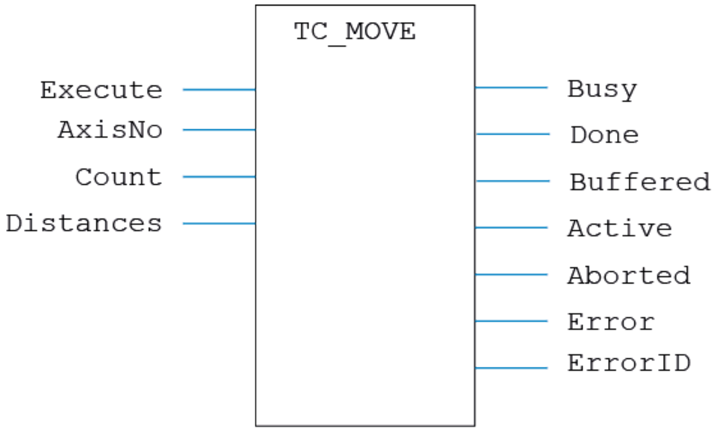
When the execute input changes from **FALSE** to **TRUE** (rising edge), the function block attempts to load the motion command into the required axis buffer. If the buffer is unavailable, the function re-tries on each PLC scan. Once the motion command has been loaded, the appropriate outputs will indicate the state of the motion; in **NTYPE**, **MTYPE**, aborted (Cancelled) or done.

A programming error, such as parameter out of range, will set the Error output and return an error ID number. For the Error ID reference, see the Trio Programming error list.

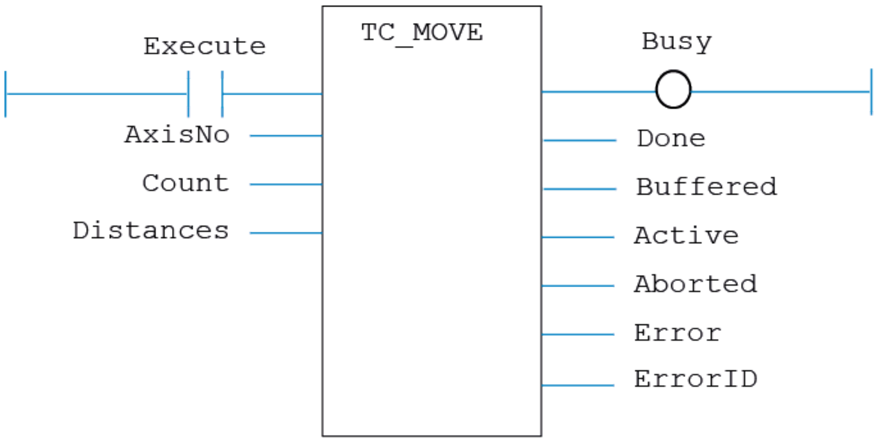
ST LANGUAGE:

```
TC_MOVE(Execute, AxisNo, Count, Distances, Busy, Done, Buffered, Active, Aborted,
Error, ErrorID);
```

FBD LANGUAGE:



LD LANGUAGE:



IL LANGUAGE:

Not available.

TC_MOVE1

TYPE:
Motion Function.

FUNCTION:
Issues a new **MOVE**(Dist) motion request for the axis specified by 'AxisNo'.

INPUTS:

Execute : **BOOL**;	Rising edge requests execution
AxisNo : **USINT**;	Axis number of base axis
Dist : **LREAL**;	The distance to be moved

OUTPUTS:

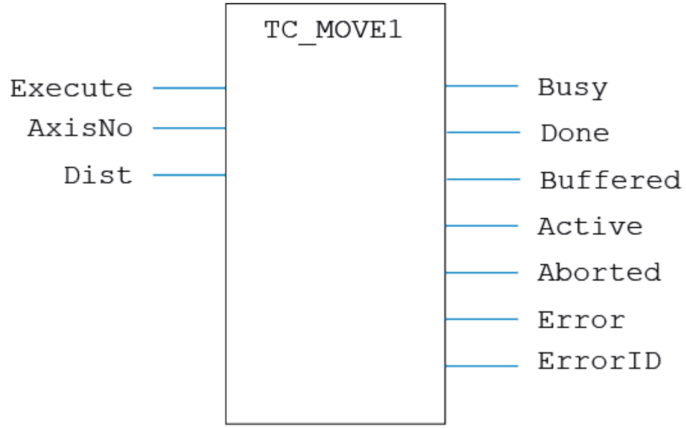
Busy : **BOOL**;	**TRUE** if function is running
Done : **BOOL**;	**TRUE** when function has completed normally
Buffered : **BOOL**;	**TRUE** when motion command is in **NTYPE** buffer
Active : **BOOL**;	**TRUE** when motion command is in **MTYPE** buffer
Aborted : **BOOL**;	**TRUE** if function terminates due to **CANCEL**
Error : **BOOL**;	**TRUE** if a program error is detected
ErrorID : **UINT**;	Returned error number

DESCRIPTION:
When the execute input changes from **FALSE** to **TRUE** (rising edge), the function block attempts to load the motion command into the required axis buffer. If the buffer is unavailable, the function re-tries on each PLC scan. Once the motion command has been loaded, the appropriate outputs will indicate the state of the motion; in **NTYPE**, **MTYPE**, aborted (Cancelled) or done.

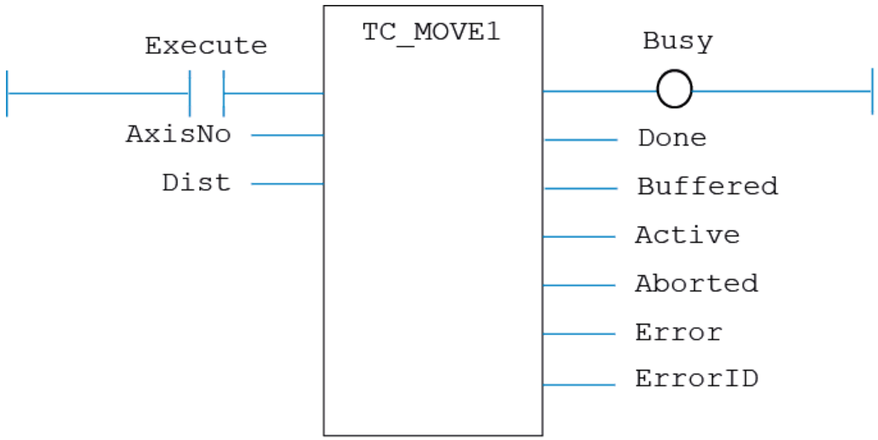
A programming error, such as parameter out of range, will set the Error output and return an error ID number. For the Error ID reference, see the Trio Programming error list.

ST LANGUAGE:
```
TC _ MOVE1(Execute, AxisNo, Dist, Busy, Done, Buffered, Active, Aborted, Error,
ErrorID);
```

FBD LANGUAGE:



LD LANGUAGE:



IL LANGUAGE:

Not available.

TC_MOVE2

TYPE:
Motion Function.

FUNCTION:
Issues a new **MOVE**(Dist1, Dist2) motion request for the pair of axes specified by ‘AxisNo’.

INPUTS:

Execute : **BOOL**;	Rising edge requests execution
AxisNo : **USINT**;	Axis number of base axis
Dist1 : **LREAL**;	Distance to be moved on the first axis
Dist2 : **LREAL**;	Distance to be moved on the second axis

OUTPUTS:

Busy : **BOOL**;	**TRUE** if function is running
Done : **BOOL**;	**TRUE** when function has completed normally
Buffered : **BOOL**;	**TRUE** when motion command is in **NTYPE** buffer
Active : **BOOL**;	**TRUE** when motion command is in **MTYPE** buffer
Aborted : **BOOL**;	**TRUE** if function terminates due to **CANCEL**
Error : **BOOL**;	**TRUE** if a program error is detected
ErrorID : **UINT**;	Returned error number

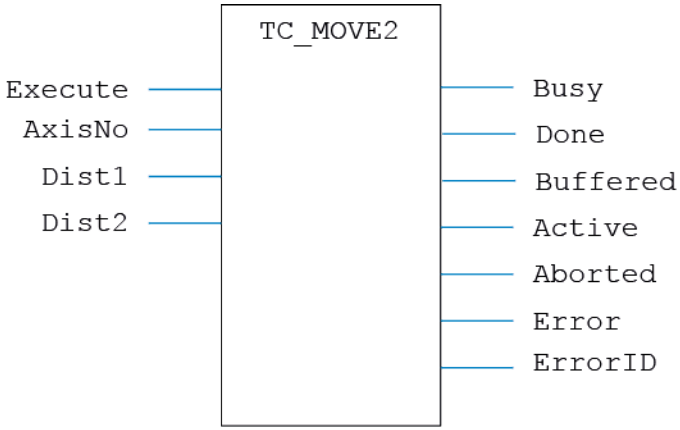
DESCRIPTION:
When the execute input changes from **FALSE** to **TRUE** (rising edge), the function block attempts to load the motion command into the required axis buffer. If the buffer is unavailable, the function re-tries on each PLC scan. Once the motion command has been loaded, the appropriate outputs will indicate the state of the motion; in **NTYPE**, **MTYPE**, aborted (Cancelled) or done.

A programming error, such as parameter out of range, will set the Error output and return an error ID number. For the Error ID reference, see the Trio Programming error list.

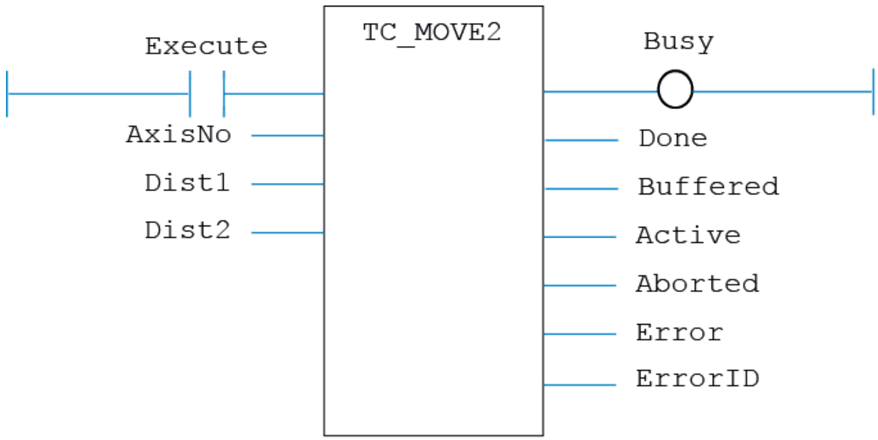
ST LANGUAGE:

```
TC _ MOVE2(Execute, AxisNo, Dist1, Dist2, Busy, Done, Buffered, Active, Aborted,
Error, ErrorID);
```

FBD LANGUAGE:



LD LANGUAGE:



IL LANGUAGE:

Not available.

TC_MOVE3

TYPE:
Motion Function.

FUNCTION:
Issues a new **MOVE**(Dist1, Dist2, Dist3) motion request for the 3 axes specified by 'AxisNo'.

INPUTS:

Execute : **BOOL**;	Rising edge requests execution
AxisNo : **USINT**;	Axis number of base axis
Dist1 : **LREAL**;	Distance to be moved on the first axis
Dist2 : **LREAL**;	Distance to be moved on the second axis
Dist3 : **LREAL**;	Distance to be moved on the third axis

OUTPUTS:

Busy : **BOOL**;	**TRUE** if function is running
Done : **BOOL**;	**TRUE** when function has completed normally
Buffered : **BOOL**;	**TRUE** when motion command is in **NTYPE** buffer
Active : **BOOL**;	**TRUE** when motion command is in **MTYPE** buffer
Aborted : **BOOL**;	**TRUE** if function terminates due to **CANCEL**
Error : **BOOL**;	**TRUE** if a program error is detected
ErrorID : **UINT**;	Returned error number

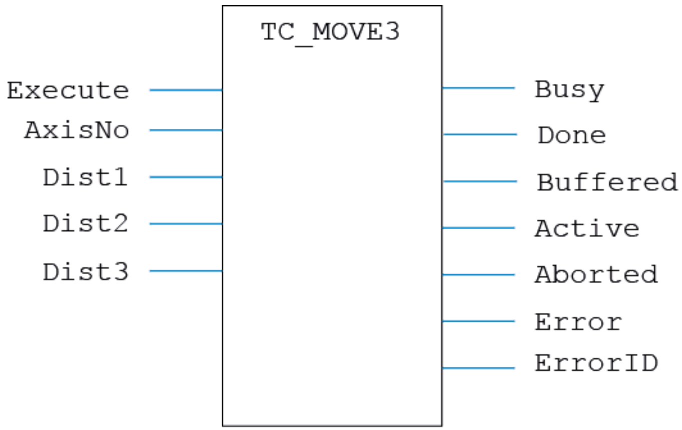
DESCRIPTION:
When the execute input changes from **FALSE** to **TRUE** (rising edge), the function block attempts to load the motion command into the required axis buffer. If the buffer is unavailable, the function re-tries on each PLC scan. Once the motion command has been loaded, the appropriate outputs will indicate the state of the motion; in **NTYPE**, **MTYPE**, aborted (Cancelled) or done.

A programming error, such as parameter out of range, will set the Error output and return an error ID number. For the Error ID reference, see the Trio Programming error list.

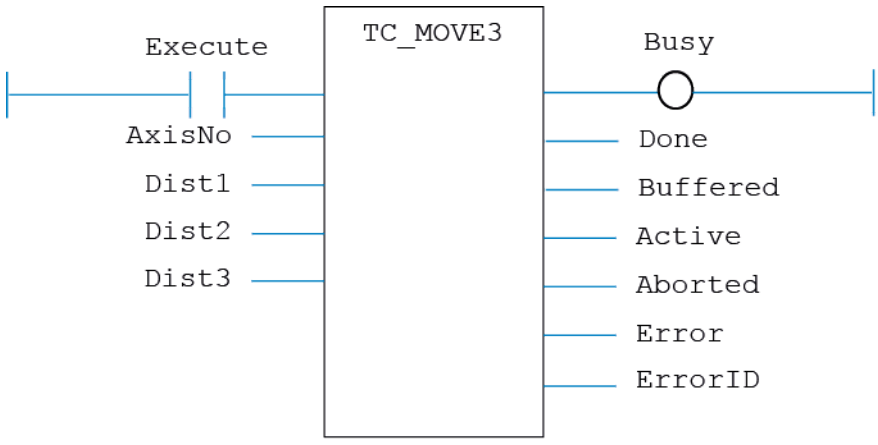
ST LANGUAGE:

```
TC _ MOVE3(Execute, AxisNo, Dist1, Dist2, Dist3, Busy, Done, Buffered, Active,
Aborted, Error, ErrorID);
```

FBD LANGUAGE:



LD LANGUAGE:



IL LANGUAGE:
Not available.

TC_MOVEABS

TYPE:
Motion Function.

FUNCTION:
Issues a new **MOVEABS** motion request for the axis specified by 'AxisNo'.

INPUTS:

Execute : **BOOL**;	Rising edge requests execution
AxisNo : **USINT**;	Axis number of base axis
Count : **USINT**;	Number of axes to be interpolated together
Positions[] : **LREAL**;	Array containing the positions to be moved to, one per axis

OUTPUTS:

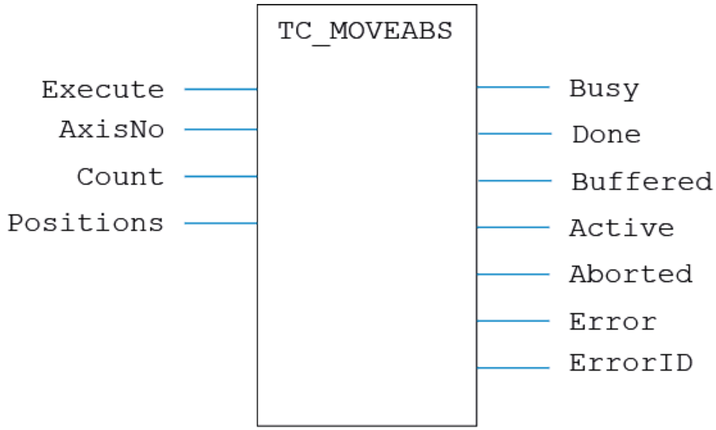
Busy : **BOOL**;	**TRUE** if function is running
Done : **BOOL**;	**TRUE** when function has completed normally
Buffered : **BOOL**;	**TRUE** when motion command is in **NTYPE** buffer
Active : **BOOL**;	**TRUE** when motion command is in **MTYPE** buffer
Aborted : **BOOL**;	**TRUE** if function terminates due to **CANCEL**
Error : **BOOL**;	**TRUE** if a program error is detected
ErrorID : **UINT**;	Returned error number

DESCRIPTION:
When the execute input changes from **FALSE** to **TRUE** (rising edge), the function block attempts to load the motion command into the required axis buffer. If the buffer is unavailable, the function re-tries on each PLC scan. Once the motion command has been loaded, the appropriate outputs will indicate the state of the motion; in **NTYPE**, **MTYPE**, aborted (Cancelled) or done.

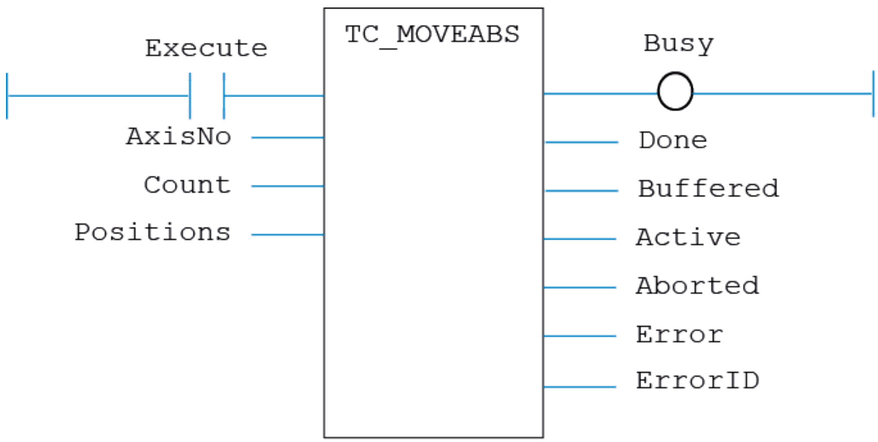
A programming error, such as parameter out of range, will set the Error output and return an error ID number. For the Error ID reference, see the Trio Programming error list.

ST LANGUAGE:
```
TC_MOVEABS(Execute, AxisNo, Count, Positions, Busy, Done, Buffered, Active,
Aborted, Error, ErrorID);
```

FBD LANGUAGE:



LD LANGUAGE:



IL LANGUAGE:

Not available.

TC_MOVEABS1

TYPE:
Motion Function.

FUNCTION:
Issues a new **MOVEABS** (Pos) motion request for the axis specified by 'AxisNo'.

INPUTS:

Execute : **BOOL**;	Rising edge requests execution
AxisNo : **USINT**;	Axis number of base axis
Pos : **LREAL**;	The absolute position to be moved to

OUTPUTS:

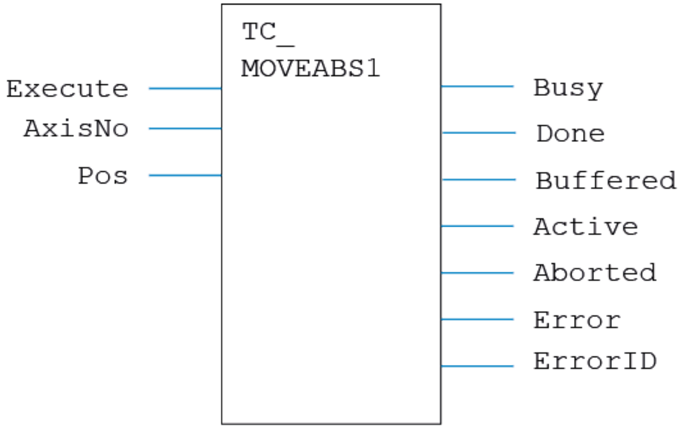
Busy : **BOOL**;	**TRUE** if function is running
Done : **BOOL**;	**TRUE** when function has completed normally
Buffered : **BOOL**;	**TRUE** when motion command is in **NTYPE** buffer
Active : **BOOL**;	**TRUE** when motion command is in **MTYPE** buffer
Aborted : **BOOL**;	**TRUE** if function terminates due to **CANCEL**
Error : **BOOL**;	**TRUE** if a program error is detected
ErrorID : **UINT**;	Returned error number

DESCRIPTION:
When the execute input changes from **FALSE** to **TRUE** (rising edge), the function block attempts to load the motion command into the required axis buffer. If the buffer is unavailable, the function re-tries on each PLC scan. Once the motion command has been loaded, the appropriate outputs will indicate the state of the motion; in **NTYPE**, **MTYPE**, aborted (Cancelled) or done.

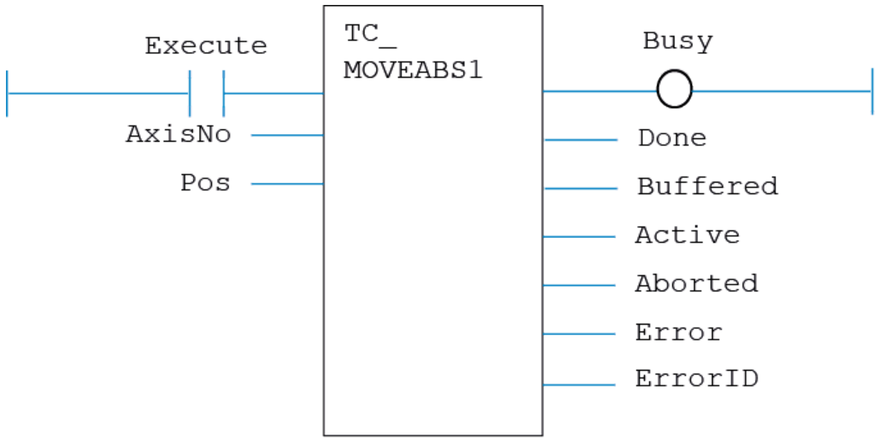
A programming error, such as parameter out of range, will set the Error output and return an error ID number. For the Error ID reference, see the Trio Programming error list.

ST LANGUAGE:
```
TC _ MOVEABS1(Execute, AxisNo, Pos, Busy, Done, Buffered, Active, Aborted, Error,
ErrorID);
```

FBD LANGUAGE:



LD LANGUAGE:



IL LANGUAGE:

Not available.

TC_MOVEABS2

TYPE:
Motion Function.

FUNCTION:
Issues a new **MOVEABS**(Pos1, Pos2) motion request for the pair of axes specified by 'AxisNo'.

INPUTS:

Execute : **BOOL**;	Rising edge requests execution
AxisNo : **USINT**;	Axis number of base axis
Pos1 : **LREAL**;	Position to be moved to on the first axis
Pos2 : **LREAL**;	Position to be moved to on the second axis

OUTPUTS:

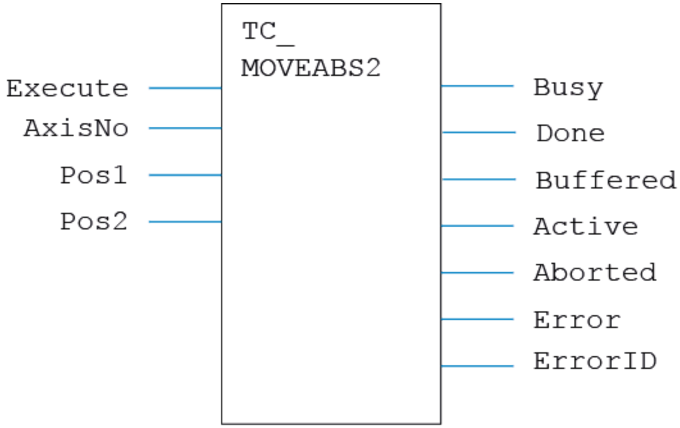
Busy : **BOOL**;	**TRUE** if function is running
Done : **BOOL**;	**TRUE** when function has completed normally
Buffered : **BOOL**;	**TRUE** when motion command is in **NTYPE** buffer
Active : **BOOL**;	**TRUE** when motion command is in **MTYPE** buffer
Aborted : **BOOL**;	**TRUE** if function terminates due to **CANCEL**
Error : **BOOL**;	**TRUE** if a program error is detected
ErrorID : **UINT**;	Returned error number

DESCRIPTION:
When the execute input changes from **FALSE** to **TRUE** (rising edge), the function block attempts to load the motion command into the required axis buffer. If the buffer is unavailable, the function re-tries on each PLC scan. Once the motion command has been loaded, the appropriate outputs will indicate the state of the motion; in **NTYPE**, **MTYPE**, aborted (Cancelled) or done.

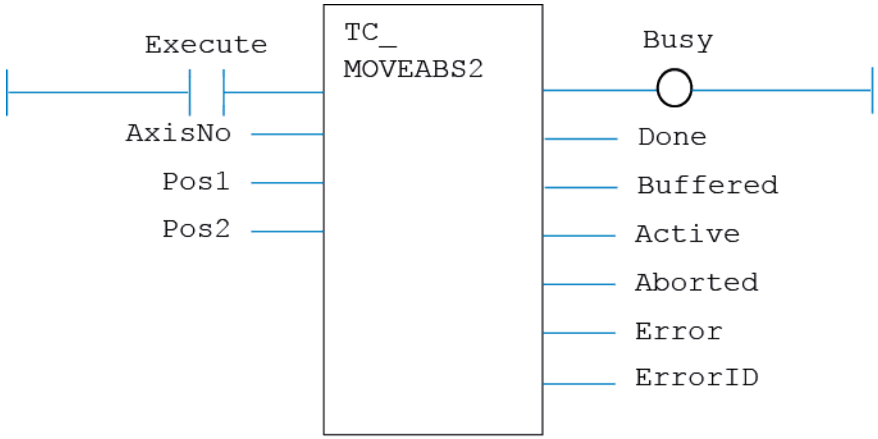
A programming error, such as parameter out of range, will set the Error output and return an error ID number. For the Error ID reference, see the Trio Programming error list.

ST LANGUAGE:
```
TC _ MOVEABS2(Execute, AxisNo, Pos1, Pos2, Busy, Done, Buffered, Active, Aborted,
Error, ErrorID);
```

FBD LANGUAGE:



LD LANGUAGE:



IL LANGUAGE:

Not available.

TC_MOVEABS3

TYPE:
Motion Function.

FUNCTION:
Issues a new **MOVEABS**(Pos1, Pos2, Pos3) motion request for the 3 axes specified by 'AxisNo'.

INPUTS:

Execute : **BOOL**;	Rising edge requests execution
AxisNo : **USINT**;	Axis number of base axis
Pos1 : **LREAL**;	Position to be moved to on the first axis
Pos2 : **LREAL**;	Position to be moved to on the second axis
Pos3 : **LREAL**;	Position to be moved to on the third axis

OUTPUTS:

Busy : **BOOL**;	**TRUE** if function is running
Done : **BOOL**;	**TRUE** when function has completed normally
Buffered : **BOOL**;	**TRUE** when motion command is in **NTYPE** buffer
Active : **BOOL**;	**TRUE** when motion command is in **MTYPE** buffer
Aborted : **BOOL**;	**TRUE** if function terminates due to **CANCEL**
Error : **BOOL**;	**TRUE** if a program error is detected
ErrorID : **UINT**;	Returned error number

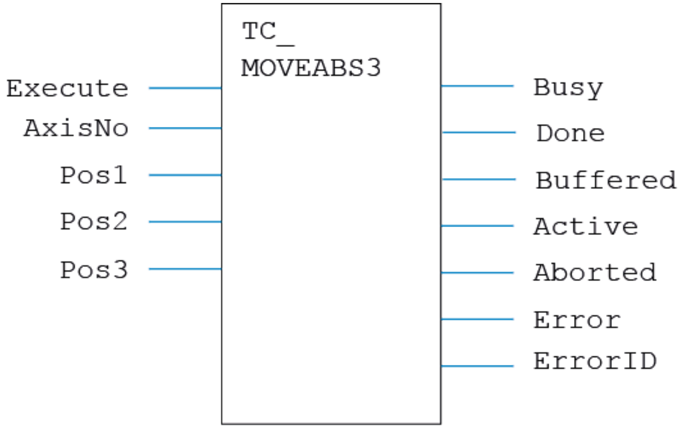
DESCRIPTION:
When the execute input changes from **FALSE** to **TRUE** (rising edge), the function block attempts to load the motion command into the required axis buffer. If the buffer is unavailable, the function re-tries on each PLC scan. Once the motion command has been loaded, the appropriate outputs will indicate the state of the motion; in **NTYPE**, **MTYPE**, aborted (Cancelled) or done.

A programming error, such as parameter out of range, will set the Error output and return an error ID number. For the Error ID reference, see the Trio Programming error list.

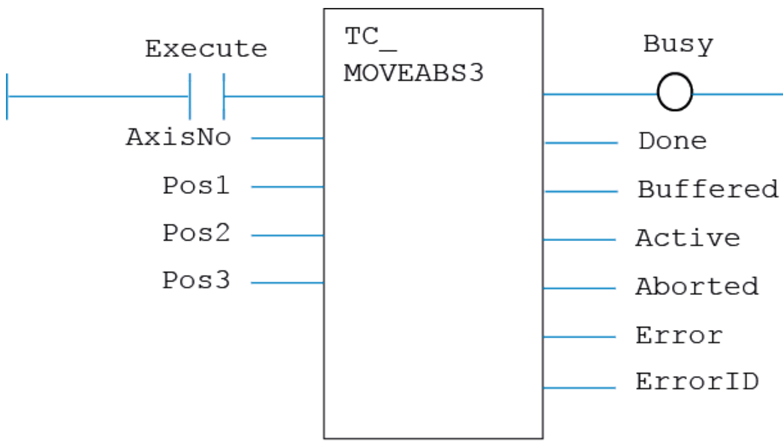
ST LANGUAGE:

```
TC _ MOVEABS3(Execute, AxisNo, Pos1, Pos2, Pos3, Busy, Done, Buffered, Active,
Aborted, Error, ErrorID);
```

FBD LANGUAGE:



LD LANGUAGE:



IL LANGUAGE:

Not available.

TC_MOVEABSSP1

TYPE:
Motion Function.

FUNCTION:
Issues a new **MOVEABSSP**(Pos) motion request for the axes specified by 'AxisNo'.

INPUTS:

Execute **: BOOL**;	Rising edge requests execution
AxisNo **: USINT**;	Axis number
Pos : **LREAL**;	Position to be moved to
ForceSpeed : **REAL**;	**FORCE_SPEED** value
EndmoveSpeed : **REAL**;	**ENDMOVE_SPEED** value

OUTPUTS:

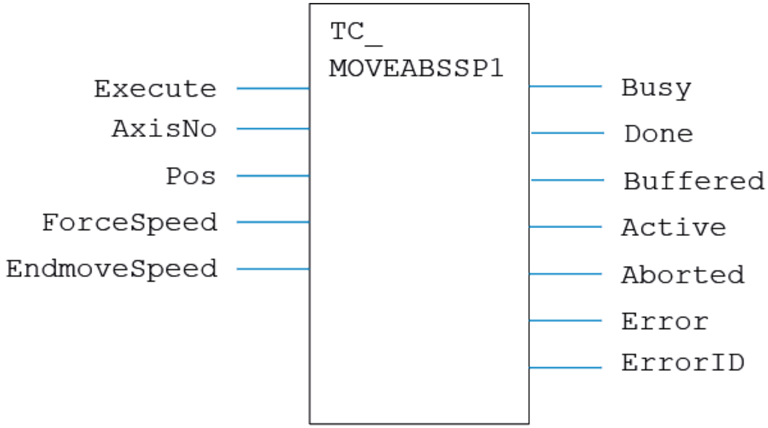
Busy : **BOOL**;	**TRUE** if function is running
Done : **BOOL**;	**TRUE** when function has completed normally
Buffered : **BOOL**;	**TRUE** when motion command is in **NTYPE** buffer
Active : **BOOL**;	**TRUE** when motion command is in **MTYPE** buffer
Aborted : **BOOL**;	**TRUE** if function terminates due to **CANCEL**
Error : **BOOL**;	**TRUE** if a program error is detected
ErrorID : **UINT**;	Returned error number

DESCRIPTION:
When the execute input changes from **FALSE** to **TRUE** (rising edge), the function block attempts to load the motion command into the required axis buffer. If the buffer is unavailable, the function re-tries on each PLC scan. Once the motion command has been loaded, the appropriate outputs will indicate the state of the motion; in **NTYPE**, **MTYPE**, aborted (Cancelled) or done.

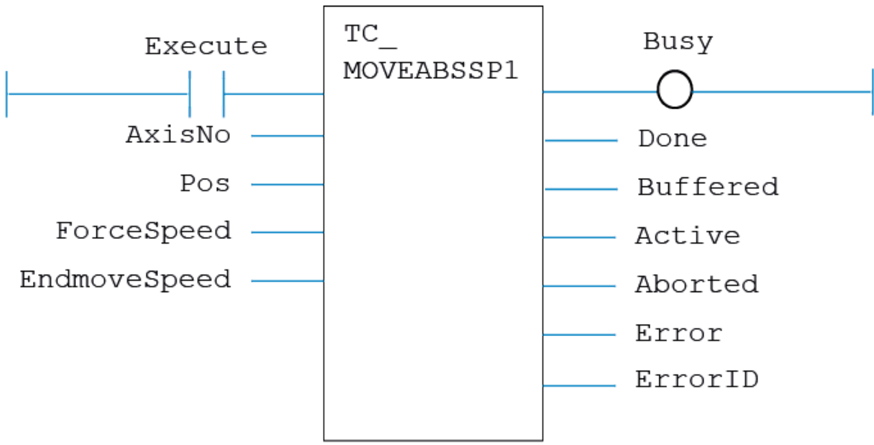
A programming error, such as parameter out of range, will set the Error output and return an error ID number. For the Error ID reference, see the Trio Programming error list.

ST LANGUAGE:
```
TC _ MOVEABSSP2(Execute, AxisNo, Pos, ForceSpeed,  EndmoveSpeed,  Busy, Done,
Buffered, Active, Aborted, Error, ErrorID);
```

FBD LANGUAGE:



LD LANGUAGE:



IL LANGUAGE:

Not available.

TC_MOVEABSSP2

TYPE:
Motion Function.

FUNCTION:
Issues a new **MOVEABSSP**(Pos1, Pos2) motion request for the axes specified by 'AxisNo'.

INPUTS:

Execute : **BOOL**;	Rising edge requests execution
AxisNo : **USINT**;	Axis number
Pos1 : **LREAL**;	Position to be moved to on the first axis
Pos2 : **LREAL**;	Position to be moved to on the second axis
ForceSpeed : **REAL**;	**FORCE_SPEED** value
EndmoveSpeed : **REAL**;	**ENDMOVE_SPEED** value

OUTPUTS:

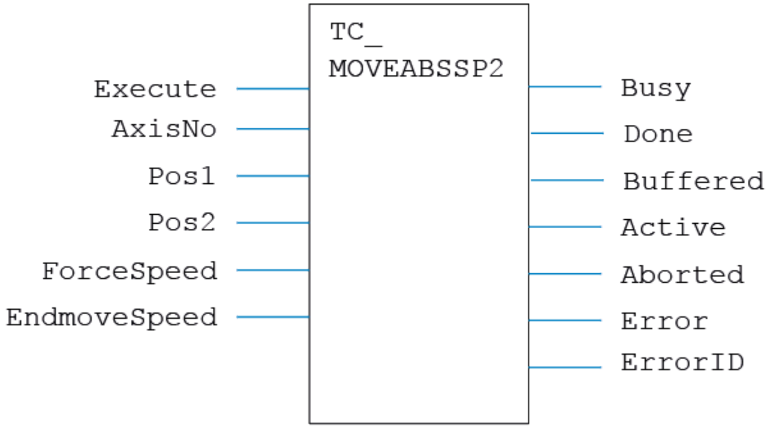
Busy : **BOOL**;	**TRUE** if function is running
Done : **BOOL**;	**TRUE** when function has completed normally
Buffered : **BOOL**;	**TRUE** when motion command is in **NTYPE** buffer
Active : **BOOL**;	**TRUE** when motion command is in **MTYPE** buffer
Aborted : **BOOL**;	**TRUE** if function terminates due to **CANCEL**
Error : **BOOL**;	**TRUE** if a program error is detected
ErrorID : **UINT**;	Returned error number

DESCRIPTION:
When the execute input changes from **FALSE** to **TRUE** (rising edge), the function block attempts to load the motion command into the required axis buffer. If the buffer is unavailable, the function re-tries on each PLC scan. Once the motion command has been loaded, the appropriate outputs will indicate the state of the motion; in **NTYPE**, **MTYPE**, aborted (Cancelled) or done.

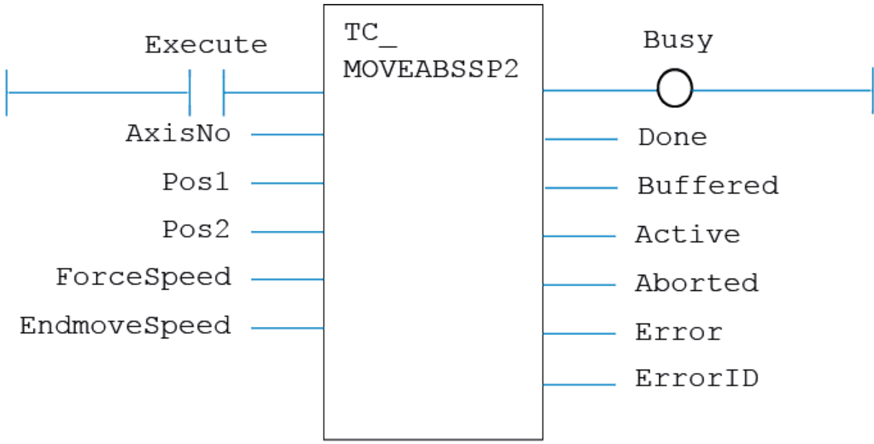
A programming error, such as parameter out of range, will set the Error output and return an error ID number. For the Error ID reference, see the Trio Programming error list.

ST LANGUAGE:
```
TC _ MOVEABSSP2(Execute, AxisNo, Pos1, Pos2, ForceSpeed,  EndmoveSpeed,  Busy,
Done, Buffered, Active, Aborted, Error, ErrorID);
```

FBD LANGUAGE:



LD LANGUAGE:



IL LANGUAGE:

Not available.

TC_MOVEABSSP3

TYPE:
Motion Function.

FUNCTION:
Issues a new **MOVEABSSP**(Pos1, Pos2, Pos3) motion request for the axes specified by 'AxisNo'.

INPUTS:

Execute : **BOOL**;	Rising edge requests execution
AxisNo : **USINT**;	Axis number
Pos1 : **LREAL**;	Position to be moved to on the first axis
Pos2 : **LREAL**;	Position to be moved to on the second axis
Pos3 : **LREAL**;	Position to be moved to on the third axis
ForceSpeed : **REAL**;	**FORCE_SPEED** value
EndmoveSpeed : **REAL**;	**ENDMOVE_SPEED** value

OUTPUTS:

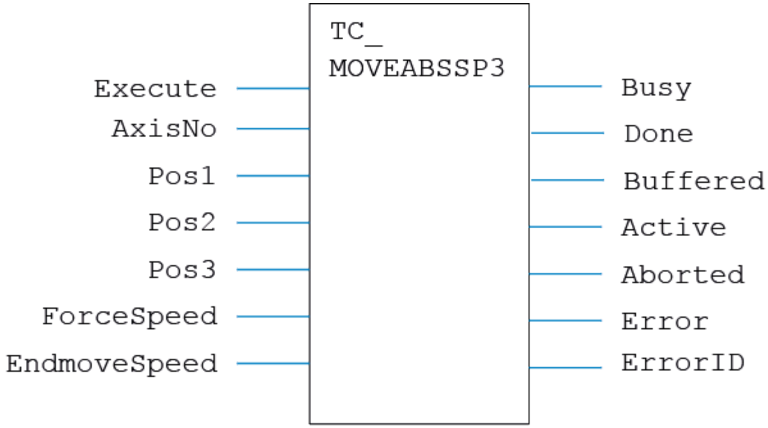
Busy : **BOOL**;	**TRUE** if function is running
Done : **BOOL**;	**TRUE** when function has completed normally
Buffered : **BOOL**;	**TRUE** when motion command is in **NTYPE** buffer
Active : **BOOL**;	**TRUE** when motion command is in **MTYPE** buffer
Aborted : **BOOL**;	**TRUE** if function terminates due to **CANCEL**
Error : **BOOL**;	**TRUE** if a program error is detected
ErrorID : **UINT**;	Returned error number

DESCRIPTION:
When the execute input changes from **FALSE** to **TRUE** (rising edge), the function block attempts to load the motion command into the required axis buffer. If the buffer is unavailable, the function re-tries on each PLC scan. Once the motion command has been loaded, the appropriate outputs will indicate the state of the motion; in **NTYPE**, **MTYPE**, aborted (Cancelled) or done.

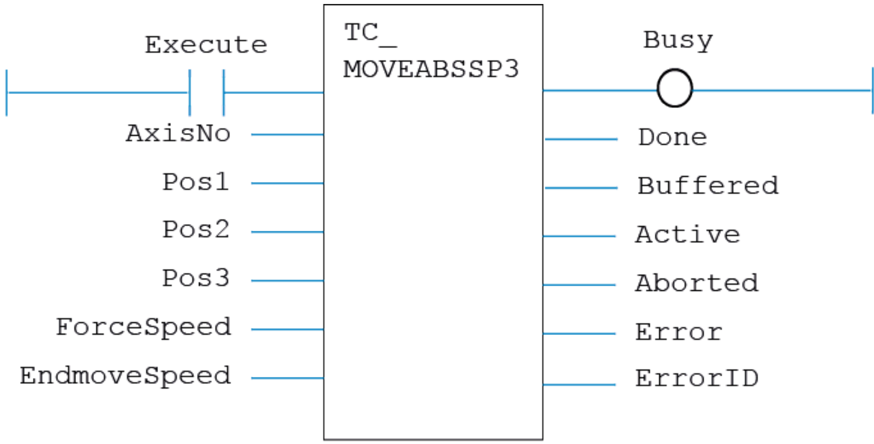
A programming error, such as parameter out of range, will set the Error output and return an error ID number. For the Error ID reference, see the Trio Programming error list.

ST LANGUAGE:
```
TC _ MOVEABSSP3(Execute, AxisNo, Pos1, Pos2, Pos3, ForceSpeed,  EndmoveSpeed,
Busy, Done, Buffered, Active, Aborted, Error, ErrorID);
```

FBD LANGUAGE:



LD LANGUAGE:



IL LANGUAGE:

Not available.

TC_MOVECIRC

TYPE:
Motion Function.

FUNCTION:
Issues a new **MOVECIRC** motion request for the axes specified by 'AxisNo'.

INPUTS:

Execute : **BOOL**;	Rising edge requests execution
AxisNo : **USINT**;	Axis number
End1 : **LREAL**;	Relative end point X
End2 : **LREAL**;	Relative end point Y
Centre1 : **LREAL**;	Relative centre point X
Centre2 : **LREAL**;	Relative centre point Y
Direction : **BOOL**;	Direction of rotation

OUTPUTS:

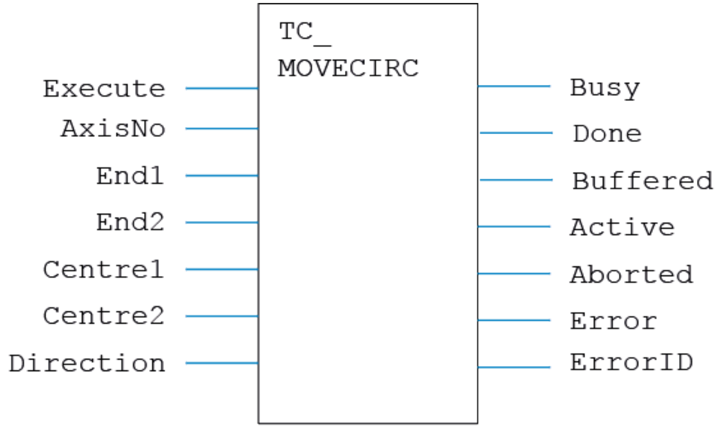
Busy : **BOOL**;	**TRUE** if function is running
Done : **BOOL**;	**TRUE** when function has completed normally
Buffered : **BOOL**;	**TRUE** when motion command is in **NTYPE** buffer
Active : **BOOL**;	**TRUE** when motion command is in **MTYPE** buffer
Aborted : **BOOL**;	**TRUE** if function terminates due to **CANCEL**
Error : **BOOL**;	**TRUE** if a program error is detected
ErrorID : **UINT**;	Returned error number

DESCRIPTION:
When the execute input changes from **FALSE** to **TRUE** (rising edge), the function block attempts to load the motion command into the required axis buffer. If the buffer is unavailable, the function re-tries on each PLC scan. Once the motion command has been loaded, the appropriate outputs will indicate the state of the motion; in **NTYPE**, **MTYPE**, aborted (Cancelled) or done.

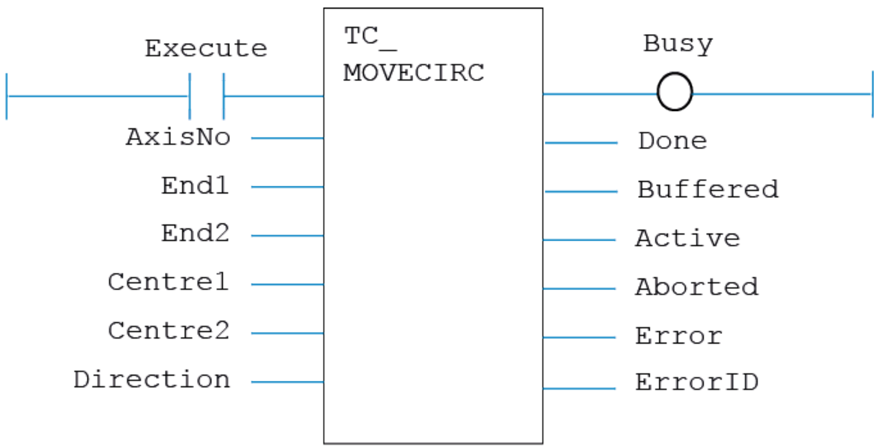
A programming error, such as parameter out of range, will set the Error output and return an error ID number. For the Error ID reference, see the Trio Programming error list.

ST LANGUAGE:
```
TC_MOVECIRC(Execute, AxisNo, End1, End2, Centre1, Centre2, Direction, Busy, Done,
Buffered, Active, Aborted, Error, ErrorID);
```

FBD LANGUAGE:



LD LANGUAGE:



IL LANGUAGE:

Not available.

TC_MOVECIRCSP

TYPE:
Motion Function.

FUNCTION:
Issues a new **MOVECIRCSP** motion request for the axes specified by 'AxisNo'.

INPUTS:

Execute : **BOOL**;	Rising edge requests execution
AxisNo : **USINT**;	Axis number
End1 : **LREAL**;	Relative end point X
End2 : **LREAL**;	Relative end point Y
Centre1 : **LREAL**;	Relative centre point X
Centre2 : **LREAL**;	Relative centre point Y
Direction : **BOOL**;	Direction of rotation
ForceSpeed : **REAL**;	**FORCE_SPEED** value
EndmoveSpeed : **REAL**;	**ENDMOVE_SPEED** value

OUTPUTS:

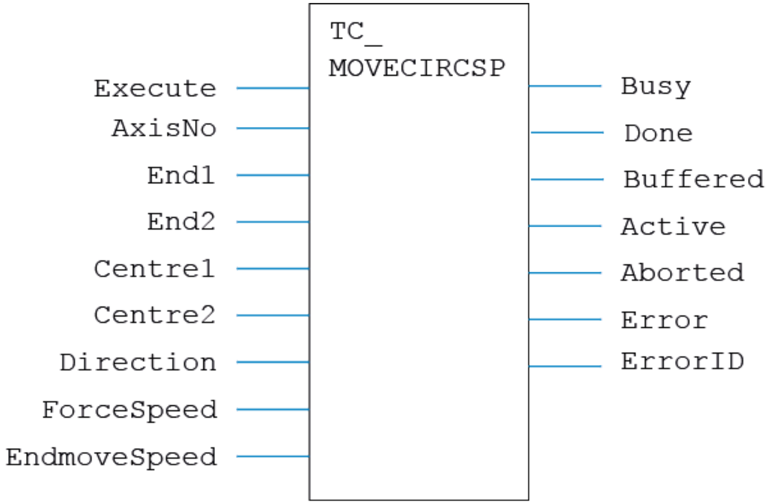
Busy : **BOOL**;	**TRUE** if function is running
Done : **BOOL**;	**TRUE** when function has completed normally
Buffered : **BOOL**;	**TRUE** when motion command is in **NTYPE** buffer
Active : **BOOL**;	**TRUE** when motion command is in **MTYPE** buffer
Aborted : **BOOL**;	**TRUE** if function terminates due to **CANCEL**
Error : **BOOL**;	**TRUE** if a program error is detected
ErrorID : **UINT**;	Returned error number

DESCRIPTION:
When the execute input changes from **FALSE** to **TRUE** (rising edge), the function block attempts to load the motion command into the required axis buffer. If the buffer is unavailable, the function re-tries on each PLC scan. Once the motion command has been loaded, the appropriate outputs will indicate the state of the motion; in **NTYPE**, **MTYPE**, aborted (Cancelled) or done.

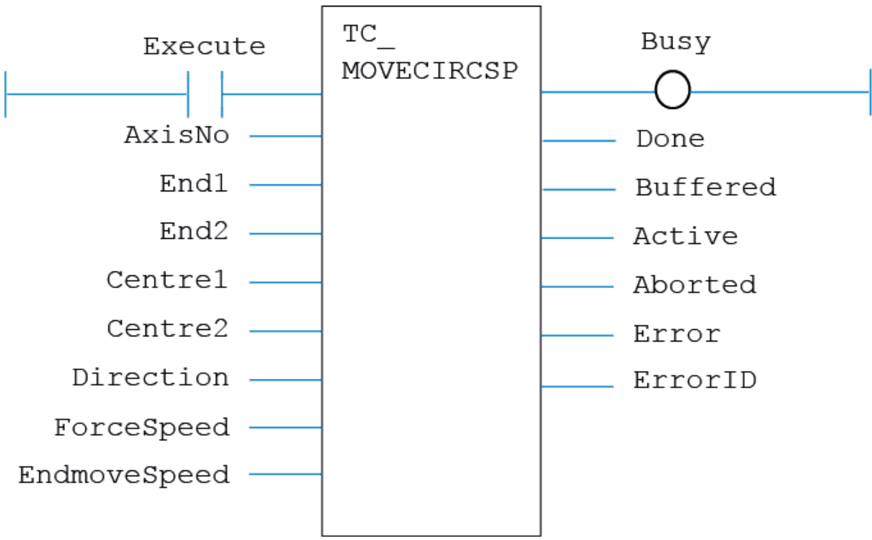
A programming error, such as parameter out of range, will set the Error output and return an error ID number. For the Error ID reference, see the Trio Programming error list.

ST LANGUAGE:
```
TC_MOVECIRCSP(Execute, AxisNo, End1, End2, Centre1, Centre2, Direction, ForceSpeed,
EndmoveSpeed, Busy, Done, Buffered, Active, Aborted, Error, ErrorID);
```

FBD LANGUAGE:



LD LANGUAGE:



IL LANGUAGE:
Not available.

TC_MOVEHELICAL

TYPE:
Motion Function.

FUNCTION:
Issues a new **MHELICAL** motion request for the axes specified by 'AxisNo'.

INPUTS:

Execute : **BOOL**;	Rising edge requests execution
AxisNo : **USINT**;	Axis number
End1 : **LREAL**;	Relative end point X
End2 : **LREAL**;	Relative end point Y
Centre1 : **LREAL**;	Relative centre point X
Centre2 : **LREAL**;	Relative centre point Y
Direction : **BOOL**;	Direction of rotation
Z : **LREAL**;	Linear distance in Z

OUTPUTS:

Busy : **BOOL**;	**TRUE** if function is running
Done : **BOOL**;	**TRUE** when function has completed normally
Buffered : **BOOL**;	**TRUE** when motion command is in **NTYPE** buffer
Active : **BOOL**;	**TRUE** when motion command is in **MTYPE** buffer
Aborted : **BOOL**;	**TRUE** if function terminates due to **CANCEL**
Error : **BOOL**;	**TRUE** if a program error is detected
ErrorID : **UINT**;	Returned error number

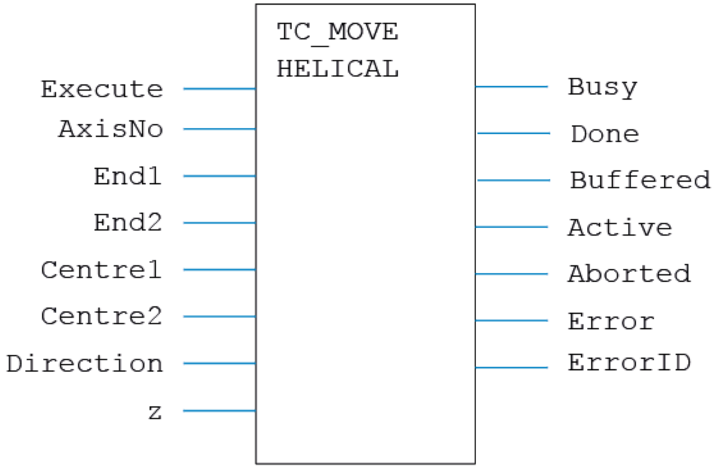
DESCRIPTION:
When the execute input changes from **FALSE** to **TRUE** (rising edge), the function block attempts to load the motion command into the required axis buffer. If the buffer is unavailable, the function re-tries on each PLC scan. Once the motion command has been loaded, the appropriate outputs will indicate the state of the motion; in **NTYPE**, **MTYPE**, aborted (Cancelled) or done.

A programming error, such as parameter out of range, will set the Error output and return an error ID number. For the Error ID reference, see the Trio Programming error list.

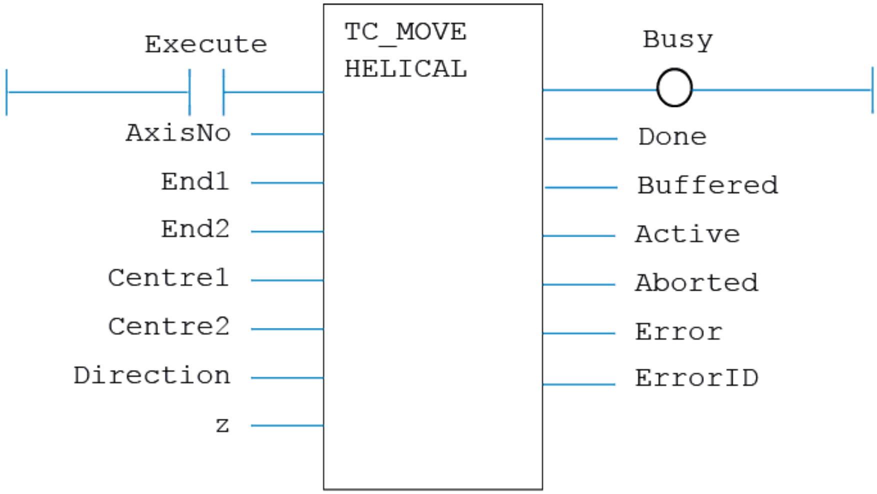
ST LANGUAGE:

```
TC_MOVEHELICAL(Execute, AxisNo, End1, End2, Centre1, Centre2,  Direction, z, Busy,
Done, Buffered, Active, Aborted, Error, ErrorID);
```

FBD LANGUAGE:



LD LANGUAGE:



IL LANGUAGE:
Not available.

TC_MOVEHELICALSP

TYPE:
Motion Function.

FUNCTION:
Issues a new **MHELICALSP** motion request for the axes specified by ‘AxisNo’.

INPUTS:

Execute : **BOOL**;	Rising edge requests execution
AxisNo : **USINT**;	Axis number
End1 : **LREAL**;	Relative end point X
End2 : **LREAL**;	Relative end point Y
Centre1 : **LREAL**;	Relative centre point X
Centre2 : **LREAL**;	Relative centre point Y

Direction : **BOOL**;	Direction of rotation
z : **LREAL**;	Linear distance for Z
ForceSpeed : **REAL**;	**FORCE_SPEED** value
EndmoveSpeed : **REAL**;	**ENDMOVE_SPEED** value

OUTPUTS:

Busy : **BOOL**;	**TRUE** if function is running
Done : **BOOL**;	**TRUE** when function has completed normally
Buffered : **BOOL**;	**TRUE** when motion command is in **NTYPE** buffer
Active : **BOOL**;	**TRUE** when motion command is in **MTYPE** buffer
Aborted : **BOOL**;	**TRUE** if function terminates due to **CANCEL**
Error : **BOOL**;	**TRUE** if a program error is detected
ErrorID : **UINT**;	Returned error number

DESCRIPTION:

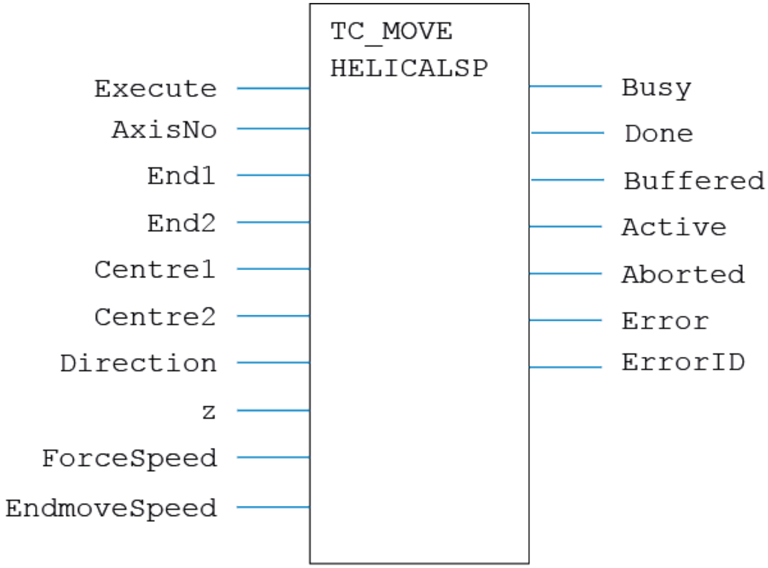
When the execute input changes from **FALSE** to **TRUE** (rising edge), the function block attempts to load the motion command into the required axis buffer. If the buffer is unavailable, the function re-tries on each PLC scan. Once the motion command has been loaded, the appropriate outputs will indicate the state of the motion; in **NTYPE**, **MTYPE**, aborted (Cancelled) or done.

A programming error, such as parameter out of range, will set the Error output and return an error ID number. For the Error ID reference, see the Trio Programming error list.

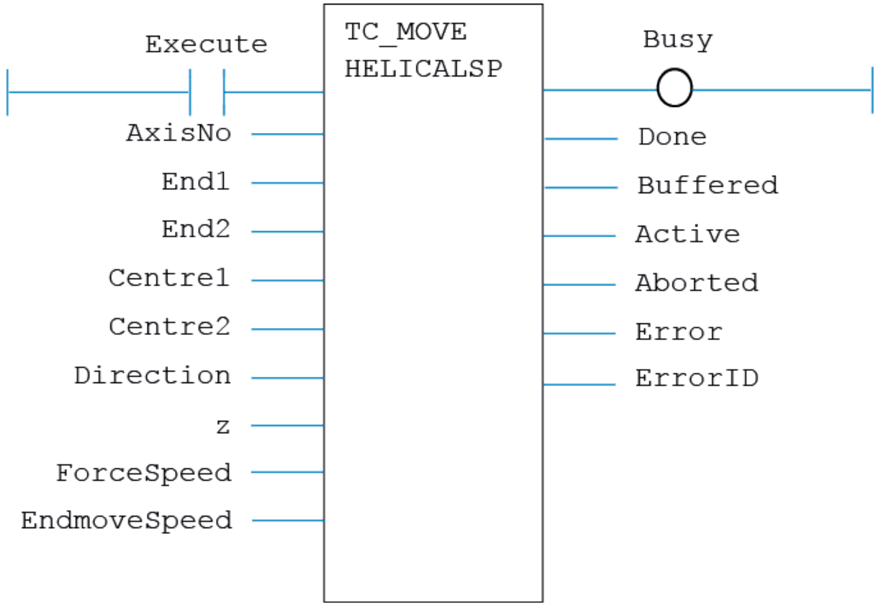
ST LANGUAGE:

```
TC_MOVEHELICALSP(Execute, AxisNo, End1, End2, Centre1, Centre2, Direction, z,
ForceSpeed, EndmoveSpeed, Busy, Done, Buffered, Active, Aborted, Error, ErrorID);
```

FBD LANGUAGE:



LD LANGUAGE:



IL LANGUAGE:
Not available.

TC_MOVELINK

TYPE:
Motion Function.

FUNCTION:
Issues a new **MOVELINK** motion request for the axis specified by ‘AxisNo’.

INPUTS:

Execute : **BOOL**;	Rising edge requests execution
AxisNo : **USINT**;	Axis number

Dist : **LREAL**;	Distance to move
LinkAxis **: USINT**;	Link axis number
LinkDist **: LREAL**;	Total distance on link axis
LinkAccDist **: USINT**;	Distance on link axis for acceleration ramp
LinkDecDist **: LREAL**;	Distance on link axis for deceleration ramp
Options : **DINT**;	Link options, set to 0 for none
LinkPos **: LREAL**;	Link Position, set to 0 if unused

OUTPUTS:

Busy : **BOOL**;	**TRUE** if function is running
Done : **BOOL**;	**TRUE** when function has completed normally
Buffered : **BOOL**;	**TRUE** when motion command is in **NTYPE** buffer
Active : **BOOL**;	**TRUE** when motion command is in **MTYPE** buffer
Aborted : **BOOL**;	**TRUE** if function terminates due to **CANCEL**
Error : **BOOL**;	**TRUE** if a program error is detected
ErrorID : **UINT**;	Returned error number

DESCRIPTION:

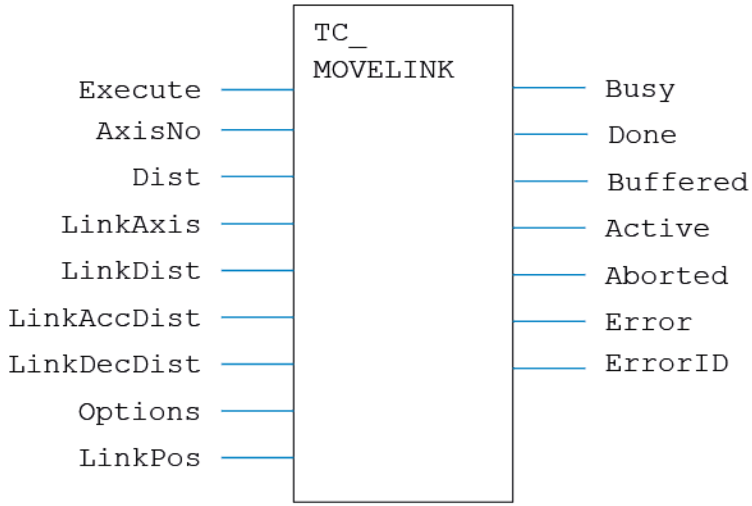
When the execute input changes from **FALSE** to **TRUE** (rising edge), the function block attempts to load the motion command into the required axis buffer. If the buffer is unavailable, the function re-tries on each PLC scan. Once the motion command has been loaded, the appropriate outputs will indicate the state of the motion; in **NTYPE**, **MTYPE**, aborted (Cancelled) or done.

A programming error, such as parameter out of range, will set the Error output and return an error ID number. For the Error ID reference, see the Trio Programming error list.

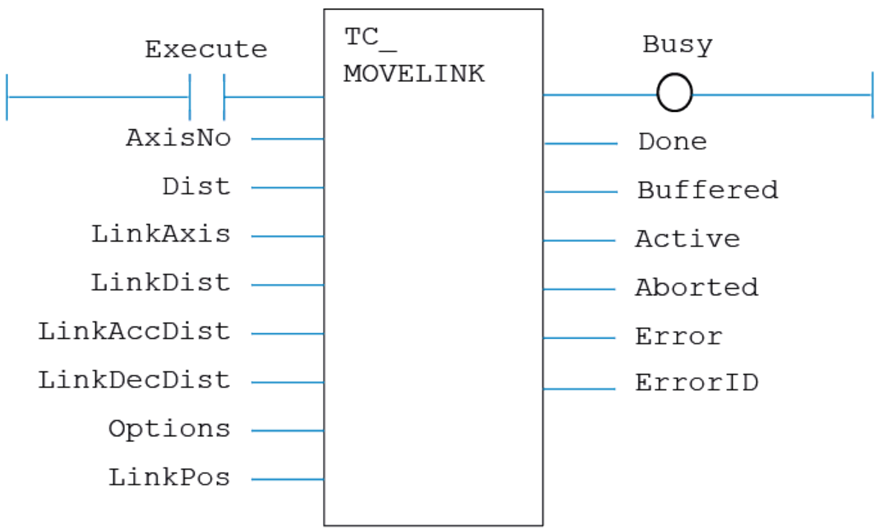
ST LANGUAGE:

```
TC_MOVELINK(Execute, AxisNo, Dist, LinkAxis, LinkDist, LinkAccDist,  LinkDecDist,
Options, LinkPos, Busy, Done, Buffered, Active, Aborted, Error, ErrorID);
```

FBD LANGUAGE:



LD LANGUAGE:



IL LANGUAGE:
Not available.

TC_MOVEMODIFY

TYPE:
Motion Function.

FUNCTION:
Issues a new **MOVEMODIFY** motion request for the axis specified by ‘AxisNo’.

INPUTS:

Execute : **BOOL**;	Rising edge requests execution
AxisNo : **USINT**;	Axis number of base axis
Pos : **LREAL**;	The absolute position to be moved to

OUTPUTS:

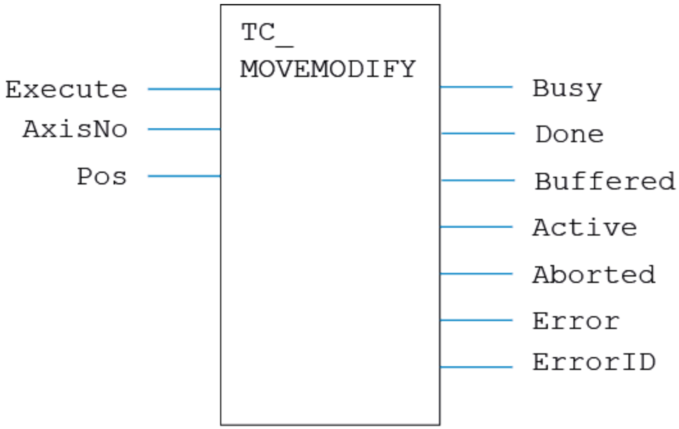
Busy : **BOOL**;	**TRUE** if function is running
Done : **BOOL**;	**TRUE** when function has completed normally
Buffered : **BOOL**;	**TRUE** when motion command is in **NTYPE** buffer
Active : **BOOL**;	**TRUE** when motion command is in **MTYPE** buffer
Aborted : **BOOL**;	**TRUE** if function terminates due to **CANCEL**
Error : **BOOL**;	**TRUE** if a program error is detected
ErrorID : **UINT**;	Returned error number

DESCRIPTION:
When the execute input changes from **FALSE** to **TRUE** (rising edge), the function block attempts to load the motion command into the required axis buffer. If the buffer is unavailable, the function re-tries on each PLC scan. Once the motion command has been loaded, the appropriate outputs will indicate the state of the motion; in **NTYPE**, **MTYPE**, aborted (Cancelled) or done.

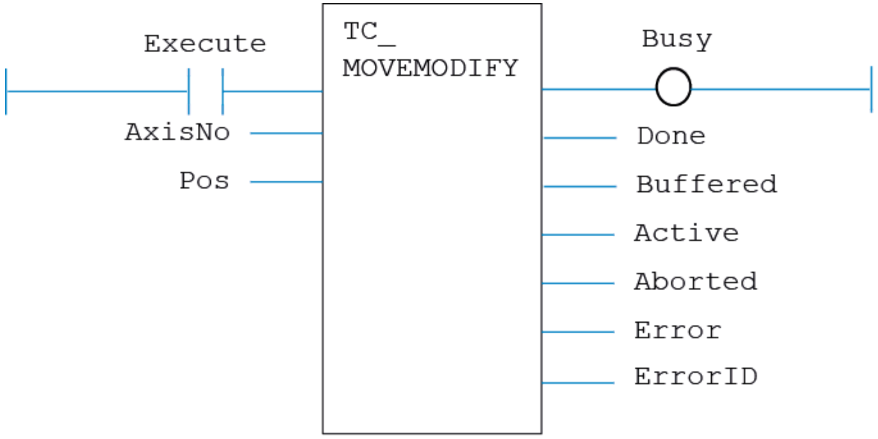
A programming error, such as parameter out of range, will set the Error output and return an error ID number. For the Error ID reference, see the Trio Programming error list.

ST LANGUAGE:
```
TC_MOVEMODIFY(Execute, AxisNo, Pos, Busy, Done, Buffered, Active, Aborted, Error,
ErrorID);
```

FBD LANGUAGE:



LD LANGUAGE:



IL LANGUAGE:

Not available.

TC_MOVESP

TYPE:
Motion Function.

FUNCTION:
Issues a new **MOVESP** motion request for the axis specified by 'AxisNo'.

INPUTS:

Execute : **BOOL**;	Rising edge requests execution
AxisNo : **USINT**;	Axis number of base axis
Count : **USINT**;	Number of axes to be interpolated together
Distances[] : **LREAL**;	Array containing the distances to be moved, one per axis
ForceSpeed : **REAL**;	**FORCE_SPEED** value
EndmoveSpeed : **REAL**;	**ENDMOVE_SPEED** value

OUTPUTS:

Busy : **BOOL**;	**TRUE** if function is running
Done : **BOOL**;	**TRUE** when function has completed normally
Buffered : **BOOL**;	**TRUE** when motion command is in **NTYPE** buffer
Active : **BOOL**;	**TRUE** when motion command is in **MTYPE** buffer
Aborted : **BOOL**;	**TRUE** if function terminates due to **CANCEL**
Error : **BOOL**;	**TRUE** if a program error is detected
ErrorID : **UINT**;	Returned error number

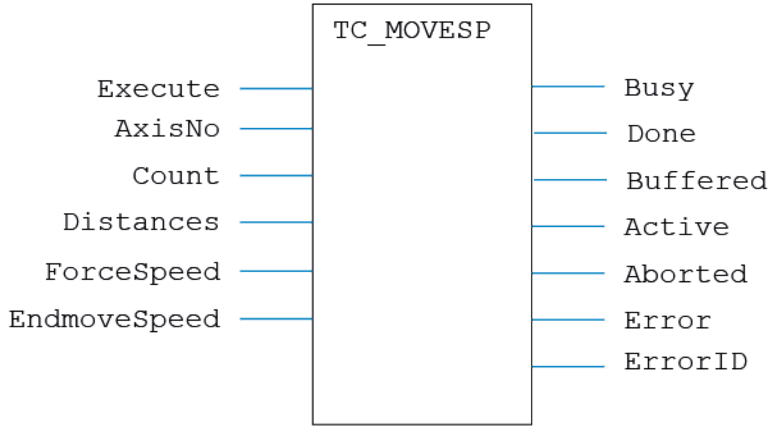
DESCRIPTION:
When the execute input changes from **FALSE** to **TRUE** (rising edge), the function block attempts to load the motion command into the required axis buffer. If the buffer is unavailable, the function re-tries on each PLC scan. Once the motion command has been loaded, the appropriate outputs will indicate the state of the motion; in **NTYPE**, **MTYPE**, aborted (Cancelled) or done.

A programming error, such as parameter out of range, will set the Error output and return an error ID number. For the Error ID reference, see the Trio Programming error list.

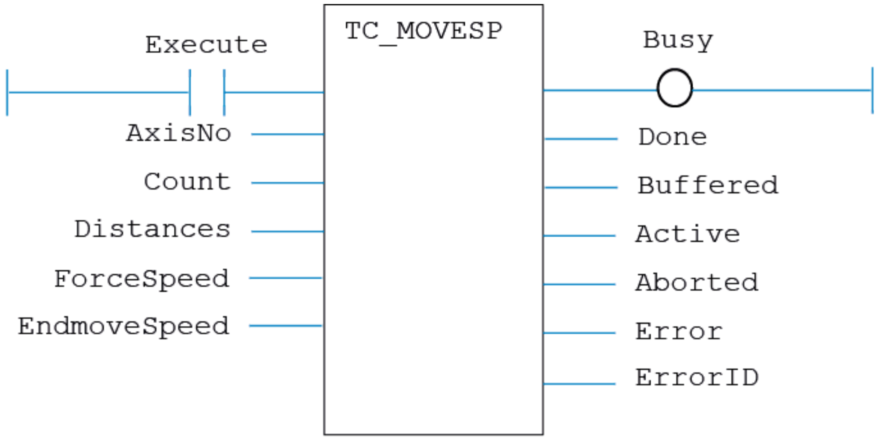
ST LANGUAGE:

```
TC_MOVESP(Execute, AxisNo, Count, Distances, ForceSpeed, EndmoveSpeed, Busy, Done,
Buffered, Active, Aborted, Error, ErrorID);
```

FBD LANGUAGE:



LD LANGUAGE:



IL LANGUAGE:

Not available.

TC_MOVESP1

TYPE:
Motion Function.

FUNCTION:
Issues a new **MOVESP**(Dist) motion request for the axis specified by 'AxisNo'.

INPUTS:

Execute : **BOOL**;	Rising edge requests execution
AxisNo : **USINT**;	Axis number of base axis
Dist : **LREAL**;	The distance to be moved
ForceSpeed : **REAL**;	**FORCE_SPEED** value
EndmoveSpeed : **REAL**;	**ENDMOVE_SPEED** value

OUTPUTS:

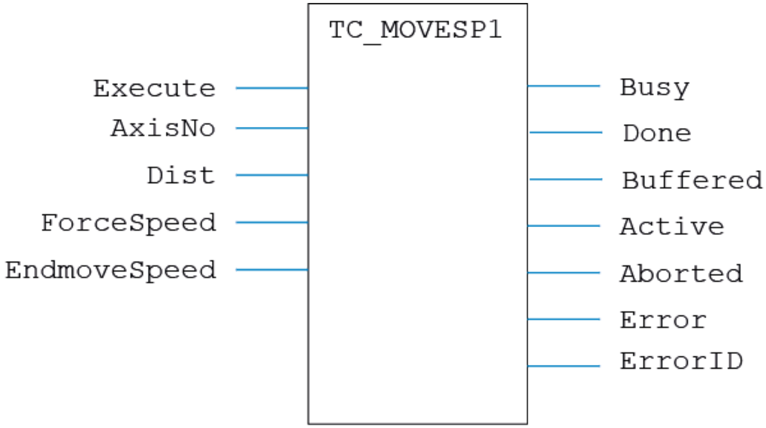
Busy : **BOOL**;	**TRUE** if function is running
Done : **BOOL**;	**TRUE** when function has completed normally
Buffered : **BOOL**;	**TRUE** when motion command is in **NTYPE** buffer
Active : **BOOL**;	**TRUE** when motion command is in **MTYPE** buffer
Aborted : **BOOL**;	**TRUE** if function terminates due to **CANCEL**
Error : **BOOL**;	**TRUE** if a program error is detected
ErrorID : **UINT**;	Returned error number

DESCRIPTION:
When the execute input changes from **FALSE** to **TRUE** (rising edge), the function block attempts to load the motion command into the required axis buffer. If the buffer is unavailable, the function re-tries on each PLC scan. Once the motion command has been loaded, the appropriate outputs will indicate the state of the motion; in **NTYPE**, **MTYPE**, aborted (Cancelled) or done.

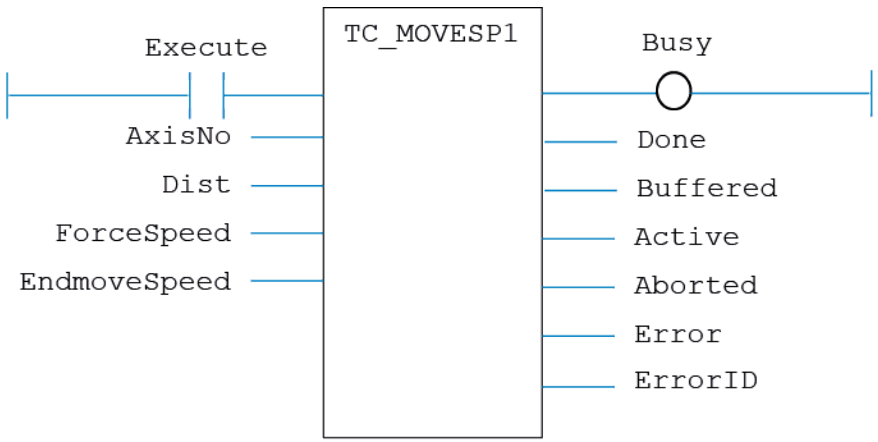
A programming error, such as parameter out of range, will set the Error output and return an error ID number. For the Error ID reference, see the Trio Programming error list.

ST LANGUAGE:
```
TC _ MOVESP1(Execute, AxisNo, Dist, ForceSpeed, EndmoveSpeed, Busy, Done, Buffered,
Active, Aborted, Error, ErrorID);
```

FBD LANGUAGE:



LD LANGUAGE:



IL LANGUAGE:
Not available.

TC_MOVESP2

TYPE:
Motion Function.

FUNCTION:
Issues a new **MOVESP**(Dist1, Dist2) motion request for the pair of axes specified by 'AxisNo'.

INPUTS:

Execute : **BOOL**;	Rising edge requests execution
AxisNo : **USINT**;	Axis number of base axis
Dist1 : **LREAL**;	Distance to be moved on the first axis
Dist2 : **LREAL**;	Distance to be moved on the second axis
ForceSpeed : **REAL**;	**FORCE_SPEED** value
EndmoveSpeed : **REAL**;	**ENDMOVE_SPEED** value

OUTPUTS:

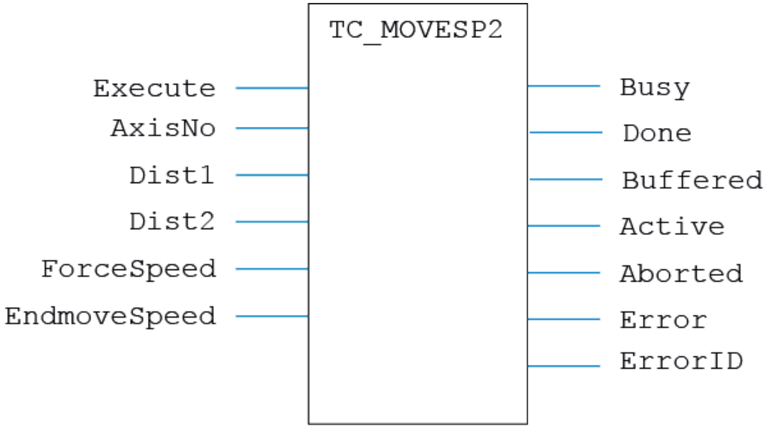
Busy : **BOOL**;	**TRUE** if function is running
Done : **BOOL**;	**TRUE** when function has completed normally
Buffered : **BOOL**;	**TRUE** when motion command is in **NTYPE** buffer
Active : **BOOL**;	**TRUE** when motion command is in **MTYPE** buffer
Aborted : **BOOL**;	**TRUE** if function terminates due to **CANCEL**
Error : **BOOL**;	**TRUE** if a program error is detected
ErrorID : **UINT**;	Returned error number

DESCRIPTION:
When the execute input changes from **FALSE** to **TRUE** (rising edge), the function block attempts to load the motion command into the required axis buffer. If the buffer is unavailable, the function re-tries on each PLC scan. Once the motion command has been loaded, the appropriate outputs will indicate the state of the motion; in **NTYPE**, **MTYPE**, aborted (Cancelled) or done.

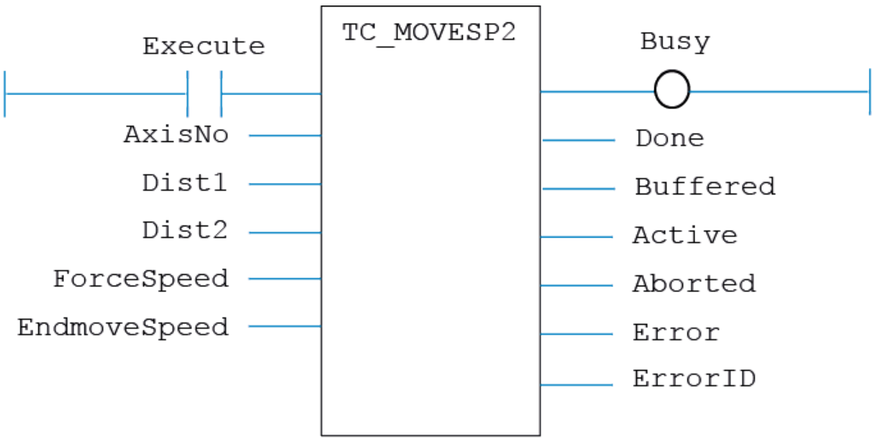
A programming error, such as parameter out of range, will set the Error output and return an error ID number. For the Error ID reference, see the Trio Programming error list.

ST LANGUAGE:
TC _ MOVESP2(Execute, AxisNo, Dist1, Dist2, ForceSpeed, EndmoveSpeed, Busy, Done, Buffered, Active, Aborted, Error, ErrorID);

FBD LANGUAGE:



LD LANGUAGE:



IL LANGUAGE:
Not available.

TC_MOVESP3

TYPE:
Motion Function.

FUNCTION:
Issues a new **MOVESP**(Dist1, Dist2, Dist3) motion request for the 3 axes specified by ‘AxisNo’.

INPUTS:

Execute : **BOOL**;	Rising edge requests execution
AxisNo : **USINT**;	Axis number of base axis
Dist1 : **LREAL**;	Distance to be moved on the first axis
Dist2 : **LREAL**;	Distance to be moved on the second axis
Dist3 : **LREAL**;	Distance to be moved on the third axis
ForceSpeed : **REAL**;	**FORCE_SPEED** value
EndmoveSpeed : **REAL**;	**ENDMOVE_SPEED** value

OUTPUTS:

Busy : **BOOL**;	**TRUE** if function is running
Done : **BOOL**;	**TRUE** when function has completed normally
Buffered : **BOOL**;	**TRUE** when motion command is in **NTYPE** buffer
Active : **BOOL**;	**TRUE** when motion command is in **MTYPE** buffer
Aborted : **BOOL**;	**TRUE** if function terminates due to **CANCEL**
Error : **BOOL**;	**TRUE** if a program error is detected
ErrorID : **UINT**;	Returned error number

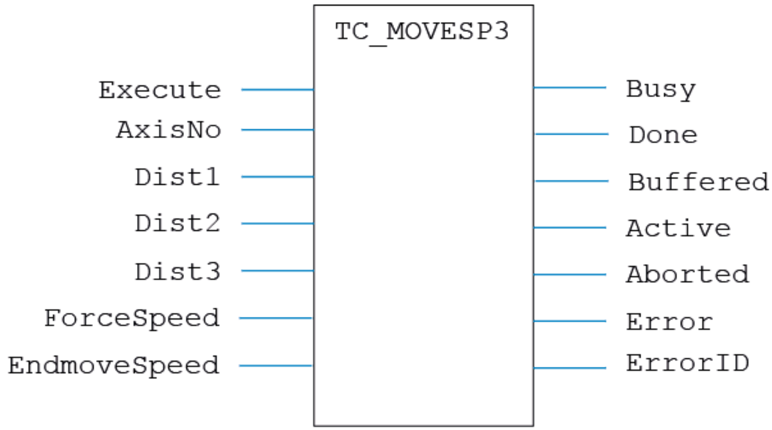
DESCRIPTION:
When the execute input changes from **FALSE** to **TRUE** (rising edge), the function block attempts to load the motion command into the required axis buffer. If the buffer is unavailable, the function re-tries on each PLC scan. Once the motion command has been loaded, the appropriate outputs will indicate the state of the motion; in **NTYPE**, **MTYPE**, aborted (Cancelled) or done.

A programming error, such as parameter out of range, will set the Error output and return an error ID number. For the Error ID reference, see the Trio Programming error list.

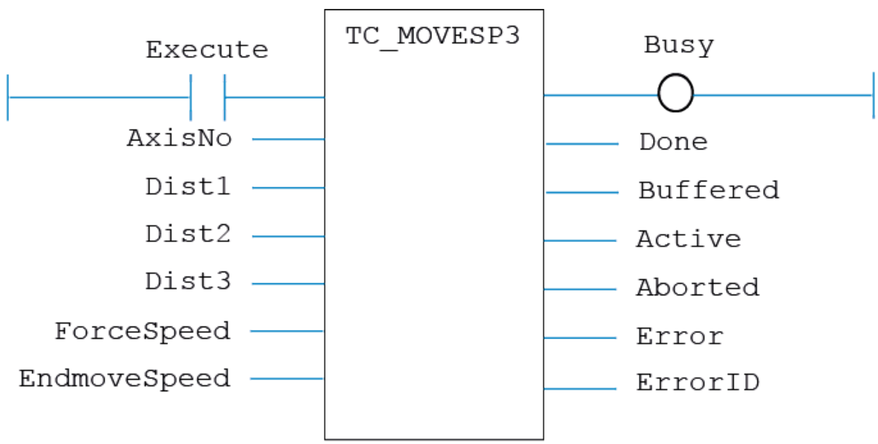
ST LANGUAGE:

```
TC _ MOVESP3(Execute, AxisNo, Dist1, Dist2, Dist3, ForceSpeed,
EndmoveSpeed, Busy, Done, Buffered, Active, Aborted, Error,
ErrorID);
```

FBD LANGUAGE:



LD LANGUAGE:



IL LANGUAGE:

Not available.

TC_MOVETANG

TYPE:
Motion Function.

FUNCTION:
Issues a new **MOVETANG** motion request for the axis specified by 'AxisNo'.

INPUTS:

Execute : **BOOL**;	Rising edge requests execution
AxisNo : **USINT**;	Axis number of base axis
EndPos : **LREAL**;	Position
LinkAxis : **USINT**;	Base axis number of the axis pair to follow
DisableLinkAxis : **BOOL**;	Operates the disable link axis function

OUTPUTS:

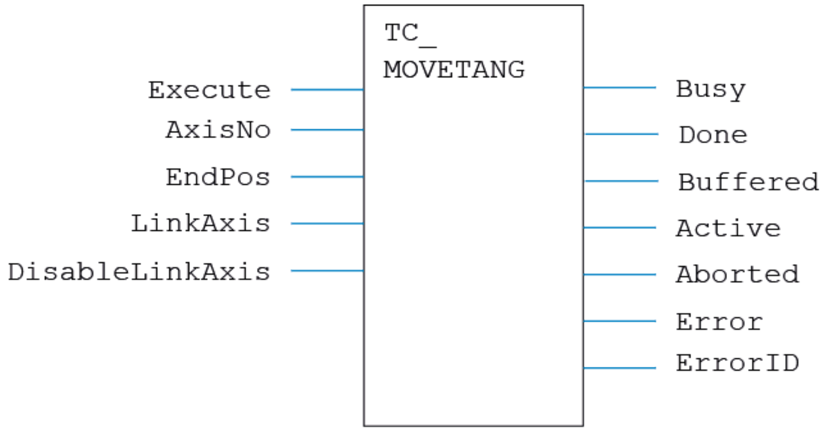
Busy : **BOOL**;	**TRUE** if function is running
Done : **BOOL**;	**TRUE** when function has completed normally
Buffered : **BOOL**;	**TRUE** when motion command is in **NTYPE** buffer
Active : **BOOL**;	**TRUE** when motion command is in **MTYPE** buffer
Aborted : **BOOL**;	**TRUE** if function terminates due to **CANCEL**
Error : **BOOL**;	**TRUE** if a program error is detected
ErrorID : **UINT**;	Returned error number

DESCRIPTION:
When the execute input changes from **FALSE** to **TRUE** (rising edge), the function block attempts to load the motion command into the required axis buffer. If the buffer is unavailable, the function re-tries on each PLC scan. Once the motion command has been loaded, the appropriate outputs will indicate the state of the motion; in **NTYPE**, **MTYPE**, aborted (Cancelled) or done.

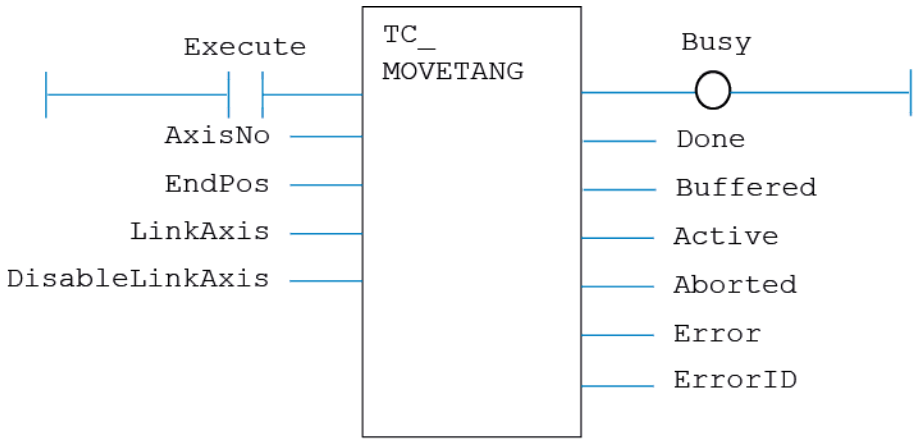
A programming error, such as parameter out of range, will set the Error output and return an error ID number. For the Error ID reference, see the Trio Programming error list.

ST LANGUAGE:
```
TC_MOVETANG(Execute, AxisNo, EndPos, LinkAxis, DisableLinkAxis, Busy, Done,
Buffered, Active, Aborted, Error, ErrorID);
```

FBD LANGUAGE:



LD LANGUAGE:



IL LANGUAGE:
Not available.

TC_MSPHERICAL

TYPE:
Motion Function.

FUNCTION:
Issues a new **MSPHERICAL** motion request for the axes specified by ‘AxisNo’.

INPUTS:

Execute : **BOOL**;	Rising edge requests execution
AxisNo : **USINT**;	Axis number
EndX : **LREAL**;	Relative end point X
EndY : **LREAL**;	Relative end point Y
EndZ : **LREAL**;	Relative end point Z
MidX : **LREAL**;	Relative mid-point X
MidY : **LREAL**;	Relative mid- point Y
MidZ : **LREAL**;	Relative mid- point Z
Mode : INT;	Mode
GtPI : INT;	Direction control

OUTPUTS:

Busy : **BOOL**;	**TRUE** if function is running
Done : **BOOL**;	**TRUE** when function has completed normally
Buffered : **BOOL**;	**TRUE** when motion command is in **NTYPE** buffer
Active : **BOOL**;	**TRUE** when motion command is in **MTYPE** buffer
Aborted : **BOOL**;	**TRUE** if function terminates due to **CANCEL**
Error : **BOOL**;	**TRUE** if a program error is detected
ErrorID : **UINT**;	Returned error number

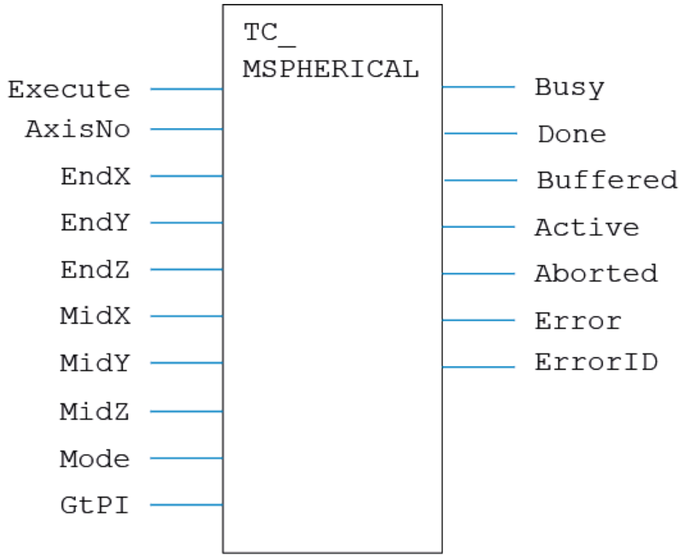
DESCRIPTION:
When the execute input changes from **FALSE** to **TRUE** (rising edge), the function block attempts to load the motion command into the required axis buffer. If the buffer is unavailable, the function re-tries on each PLC scan. Once the motion command has been loaded, the appropriate outputs will indicate the state of the motion; in **NTYPE**, **MTYPE**, aborted (Cancelled) or done.

A programming error, such as parameter out of range, will set the Error output and return an error ID number. For the Error ID reference, see the Trio Programming error list.

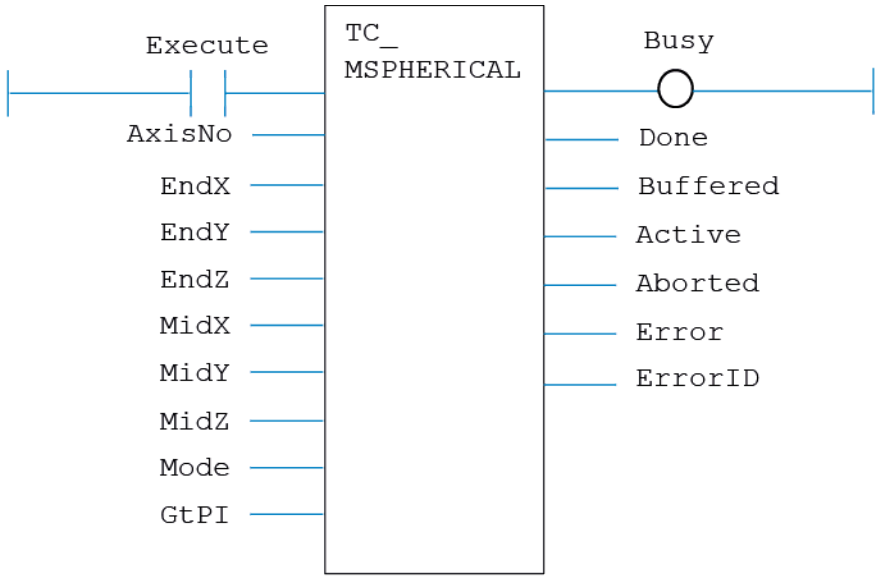
ST LANGUAGE:

```
TC_MSPHERICAL(Execute, AxisNo, EndX, EndY, EndZ, MidX, MidY, MidZ, Mode, GtPI,
Busy, Done, Buffered, Active, Aborted, Error, ErrorID);
```

FBD LANGUAGE:



LD LANGUAGE:



IL LANGUAGE:
Not available.

TC_MSPHERICALSP

TYPE:
Motion Function.

FUNCTION:
Issues a new **MSPHERICALSP** motion request for the axes specified by ‘AxisNo’.

INPUTS:

Execute : **BOOL**;	Rising edge requests execution
AxisNo : **USINT**;	Axis number
EndX : **LREAL**;	Relative end point X
EndY : **LREAL**;	Relative end point Y

EndZ : **LREAL**;	Relative end point Z
MidX : **LREAL**;	Relative mid-point X
MidY : **LREAL**;	Relative mid- point Y
MidZ : **LREAL**;	Relative mid- point Z
Mode : INT;	Mode
GtPI : INT;	Direction control
ForceSpeed : **REAL**;	**FORCE_SPEED** value
EndmoveSpeed : **REAL**;	**ENDMOVE_SPEED** value

OUTPUTS:

Busy : **BOOL**;	**TRUE** if function is running
Done : **BOOL**;	**TRUE** when function has completed normally
Buffered : **BOOL**;	**TRUE** when motion command is in **NTYPE** buffer
Active : **BOOL**;	**TRUE** when motion command is in **MTYPE** buffer
Aborted : **BOOL**;	**TRUE** if function terminates due to **CANCEL**
Error : **BOOL**;	**TRUE** if a program error is detected
ErrorID : **UINT**;	Returned error number

DESCRIPTION:

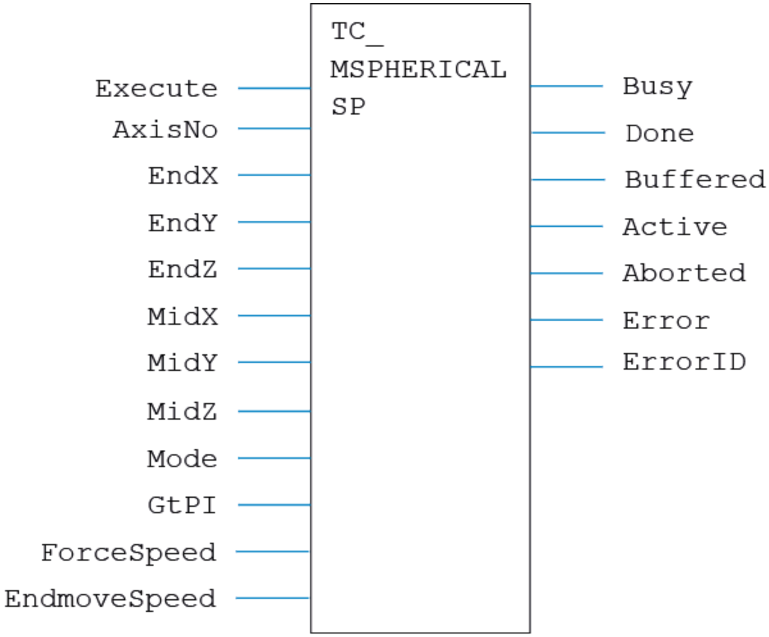
When the execute input changes from **FALSE** to **TRUE** (rising edge), the function block attempts to load the motion command into the required axis buffer. If the buffer is unavailable, the function re-tries on each PLC scan. Once the motion command has been loaded, the appropriate outputs will indicate the state of the motion; in **NTYPE**, **MTYPE**, aborted (Cancelled) or done.

A programming error, such as parameter out of range, will set the Error output and return an error ID number. For the Error ID reference, see the Trio Programming error list.

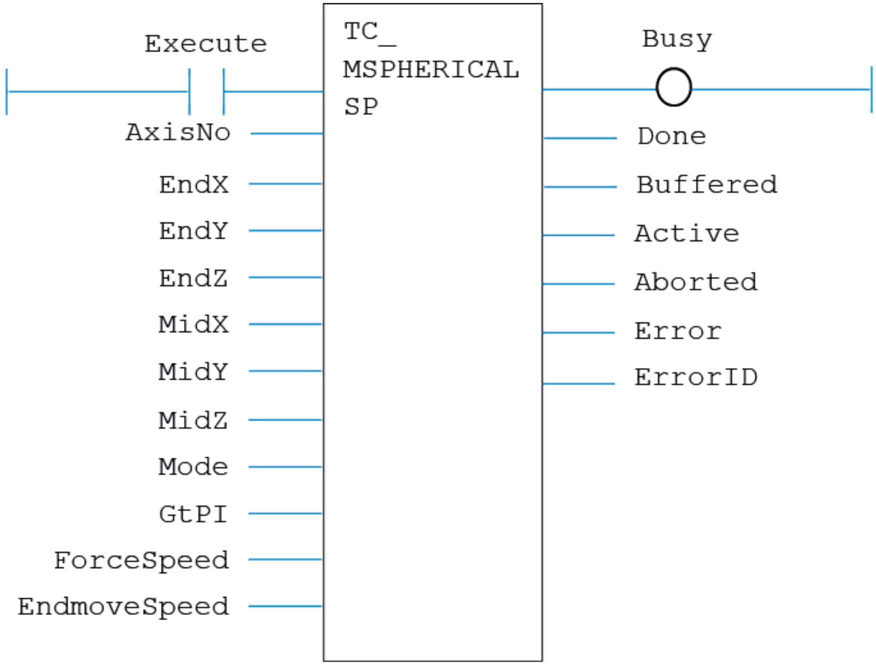
ST LANGUAGE:

```
TC_MSPHERICALSP(Execute, AxisNo, EndX, EndY, EndZ, MidX, MidY, MidZ, Mode, GtPI,
ForceSpeed, EndmoveSpeed, Busy, Done, Buffered, Active, Aborted, Error, ErrorID);
```

FBD LANGUAGE:



LD LANGUAGE:



IL LANGUAGE:
Not available.

TC_OP

TYPE:

I/O Function.

FUNCTION:
Applies a new OP request for the digital output specified.

INPUTS:

Index : INT;	Output number

Value : **SINT**;	Output value

OUTPUTS:

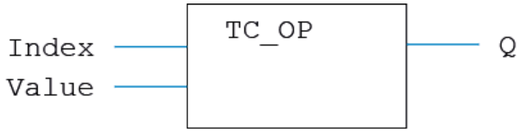
Q : **SINT**;	

DESCRIPTION:
Sets the digital outputs to the binary pattern given in Value.

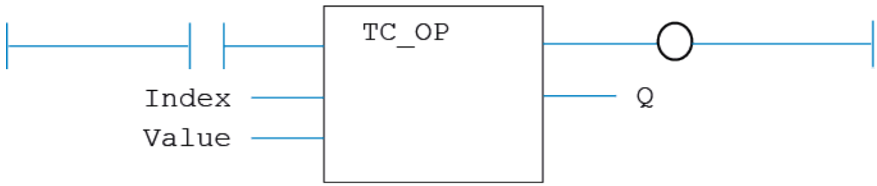
ST LANGUAGE:

```
TC_OP(Index, Value, Q);
```

FBD LANGUAGE:



LD LANGUAGE:



IL LANGUAGE:
Not available.

TC_PSWITCH

TYPE:
Motion Function.

FUNCTION:
Issues a new **PSWITCH** request for the axis specified by ‘AxisNo’.

INPUTS:

Execute : **BOOL**;	Rising edge requests execution
AxisNo : **USINT**;	Axis number
Mode : **USINT**;	PSwitch mode
Switch : **USINT**;	PSwitch number
Output : **USINT**;	Digital output number
OpState : **USINT**;	Output state required when PSwitch is in range
SetPosition : **LREAL**;	Start position where output will assume the defined state
ResetPosition : **LREAL**;	End position where output will go to the opposite state

OUTPUTS:

Done : **BOOL**;	**TRUE** when function has completed normally
Error : **BOOL**;	**TRUE** if a program error is detected
ErrorID : **UINT**;	Returned error number

DESCRIPTION:

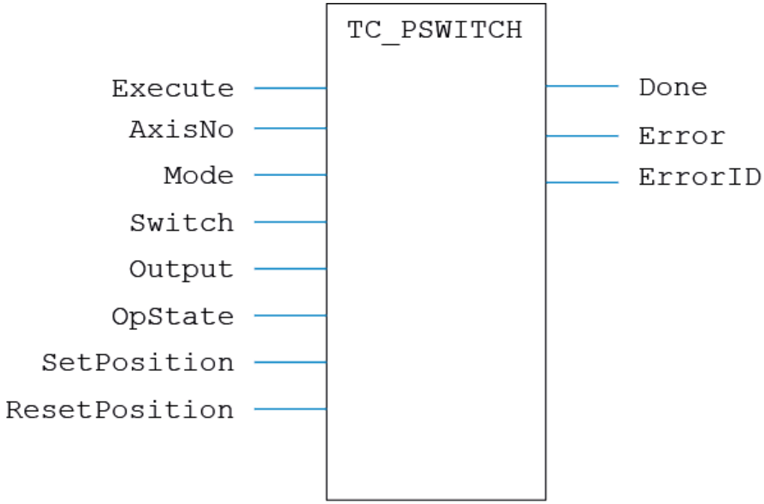
When the execute input changes from **FALSE** to **TRUE** (rising edge), the function block runs the command.

A programming error, such as parameter out of range, will set the Error output and return an error ID number. For the Error ID reference, see the Trio Programming error list.

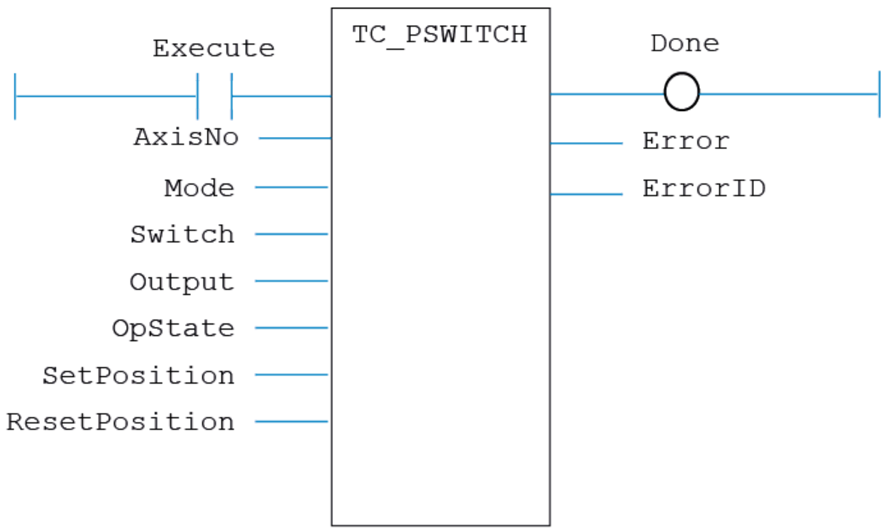
ST LANGUAGE:

```
TC_PSWITCH(Execute, AxisNo, Mode, Switch, Output, OpState,  SetPosition,
ResetPosition, Done, Error, ErrorID);
```

FBD LANGUAGE:



LD LANGUAGE:



IL LANGUAGE:
Not available.

TC_RAPIDSTOP

TYPE:
Motion Function.

FUNCTION:
Issues a new **RAPIDSTOP** motion request for the axis specified by 'AxisNo'.

INPUTS:

Execute : **BOOL**;	Rising edge requests execution
Mode : **USINT**;	**RAPIDSTOP** mode

OUTPUTS:

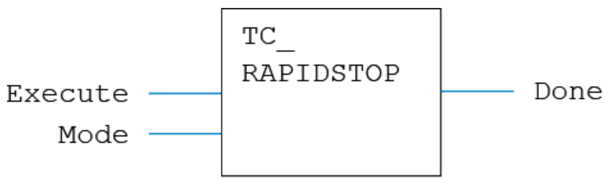
Done : **BOOL**;	**TRUE** when function has completed normally

DESCRIPTION:
When the Execute input changes from **FALSE** to **TRUE** (rising edge), the function block loads the motion command.

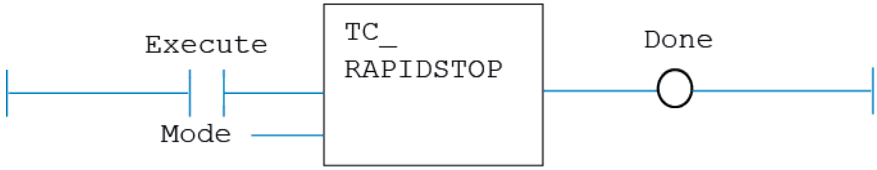
A programming error, such as parameter out of range, will set the Error output and return an error ID number. For the Error ID reference, see the Trio Programming error list.

ST LANGUAGE:
```
TC_RAPIDSTOP(Execute, Mode, Done);
```

FBD LANGUAGE:



LD LANGUAGE:



IL LANGUAGE:
Not available.

TC_READOP

TYPE:
I/O Function.

FUNCTION:
Applies a new **READ_OP** request for the digital output specified.

INPUTS:

Index : INT;	Output number

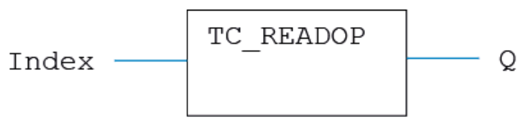
OUTPUTS:

Q : **SINT**;	Output state

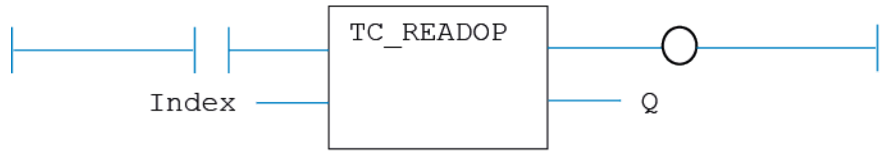
DESCRIPTION:
Sets the digital outputs to the binary pattern given in Value.

ST LANGUAGE:
TC_READOP(Index, Q);

FBD LANGUAGE:



LD LANGUAGE:



IL LANGUAGE:
Not available.

TC_REVERSE

TYPE:
Motion Function.

FUNCTION:
Issues a new **REVERSE** motion request for the axis specified by 'AxisNo'.

INPUTS:

Execute : **BOOL**;	Rising edge requests execution
AxisNo : **USINT**;	Axis number

OUTPUTS:

Busy : **BOOL**;	**TRUE** if function is running
Done : **BOOL**;	**TRUE** when function has completed normally
Buffered : **BOOL**;	**TRUE** when motion command is in **NTYPE** buffer
Active : **BOOL**;	**TRUE** when motion command is in **MTYPE** buffer
Aborted : **BOOL**;	**TRUE** if function terminates due to **CANCEL**
Error : **BOOL**;	**TRUE** if a program error is detected
ErrorID : **UINT**;	Returned error number

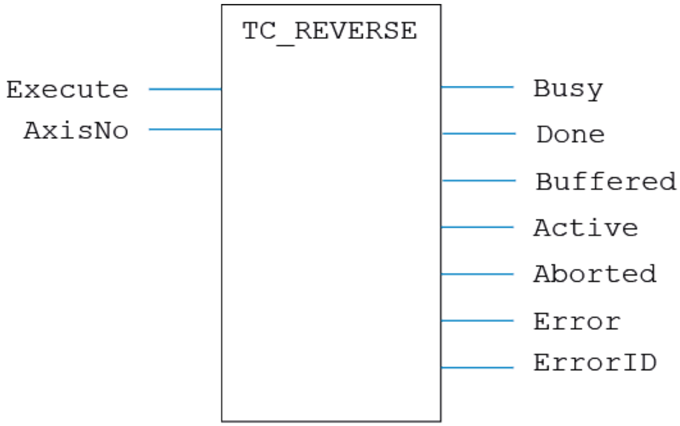
DESCRIPTION:
When the execute input changes from **FALSE** to **TRUE** (rising edge), the function block attempts to load the motion command into the required axis buffer. If the buffer is unavailable, the function re-tries on each PLC scan. Once the motion command has been loaded, the appropriate outputs will indicate the state of the motion; in **NTYPE**, **MTYPE**, aborted (Cancelled) or done.

A programming error, such as parameter out of range, will set the Error output and return an error ID number. For the Error ID reference, see the Trio Programming error list.

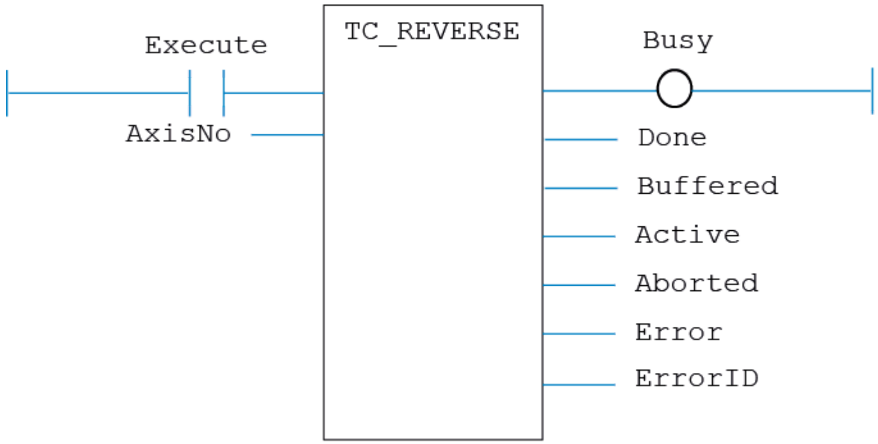
ST LANGUAGE:

```
TC_REVERSE(Execute, AxisNo, Busy, Done, Buffered, Active, Aborted, Error, ErrorID);
```

FBD LANGUAGE:



LD LANGUAGE:



IL LANGUAGE:
Not available.

TC_SELECTTOOLOFFSET

TYPE:
Motion Function.

FUNCTION:
Selects a previously defined **TOOL_OFFSET** to become active.

INPUTS:

EN : **BOOL**;	Set **TRUE** to enable the function
AxisNo : **USINT**;	Axis number
ID : **USINT**;	Tool offset identity number

OUTPUTS:

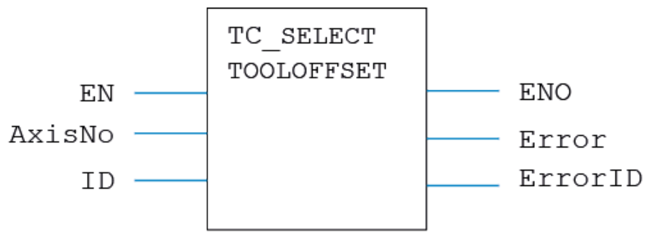
ENO : **BOOL**;	**TRUE** if function is enabled
Error : **BOOL**;	**TRUE** if a program error is detected
ErrorID : **UINT**;	Returned error number

DESCRIPTION:
When the EN input is **TRUE**, the function block applies the command to the axis indicated by AxisNo.

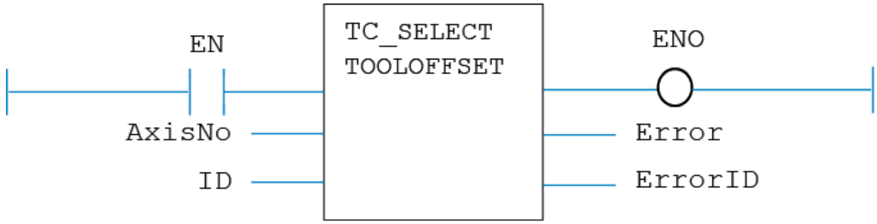
A programming error, such as parameter out of range, will set the Error output and return an error ID number. For the Error ID reference, see the Trio Programming error list.

ST LANGUAGE:
TC_SELECTTOOLOFFSET**(EN, AxisNo, ID, ENO, Error, ErrorID);**

FBD LANGUAGE:



LD LANGUAGE:



IL LANGUAGE:
Not available.

TC_SELECTUSERFRAME

TYPE:
Motion Function.

FUNCTION:
Selects a previously defined **USER_FRAME** to become active.

INPUTS:

EN : **BOOL**;	Set **TRUE** to enable the function
AxisNo : **USINT**;	Axis number
ID : **USINT**;	Tool offset identity number

OUTPUTS:

ENO : **BOOL**;	**TRUE** if function is enabled
Error : **BOOL**;	**TRUE** if a program error is detected
ErrorID : **UINT**;	Returned error number

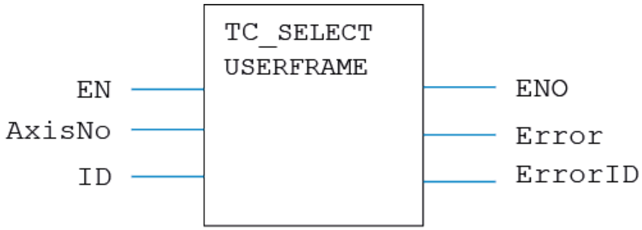
DESCRIPTION:
When the EN input is **TRUE**, the function block applies the command to the axis indicated by AxisNo.

A programming error, such as parameter out of range, will set the Error output and return an error ID number. For the Error ID reference, see the Trio Programming error list.

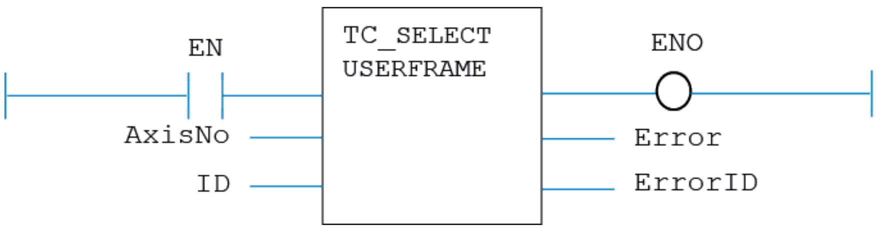
ST LANGUAGE:

```
TC_SELECTUSERFRAME(EN, AxisNo, ID, ENO, Error, ErrorID);
```

FBD LANGUAGE:



LD LANGUAGE:



IL LANGUAGE:

Not available.

TC_SELECTUSERFRAMEB

TYPE:

Motion Function.

FUNCTION:

Selects a secondary **USER_FRAME** to be used when **SYNC** mode 20 is activated.

INPUTS:

EN : **BOOL**;	Set **TRUE** to enable the function
AxisNo : **USINT**;	Axis number
ID : **USINT**;	Tool offset identity number

OUTPUTS:

ENO : **BOOL**;	**TRUE** if function is enabled

Error : **BOOL**;	**TRUE** if a program error is detected
ErrorID : **UINT**;	Returned error number

DESCRIPTION:

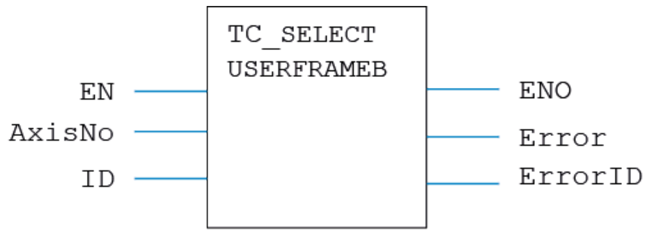
When the EN input is **TRUE**, the function block applies the command to the axis indicated by AxisNo.

A programming error, such as parameter out of range, will set the Error output and return an error ID number. For the Error ID reference, see the Trio Programming error list.

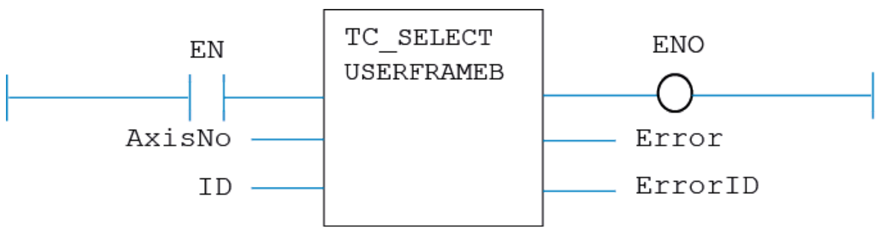
ST LANGUAGE:

TC_SELECTUSERFRAMEB**(EN, AxisNo, ID, ENO, Error, ErrorID);**

FBD LANGUAGE:



LD LANGUAGE:



IL LANGUAGE:

Not available.

TC_SetFRAME

TYPE:

Motion Function.

FUNCTION:
Applies a new **FRAME** request for the axis specified by 'AxisNo'.

INPUTS:

EN : **BOOL**;	Set **TRUE** to enable the function
AxisNo : **USINT**;	Axis number
FRAME : **USINT**;	Frame number to apply

OUTPUTS:

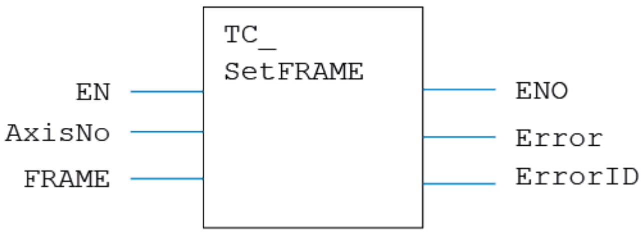
ENO : **BOOL**;	**TRUE** if function is enabled
Error : **BOOL**;	**TRUE** if a program error is detected
ErrorID : **UINT**;	Returned error number

DESCRIPTION:
When the EN input is **TRUE**, the function block applies the command to the axis indicated by AxisNo.

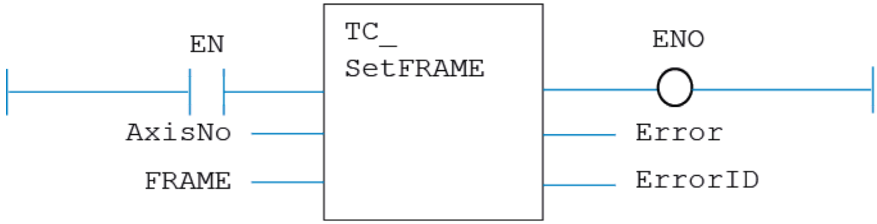
A programming error, such as parameter out of range, will set the Error output and return an error ID number. For the Error ID reference, see the Trio Programming error list.

ST LANGUAGE:
TC _ SetFRAME**(EN, AxisNo, FRAME, ENO, Error, ErrorID);**

FBD LANGUAGE:



LD LANGUAGE:



IL LANGUAGE:

Not available.

TC_STEPRATIO

TYPE:

Motion Function.

FUNCTION:

Issues a new **STEP_RATIO** motion request for the axis specified by ‘AxisNo’.

INPUTS:

EN : **BOOL**;	**TRUE** enables the function
AxisNo : **USINT**;	Axis number
Numerator: **LINT**;	The output count
Denominator: **LINT**;	The **DPOS** count (input of the function)

OUTPUTS:

ENO : **BOOL**;	**TRUE** when function is enabled
Error : **BOOL**;	**TRUE** if a program error is detected
ErrorID : **UINT**;	Returned error number

DESCRIPTION:

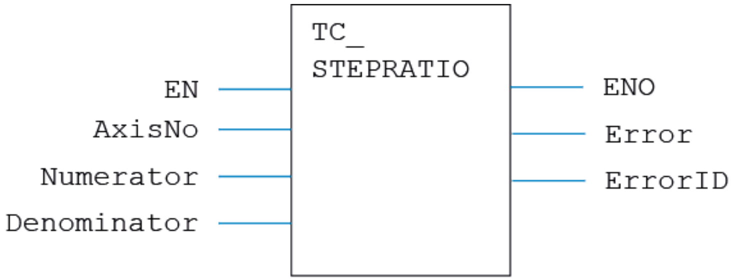
When the EN input is **TRUE**, the function block applies the command to the axis indicated.

A programming error, such as parameter out of range, will set the Error output and return an error ID number. For the Error ID reference, see the Trio Programming error list.

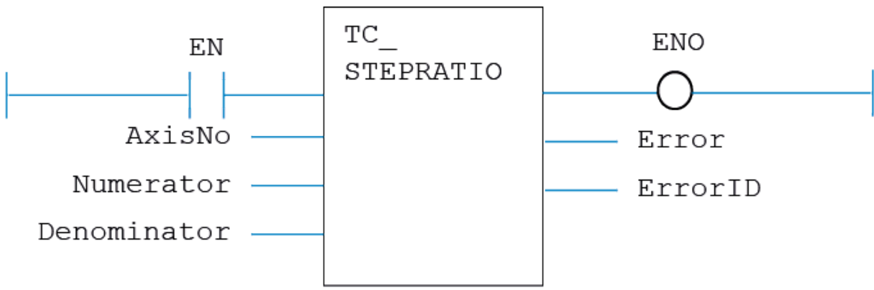
ST LANGUAGE:

```
TC_STEPRATIO(EN, AxisNo, Numerator, Denominator, ENO, Error, ErrorID);
```

FBD LANGUAGE:



LD LANGUAGE:



IL LANGUAGE:
Not available.

TC_SYNC

TYPE:
Motion Function.

FUNCTION:
Issues a new **SYNC** motion request for the axes specified by ‘AxisNo’.

INPUTS:

EN : **BOOL**;	Set **TRUE** to enable the function
AxisNo **: USINT**;	Axis number
Control : **USINT**;	Control value
SyncPos : **LINT**;	Sync Position
SyncAxis **: USINT**;	Master axis to follow
SyncTime : **DINT**;	Time duration for axes to become synchronised
SyncPosX : **LINT**;	Synchronisation position X
SyncPosY : **LINT**;	Synchronisation position Y
SyncPosZ : **LINT**;	Synchronisation position Z

OUTPUTS:

EN : **BOOL**;	**TRUE** if function is enabled
Error : **BOOL**;	**TRUE** if a program error is detected
ErrorID : **UINT**;	Returned error number

DESCRIPTION:

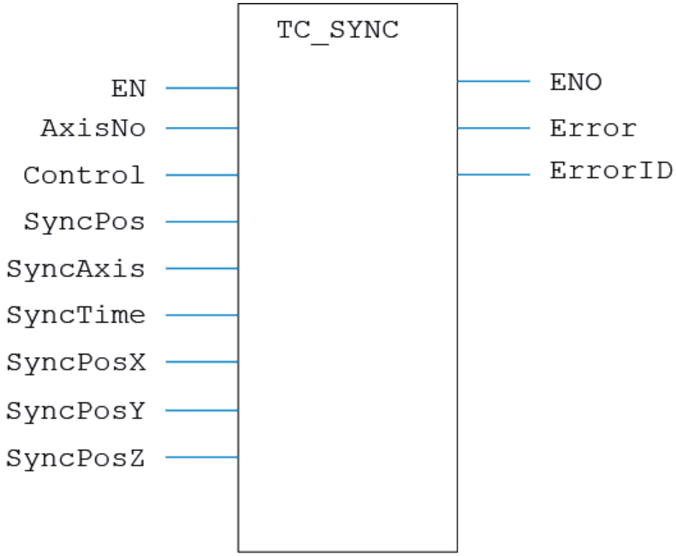
When the EN input is **TRUE**, the function block applies the command to the axis indicated by AxisNo.

A programming error, such as parameter out of range, will set the Error output and return an error ID number. For the Error ID reference, see the Trio Programming error list.

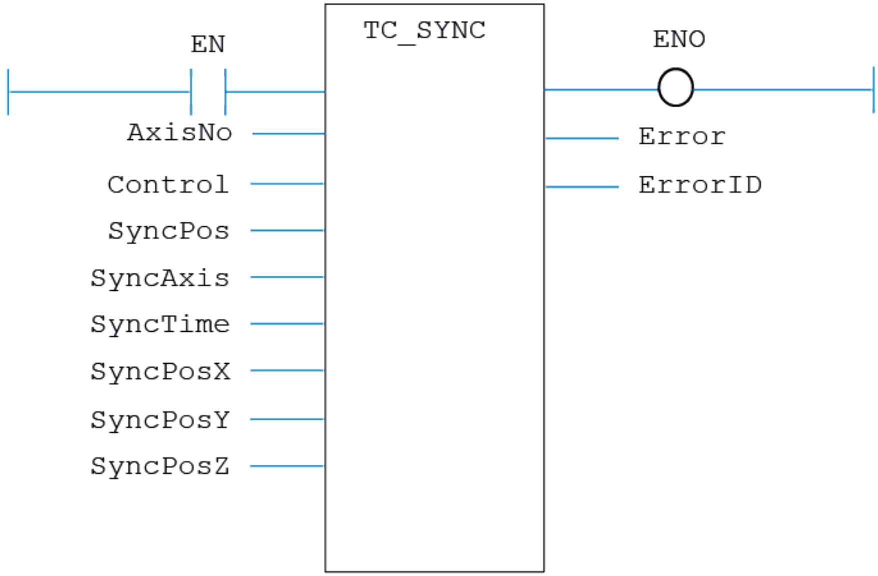
ST LANGUAGE:

```
TC_SYNC(EN, AxisNo, Control, SyncPos, SyncAxis, SyncTime, SyncPosX, SyncPosY,
SyncPosZ, ENO, Error, ErrorID);
```

FBD LANGUAGE:



LD LANGUAGE:



IL LANGUAGE:
Not available.

TC_USERFRAMETRANS

TYPE:
Motion Function.

FUNCTION:
Executes a single **USER_FRAME_TRANS** on the specified table data.

INPUTS:

EN : **BOOL**;	Set **TRUE** to enable the function
UF1 : **USINT**;	User Frame In; The **USER_FRAME** identity that the points are supplied in
UF2 : **USINT**;	User Frame Out; The **USER_FRAME** identity that the points are transformed to

TO1 : **USINT**;	Tool Offset In; The **TOOL_OFFSET** identity that the points are supplied in
TO2 : **USINT**;	Tool Offset Out; The **TOOL_OFFSET** identity that the points are transformed to
DataIn : **DINT**;	The table index for the input positions
DataOut : **LINT**;	The table index for the start of the generated positions
Scale : **LREAL**;	Scale factor for the table values (default 1000)

OUTPUTS:

EN : **BOOL**;	**TRUE** if function is enabled
Error : **BOOL**;	**TRUE** if a program error is detected
ErrorID : **UINT**;	Returned error number

DESCRIPTION:

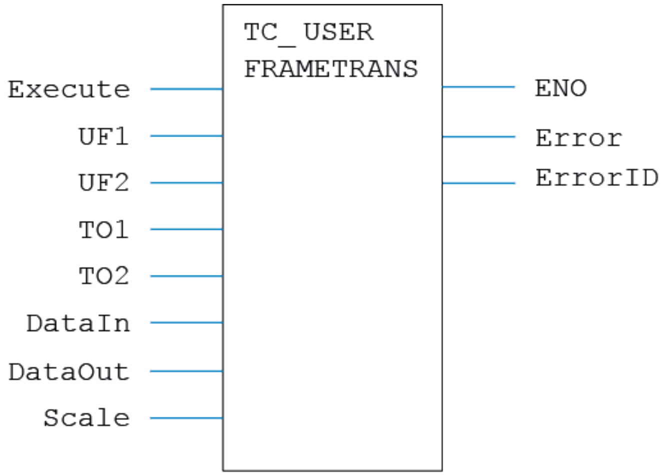
When the EN input is **TRUE**, the function block applies the command.

A programming error, such as parameter out of range, will set the Error output and return an error ID number. For the Error ID reference, see the Trio Programming error list.

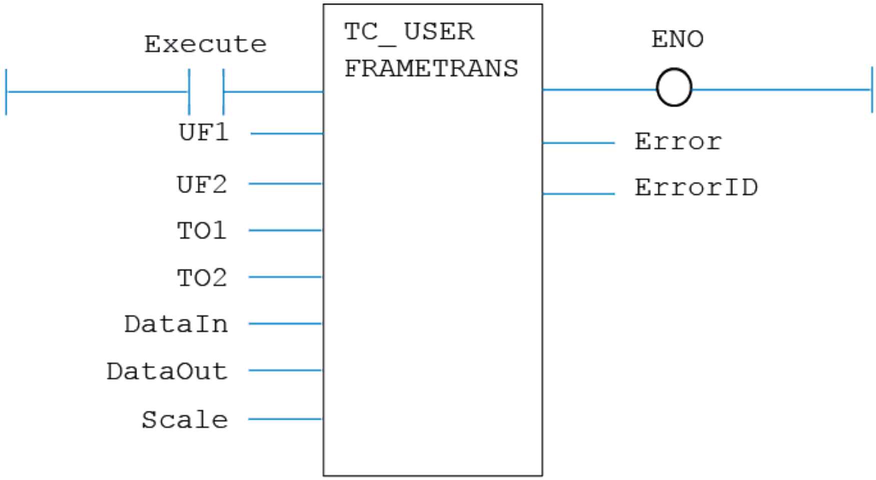
ST LANGUAGE:

TC_USERFRAMETRANS(EN, UF1, UF2, TO1, TO2, DataIn, DataOut, Scale, ENO, Error, ErrorID);

FBD LANGUAGE:



LD LANGUAGE:



IL LANGUAGE:
Not available.

TC_VOLUMELIMIT

TYPE:
Motion Function.

FUNCTION:
Configures a new 3D **VOLUME_LIMIT**.

INPUTS:

EN : **BOOL**;	Set **TRUE** to enable the function
AxisNo : **USINT**;	Axis number
Mode : **USINT**;	**VOLUME_LIMIT** mode
TableIndex : **DINT**	Location of table data for **VOLUME_LIMIT**

OUTPUTS:

ENO : **BOOL**;	**TRUE** if function is enabled

Error : **BOOL**;	**TRUE** if a program error is detected
ErrorID : **UINT**;	Returned error number

DESCRIPTION:

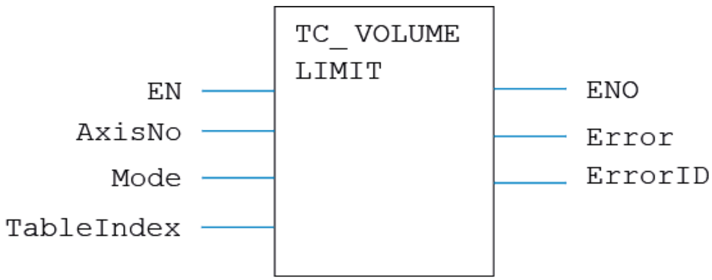
When the EN input is **TRUE**, the function block applies the command to the axis indicated by AxisNo.

A programming error, such as parameter out of range, will set the Error output and return an error ID number. For the Error ID reference, see the Trio Programming error list.

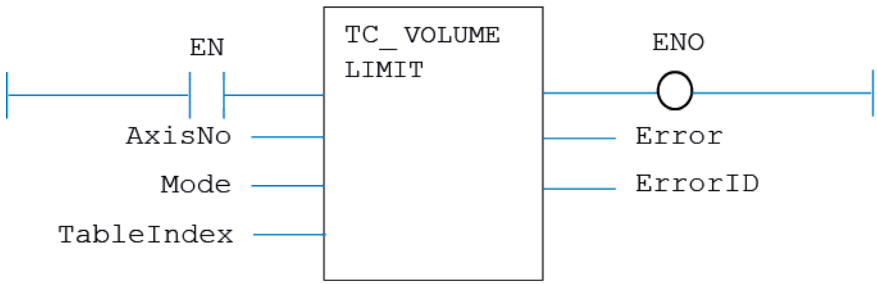
ST LANGUAGE:

TC_VOLUMELIMIT**(EN, AxisNo, Mode, TableIndex, ENO, Error, ErrorID);**

FBD LANGUAGE:



LD LANGUAGE:



IL LANGUAGE:

Not available.

TCR_AxisParameter

TYPE:
Axis Parameter.

FUNCTION:
Reads from the named axis parameter.

INPUTS:

AxisNo : **USINT**;	Axis number

OUTPUTS:

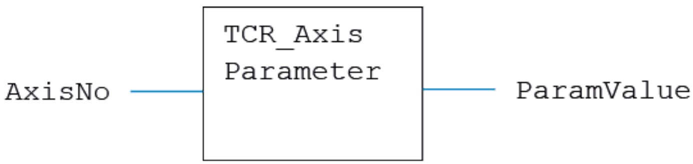
ParamValue : Various;	Parameter value

DESCRIPTION:
Reads the value of AxisParameter. Value is returned in ParamValue. See the function block tooltips in the Motion Perfect v3 editor for parameter names and data sizes.

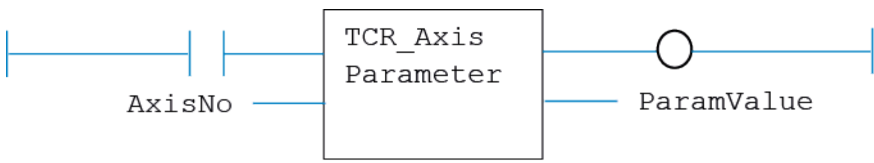
ST LANGUAGE:

```
TCR _ AxisParameter(AxisNo, ParamValue);
```

FBD LANGUAGE:



LD LANGUAGE:



IL LANGUAGE:
Not available.

TCR_ErrorID

TYPE:
System Parameter.

FUNCTION:
Reads the latest error produced by any of the TCR/TCW functions.

INPUTS:

None	

OUTPUTS:

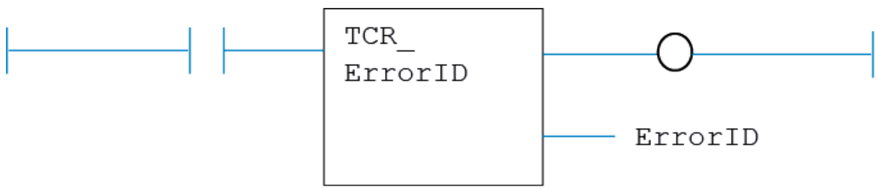
ErrorID : **UINT**;	Error ID value

DESCRIPTION:
Reads the Error ID value caused by the most recent TCR or TCW function to be processed.

ST LANGUAGE:
```
TCR _ ErrorID(ErrorID);
```

FBD LANGUAGE:

LD LANGUAGE:



IL LANGUAGE:
Not available.

TCR_TABLE

TYPE:
Motion Parameter.

FUNCTION:
Reads from a **TABLE** entry.

INPUTS:

Index : INT;	**TABLE** Index number

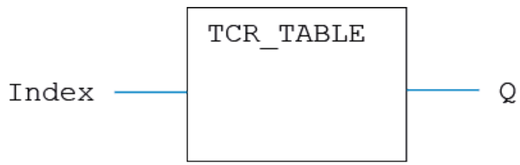
OUTPUTS:

Q : **LREAL**;	**TABLE** value

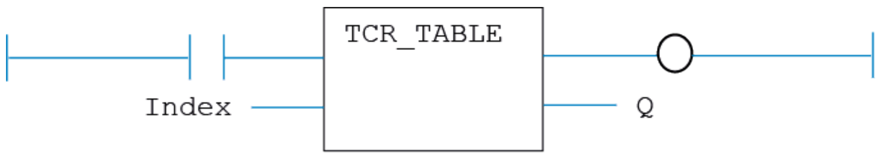
DESCRIPTION:
Reads from the **TABLE** variable number indicated in Index. Value is returned in Q.

ST LANGUAGE:
TCR_TABLE(Index, Q);

FBD LANGUAGE:



LD LANGUAGE:



IL LANGUAGE:
Not available.

TCR_TICKS

TYPE:
Motion Parameter.

FUNCTION:
Reads from the process **TICKS** value.

INPUTS:

None	

OUTPUTS:

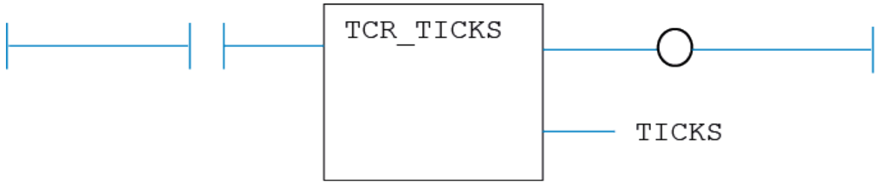
TICKS : **LINT**;	**TICKS** value

DESCRIPTION:
Reads from the **TICKS** value associated with the current process. Value is returned in **TICKS**.

ST LANGUAGE:
TCR_TICKS(TICKS);

FBD LANGUAGE:

LD LANGUAGE:



IL LANGUAGE:
Not available.

TCR_VR

TYPE:
Motion Parameter.

FUNCTION:
Reads from a VR variable.

INPUTS:

Index : INT;	VR Index number

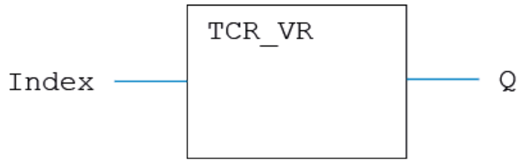
OUTPUTS:

Q : **LREAL**;	VR value

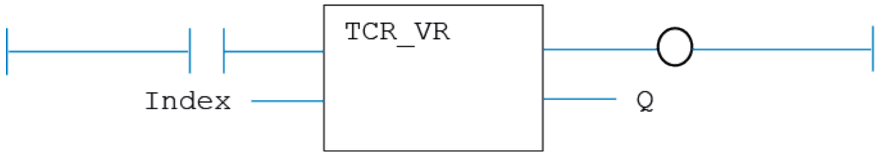
DESCRIPTION:
Reads from the VR variable number indicated in Index. Value is returned in Q.

ST LANGUAGE:
```
TCR_VR(Index, Q);
```

FBD LANGUAGE:



LD LANGUAGE:



IL LANGUAGE:
Not available.

TCR_WDOG

TYPE:
System Parameter.

FUNCTION:
Reads the state of the **WDOG** system variable.

INPUTS:

None	

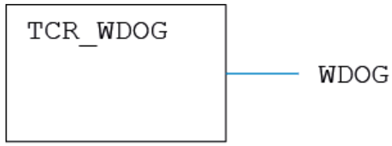
OUTPUTS:

WDOG : **DINT**;	**WDOG** state

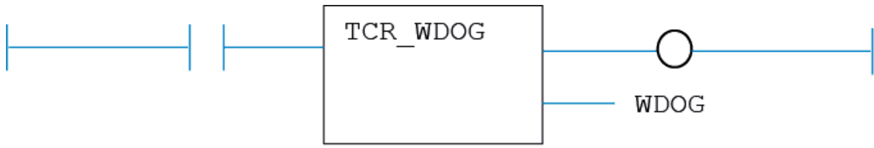
DESCRIPTION:
Reads the current **WDOG** state.

ST LANGUAGE:
TCR_WDOG(WDOG);

FBD LANGUAGE:



LD LANGUAGE:



IL LANGUAGE:
Not available.

TCW_AxisParameter

TYPE:
Axis Parameter.

FUNCTION:
Writes to the named axis parameter.

INPUTS:

AxisNo : **USINT**;	Axis number
ParamValue : Various;	Parameter value

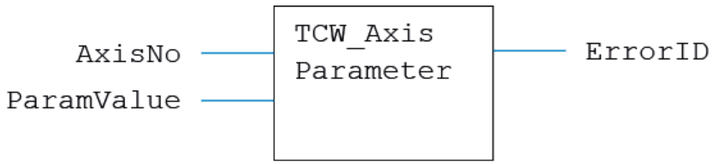
OUTPUTS:

ErrorID : **UINT**;	Error ID number

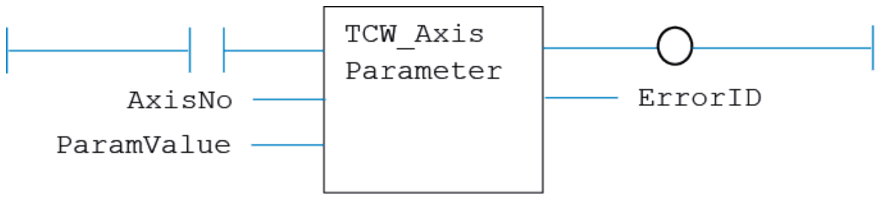
DESCRIPTION:
Writes the specified value to the AxisParameter. See the function block tooltips in the Motion Perfect v3 editor for parameter names and data sizes.

ST LANGUAGE:
```
TCW _ AxisParameter(AxisNo, ParamValue, ErrorID);
```

FBD LANGUAGE:



LD LANGUAGE:



IL LANGUAGE:
Not available.

TCW_TABLE

TYPE:
System Function.

FUNCTION:
Writes to a **TABLE** location.

INPUTS:

Index : INT;	**TABLE** index number
Value : **LREAL**;	**TABLE** value

OUTPUTS:

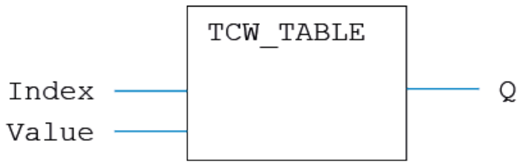
Q : **SINT**;	

DESCRIPTION:
Sets the VR at VR(index) to the given Value.

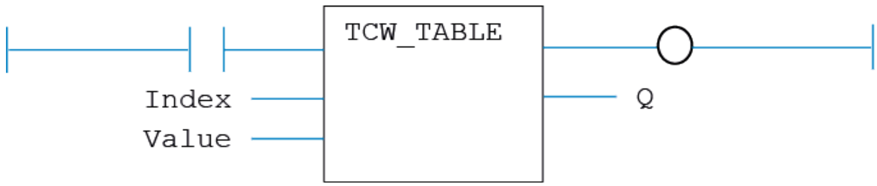
ST LANGUAGE:

```
TCW_TABLE(Index, Value, Q);
```

FBD LANGUAGE:



LD LANGUAGE:



IL LANGUAGE:
Not available.

TCW_TICKS

TYPE:
System Function.

FUNCTION:
Writes to the process **TICKS** value.

INPUTS:

TICKS : **LINT**;	**TICKS** value

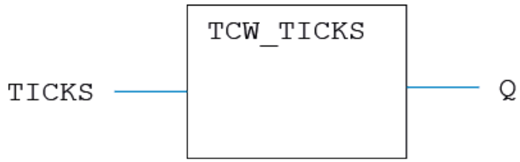
OUTPUTS:

Q : **DINT**;	

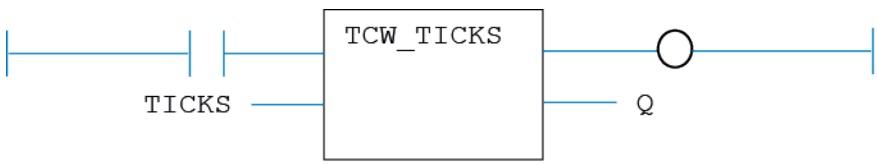
DESCRIPTION:
Sets the **TICKS** value in the current process.

ST LANGUAGE:
TCW_TICKS(TICKS, Q);

FBD LANGUAGE:



LD LANGUAGE:



IL LANGUAGE:
Not available.

TCW_VR

TYPE:
System Function.

FUNCTION:
Writes to a VR variable.

INPUTS:

Index : INT;	VR number
Value : **LREAL**;	VR value

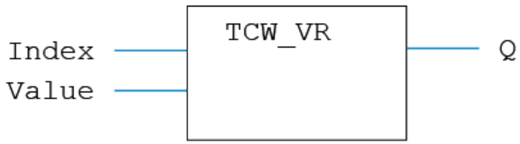
OUTPUTS:

Q : **SINT**;	

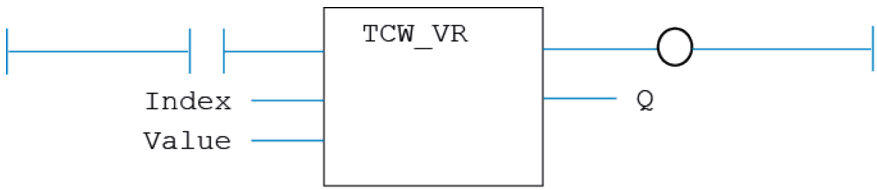
DESCRIPTION:
Sets the VR at VR(index) to the given Value.

ST LANGUAGE:
```
TCW_VR(Index, Value, Q);
```

FBD LANGUAGE:



LD LANGUAGE:



IL LANGUAGE:
Not available.

TCW_WDOG

TYPE:
System Function.

FUNCTION:
Writes to the **WDOG** parameter.

INPUTS:

WDOG : **DINT**;	**WDOG** state

OUTPUTS:

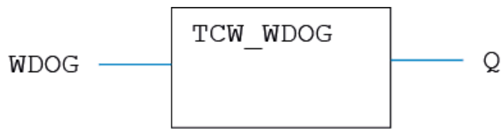
Q : **DINT**;	

DESCRIPTION:
Sets the **WDOG** state.

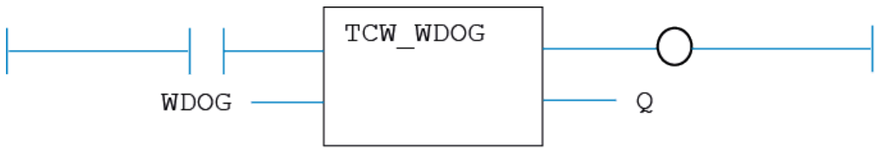
ST LANGUAGE:

```
TCW_WDOG(WDOG, Q);
```

FBD LANGUAGE:



LD LANGUAGE:



IL LANGUAGE:
Not available.

4
STANDARD IEC 61131-3 COMMANDS

Contents

Introduction to the Standard IEC Language

Below are the available programming languages of the IEC61131-3 standard:

SFC: Sequential Function Chart

FBD: Function Block Diagram

LD: Ladder Diagram

ST: Structured Text

Use of ST instructions in graphic languages

You have to select a language for each program or User Defined Function Block of the application.

Sequential Function Chart (SFC)

The SFC language is a state diagram. Graphical steps are used to represent stable states, and transitions describe the conditions and events that lead to a change of state. Using SFC highly simplifies the programming of sequential operations as it saves a lot of variables and tests just for maintaining the program context.

YOU MUST NOT USE SFC AS A DECISION DIAGRAM. USING A STEP AS A POINT OF DECISION AND TRANSITIONS AS CONDITIONS IN AN ALGORITHM SHOULD NEVER APPEAR IN A SFC CHART. USING SFC AS A DECISION LANGUAGE LEADS TO POOR PERFORMANCE AND COMPLICATE CHARTS. ST MUST BE PREFERRED WHEN PROGRAMMING A DECISION ALGORITHM THAT HAS NO SENSE IN TERM OF "PROGRAM STATE".

Below are basic components of an SFC chart:

Chart	Programming
Steps and initial steps	Actions within a step
Transitions and divergences	Timeout on a step
Parallel branches	Programming a transition condition
Jump to a step	How SFC is executed
	UDFBs programmed in SFC

The workbench fully supports SFC programming with several hierarchical levels of charts: i.e. a chart that controls another chart. Working with a hierarchy of SFC charts is an easy and powerful way for managing complex sequences and saves performances at run time. Refer to the following sections for further details:

Hierarchy of SFC programs

Controlling a SFC child program

Actions in a SFC step

Each step has a list of action blocks, that are instructions to be executed according to the activity of the step. Actions can be simple boolean or SFC actions, that consists in assigning a boolean variable or control a child SFC program using the step activity, or action blocks entered using another language (FBD, LD or ST).

RUNTIME CHECK

Below are the possible syntaxes you can use within an SFC step to perform runtime safety checks:

Syntax	Description
StepTimeout (...);	Check for a timeout on the step activity duration.

SIMPLE BOOLEAN ACTIONS

Below are the possible syntaxes you can use within an SFC step to perform a simple boolean action:

Syntax	Description
BoolVar (N);	Forces the variable BoolVar to TRUE when the step is activated, and to FALSE when the step is de-activated.
BoolVar (S);	Sets the variable BoolVar to TRUE when step is activated
BoolVar (R);	Sets the variable BoolVar to FALSE when step is activated
/ BoolVar;	Forces the variable BoolVar to FALSE when the step is activated, and to TRUE when the step is de-activated.

ALARMS

The following syntax enables you to manage timeout alarm variables:

Syntax	Description
BoolVar (A, duration);	Specifies a timeout variable to be associated to the step. BoolVar must be a simple boolean variable duration is the timeout, expressed eiter as a constant or as a single TIME variable (complex expressions cannot be used for this parameter) When the timeout is elapsed, the alarm variable is turned to TRUE, and the transition(s) following the step cannot be crossed until the alarm variable is reset.

SIMPLE SFC ACTIONS

Below are the possible syntaxes you can use within an SFC step to control a child SFC program:

Syntax	Description
Child (N);	Starts the child program when the step is activated and stops (kills) it when the step is de-activated.
Child (S);	Starts the child program when the step is activated

Syntax	Description
Child (R);	Stops (kills) the child program when the step is activated

PROGRAMMED ACTION BLOCKS

Programs in other languages (FBD, LD or ST) can be entered to describe an SFC step action. There are three main types of programmed action blocks, that correspond to the following identifiers:

Identifier	Description
P1	Executed only once when the step becomes active.
N	Executed on each cycle while the step is active.
P0	Executed only once when the step becomes inactive.

The workbench provides you templates for entering P1, N and P0 action blocks in either ST, LD or FBD language. Alternatively, you can insert action blocks programmed in ST language directly in the list of simple actions, using the following syntax:

```
ACTION ( qualifier ) :
    statements...
END_ACTION;
```

Where qualifier is P1, 0 or P0.

CHECK TIMEOUT ON A SFC STEP

The system can check timeout on any SFC step activity duration. For that you need to enter the following instruction in the main "Action" list of the step:

```
_ _StepTimeout ( timeOut , errString );
```

Where:

timeout is a time constant or a time variable specifying the timeout duration.
errString is a string constant or a string variable specifying the error message to be output.

At runtime, each time the activation time of the step becomes greater than the specified timeout, the error string is sent to the Workbench and displayed in the Log window.

SENDING LOG MESSAGE STRINGS TO THE LOG WINDOW REQUIRES THE RUNTIME TO BE CONNECTED THROUGH ETHERNET, AND THAT YOUR T5 RUNTIME SYSTEM SUPPORTS PLAIN TEXT TRACE MESSAGES.

You can also put this statement within a #ifdef **__DEBUG** test so that timeout checking is enabled only in debug mode.

Alternatively, if you need to make more specific handling of timeouts, you can enter the following **ST** program in the "N" action block of the step:

```
if GSn.T > timeout then /* 'n' is the number of the step */
   ...statements...
end_if;
```

CONDITION OF A SFC TRANSITION

Each SFC transitions must have a boolean condition that indicates if the transition can be crossed. The condition is a boolean expression that can be programmed either in ST or LD language.

In ST language, enter a boolean expression. In can be a complex expression including function calls and parenthesis.

EXAMPLE

```
bForce AND (bAlarm OR min (iLevel, 1) <> 1)
```

In LD language, the condition is represented by a single rung. The coil at the end of the rung represents the transition and should have no symbol attached.

EXAMPLE

CONTROLLING A SFC CHILD PROGRAM

Controlling a child program may be simply achieved by specifying the name of the child program as an action block in a step of its parent program. Below are possible qualifiers that can be applied to an action block for handling a child program:

Qualifier	Description
Child (N);	Starts the child program when the step is activated and stops (kills) it when the step is de-activated.
Child (S);	Starts the child program when the step is activated. (Inital steps of the child program are activated)
Child (R);	Stops (kills) the child program when the step is activated. (All active steps of the child program are deactivated)

Alternatively, you can use the following statements in an action block programmed in ST language. In the following table, prog represents the name of the child program:

Statement	Description
GSTART (prog);	Starts the child program when the step is activated. (Inital steps of the child program are activated)
GKILL (prog);	Stops (kills) the child program when the step is activated. (All active steps of the child program are deactivated)
GFREEZE (prog);	Suspends the execution of a child program.
GRST (prog);	Restarts a program suspended by a GFREEZE command.

You can also use the "**GSTATUS**" function in expressions. This function returns the current state of a child SFC program:

Statement	Description
GSTATIS (prog);	Returns the current state of a child SFC program: 0: program is inactive 1: program is active 2: program is suspended

When a child program is started by its parent program, it keeps the inactive status until it is executed (further in the cycle). If you start a child program in a **SFC** chart, **GSTATUS** will return 1 (active) on the next cycle.

Hierarchy of SFC programs

Each SFC program may have one or more “child programs”. Child programs are written in SFC and are started (launched) or stopped (killed) in the actions of the father program. A child program may also have children. The number of hierarchy levels should not exceed 19.

When a child program is stopped, its children are also implicitly stopped.

When a child program is started, it must excplicitly in its actions start its children.

A child program is controlled (started or stopped) from the action blocks of its parent program. Designing a child program is a simple way to program an action block in SFC language.

Using child programs is very useful for designing a complex process and separate operations due to different aspects of the process. For instance, it is common to manage the execution modes in a parent program and to handle details of the process operations in child programs.

JUMP TO A SFC STEP

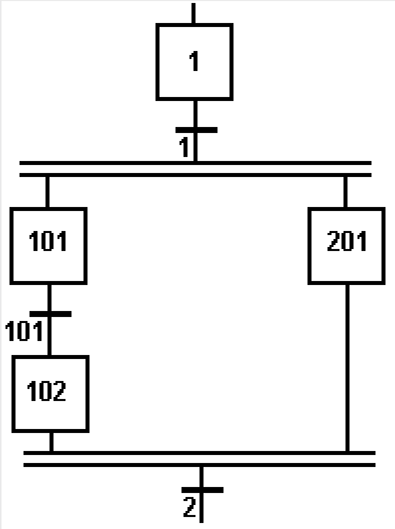
Jump symbols can be used in SFC charts to represent a link from a transition to a step without actually drawing it. The jump is represented by an arrow identified with the number of the target step.

To change the number of a step, transition or jump, select it and hit Ctrl+**ENTER** keys.

You cannot insert a jump to a transition as it may lead to a non explicit convergence of parallel branches (several steps leading to the same transition) and generally leads to mistakes due to a bad understanding of the chart.

SFC PARALLEL BRANCHES

Parallel branches are used in SFC charts to represent parallel operations. Parallel branches occur when more than several steps are connected after the same transition. Parallel branches are drawn as double horizontal lines:



When the transition before the divergence (1 on this example) is crossed, all steps beginning the parallel branches (101 and 201 here) are activated.
Processing of parallel branches may take different timing according to each branch execution.
The transition after the convergence (2 on this example) is crossed when all the steps connected before the convergence line (last step of each branch) are active. The transition indicates a synchronization of all parallel branches.
If needed, a branch may be finished with an empty step (with no action). It represents the state where the branch “waits” for the other ones to be completed.

You must take care of the following rules when drawing parallel lines in order to avoid dead locks in the execution of the program:

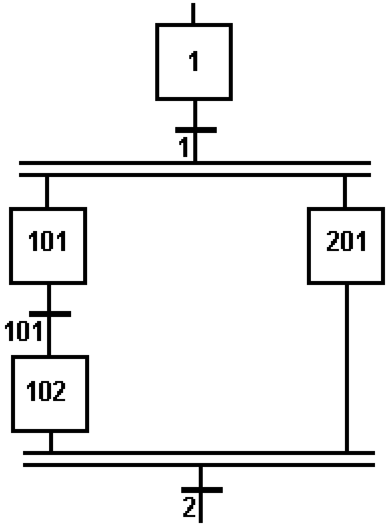
- All branches must be connected to the divergence and the convergence.
- An element of a branch must not be connected to an element outside the divergence.

SFC execution at run time

SFC programs are executed sequentially within a target cycle, according to the order defined when entering programs in the hierarchy tree. A parent SFC program is executed before its children. This implies that when a parent starts or stops a child, the corresponding actions in the child program are performed during the same cycle.

Within a chart, all valid transitions are evaluated first, and then actions of active steps are performed. The chart is evaluated from the left to the right and from the top to the bottom.

EXAMPLE



Execution order:

- Evaluate transitions:
- 1, 101, 2
- Manage steps:
- 1, 101, 201, 102

In case of a divergence, all conditions are considered as exclusive, according to a left to right priority order. It means that a transition is considered as **FALSE** if at least one of the transitions connected to the same divergence on its left side is **TRUE**.

The initial steps define the initial status of the program when it is started. All top level (main) programs are started when the application starts. Child programs are explicitly started from action blocks within the parent programs.

The evaluation of transitions leads to changes of active steps, according to the following rules:

A transition is crossed if:

- its condition is **TRUE**.
- and if all steps linked to the top of the transition (before) are active.

When a transition is crossed:

- all steps linked to the top of the transition (before) are de-activated.
- all steps linked to the bottom of the transition (after) are activated.

EXECUTION OF SFC WITHIN THE T5 TARGET IS SAMPLED ACCORDING TO THE TARGET CYCLES. WHEN A TRANSITION IS CROSSED WITHIN A CYCLE, THE FOLLOWING STEPS ARE ACTIVATED, AND THE EVALUATION OF THE CHART WILL CONTINUE ON THE NEXT CYCLE. IF SEVERAL CONSECUTIVE TRANSITIONS ARE TRUE WITHIN A BRANCH, ONLY ONE OF

THEM IS CROSSED WITHIN ONE TARGET CYCLE.

THIS SECTION DESCRIBES THE EXECUTION MODEL OF A STANDARD T5 TARGET. SFC EXECUTION RULES MAY DIFFER FOR OTHER TARGET SYSTEMS. PLEASE REFER TO OEM INSTRUCTIONS FOR FURTHER DETAILS ABOUT SFC EXECUTION AT RUN TIME.

SOME RUN-TIME SYSTEMS MAY NOT SUPPORT EXCLUSIVITY OF THE TRANSITIONS WITHIN A DIVERGENCE. PLEASE REFER TO OEM INSTRUCTIONS FOR FURTHER INFORMATION ABOUT SFC SUPPORT.

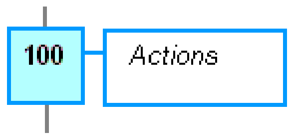
SFC STEPS

A step represents a stable state. It is drawn as a square box in the SFC chart. Each must step of a program is identified by a unique number. At run time, a step can be either active or inactive according to the state of the program.

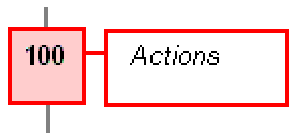
To change the number of a step, transition or jump, select it and hit Ctrl+**ENTER** keys.

All actions linked to the steps are executed according to the activity of the step.

Inactive step



Active step



In conditions and actions of the SFC program, you can test the step activity by specifying its name (“GS” plus the step number) followed by “.X”.

EXAMPLE

GS100.X Is **TRUE** if step 100 is active.
(Expression has the **BOOL** data type).

You can also test the activity time of a step, by specifying the step name followed by “.T”. It is the time elapsed since the activation of the step. When the step is de-activated, this time remains unchanged. It will be reset to 0 on the next step activation.

EXAMPLE

GS100.T Is the time elapsed since step 100 was activated.
(Expression has the **TIME** data type).

INITIAL STEPS

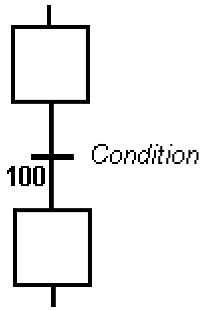
Initial steps represent the initial situation of the chart when the program is started. There must be at least one initial step in each SFC chart. An initial step is marked with a double line:

SFC TRANSITIONS

Transitions represent a condition that changes the program activity from a step to another.

To change the number of a step, transition or jump, select it and hit Ctrl+**ENTER** keys.

The transition is marked by a small horizontal line that crosses a link drawn between the two steps:



Each transition is identified by a unique number in the SFC program.
Each transition must be completed with a boolean condition that indicates if the transition can be crossed. The condition is a BOOL expression.
In order to simplify the chart and reduce the number of drawn links, you can specify the activity flag of a step (GSnnn.X) in the condition of the transition.

Transitions define the dynamic behaviour of the SFC chart, according to the following rules:

A transition in crossed if:

- its condition is **TRUE**.
- and if all steps linked to the top of the transition (before) are active.

When a transition is crossed:

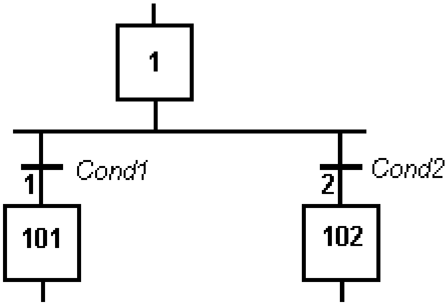
- all steps linked to the top of the transition (before) are de-activated.
- all steps linked to the bottom of the transition (after) are activated.

DIVERGENCES

It is possible to link a step to several transitions and thus create a divergence. The divergence is represented by a horizontal line. Transitions after the divergence represent several possible changes in the situation of the program.

All conditions are considered as exclusive, according to a left to right priority order. It means that a transition is considered as **FALSE** if at least one of the transitions connected to the same divergence on its left side is **TRUE**.

EXAMPLE



Transition 1 is crossed if:
step 1 is active
and Cond1 is **TRUE**

Transition 2 is crossed if:
step 1 is active
and Cond2 is **TRUE**
and Cond1 is **FALSE**

SOME RUN-TIME SYSTEMS MAY NOT SUPPORT EXCLUSIVITY OF THE TRANSITIONS WITHIN A DIVERGENCE. PLEASE REFER TO OEM INSTRUCTIONS FOR FURTHER INFORMATION ABOUT SFC SUPPORT.

User Defined Function Blocks programmed in SFC

The Workbench enables you to create User Defined Function Blocks (UDFBs) programmed with SFC language. This section details specific features related to such function blocks.

The execution of UDFBs written in SFC requires a runtime system version SR7-1 or later.

DECLARATION

From the Workspace contextual menu, run the Insert New Program command. Then specify a valid name for the function block. Select “SFC” language and “**UDFB**” execution style.

PARAMETERS

When a **UDFB** programmed in SFC is created, the Workbench automatically declares 3 special inputs to the block:

RUN: The SFC state machine is not activated when this input is **FALSE**.
RESET: The SFC chart is reset to its initial situation when this input is **TRUE**.
KILL: Any active step of the SFC chart is deactivated when this input is **TRUE**.

You can freely add other input and output variables to the **UDFB**. You can also remove any of the automatically created input if not needed. If the RUN input is removed, then it is considered as always

TRUE. If **RESET** or **KILL** inputs are removed, then they are considered as always **FALSE**.

Below is the truth table showing priorities among special input:

RUN	RESET	KILL	Description
FALSE	FALSE	FALSE	do nothing
FALSE	FALSE	TRUE	kill the SFC chart
FALSE	TRUE	TRUE	reset the SFC chart
FALSE	TRUE	FALSE	kill the SFC chart
TRUE	FALSE	TRUE	activate the SFC chart
TRUE	FALSE	FALSE	kill the SFC chart
TRUE	TRUE	TRUE	reset the SFC chart
TRUE	TRUE	FALSE	kill the SFC chart

STEPS

All steps inserted in the SFC chart of the **UDFB** are automatically declared as local instances of special reserved function blocks with the local variables of the UDFBs. The following FB types are used:

isfcSTEP : a normal step
isfcINITSTEP : an initial step

The editor takes care of updating the list of declared step instances. You should never remove, rename or change them in the variable editor. All steps are named with GS followed by their number.

EXECUTION

The SFC chart is operated only when the **UDFB** is called by its parent program.

If the **RESET** input is **TRUE**, the SFC chart is reset to its initial situation. If the **KILL** input is **TRUE**, any active step of the SFC chart is deactivated.

When the RUN input is **TRUE** and **KILL**/**RESET** are **FALSE**, the SFC chart is operated in the same way as for other SFC programs:

- Check valid transitions and evaluate related conditions.
- Cross **TRUE** valid transitions.
- Execute relevant actions of the active steps.

In a **UDFB** programmed in **SFC**, you cannot use **SFC** actions to pilot a “child **SFC** program”. This feature is reserved for **SFC** programs only. Instead, a **UDFB** programmed in **SFC** can pilot from its actions another **UDFB** programmed in **SFC**.

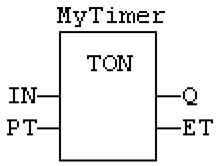
Function Block Diagram (FBD)

A Function Block Diagram is a data flow between constant expressions or variables and operations represented by rectangular blocks. Operations can be basic operations, function calls, or function block calls.

USE OF ST INSTRUCTIONS IN GRAPHIC LANGUAGES

The name of the operation or function, or the type of function block is written within the block rectangle. In case of a function block call, the name of the called instance must be written upon the block rectangle, such as in the example below:

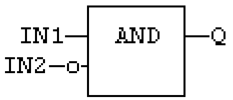
EXAMPLE



The data flow may represent values of any data type. All connections must be from input and outputs points having the same data type. In case of a boolean connection, you can use a connection link terminated by a small circle, that indicates a boolean negation of the data flow.

EXAMPLE

Use of a negated link: Q is IN1 AND NOT IN2!



The data flow must be understood from the left to the right and from the top to the bottom. It is possible to use labels and jumps to change the default data flow execution.

LD SYMBOLS

LD symbols may also be entered in FBD diagrams and linked to FBD objects. Refer to the following sections for further information about components of the LD language:

Contacts, Coils, Power Rails

Special vertical lines are available in FBD language for representing the merging of LD parallel lines. Such vertical lines represent a OR operation between the connected inputs. Below is an example of an OR vertical line used in a FBD diagram:

Ladder Diagram (LD)

A Ladder Diagram is a list of rungs. Each rung represents a boolean data flow from a power rail on the left to a power rail on the right. The left power rail represents the **TRUE** state. The data flow must be understood from the left to the right. Each symbol connected to the rung either changes the rung state or performs an operation. Below are possible graphic items to be entered in LD diagrams:

Power Rails
Contacts and Coils

Operations, Functions and Function blocks, represented by rectangular blocks
Labels and Jumps
Use of ST instructions in graphic languages

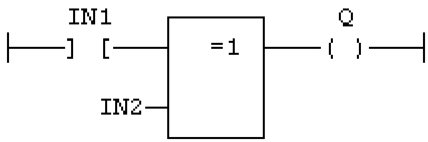
USE OF THE EN INPUT AND THE ENO OUTPUT FOR BLOCKS

The rung state in a LD diagram is always boolean. Blocks are connected to the rung with their first input and output. This implies that special EN and ENO input and output are added to the block if its first input or output is not boolean.

The EN input is a condition. It means that the operation represented by the block is not performed if the rung state (EN) is **FALSE**. The ENO output always represents the sane status as the EN input: the rung state is not modified by a block having an ENO output.

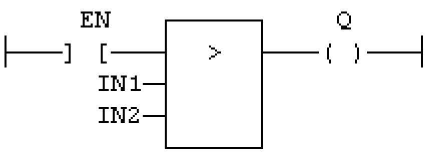
Below is the example of the XOR block, having boolean inputs and outputs, and requiring no EN or ENO pin:

First input is the rung. The rung ist the output.



Below is the example of the > (greater than) block, having non boolean inputs and a boolean output. This block has an EN input in LD language:

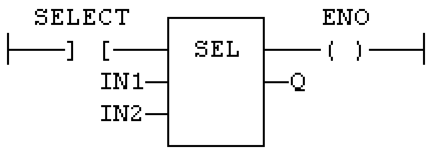
The comparison is executed only if **EN** is **TRUE**.



Below is the example of the SEL function, having a first boolean input, but an integer output. This block has an ENO output in LD language:

The input rung is the selector.

ENO has the same value as **SELECT**.



Finally, below is the example of an addition, having only numerical arguments. This block has both EN and ENO pins in LD language:

The addition is executed only if **EN** is **TRUE**.

ENO is equal to **EN**.

Contacts

Contacts are basic graphic elements of the LD language. A contact is associated to a boolean variable written upon its graphic symbol. A contact sets the state of the rung on its right side, according to the value of the associated variable and the rung state on its left side.

Below are the possible contact symbols and how they change the rung state:

Symbol	Action	Description
BoolVariable ——] [——	Normal	The rung state on the right is the boolean AND between the rung state on the left and the associated variable.
BoolVariable ——]/[——	Negated	The rung state on the right is the boolean AND between the rung state on the left and the negation of the associated variable.
BoolVariable ——]P[——	Positive pulse	The rung state on the right is TRUE only when the rung state on the left is TRUE and the associated variable changes from FALSE to TRUE (rising edge).
BoolVariable ——]N[——	Negative pulse	The rung state on the right is TRUE only when the rung state on the left is TRUE and the associated variable changes from TRUE to FALSE (falling edge).

When a contact or a coil is selected, You can press the SPACE bar to change its type (normal, negated, pulse...).

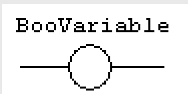
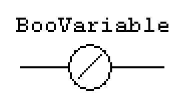
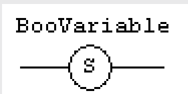
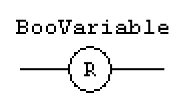
Two serial normal contacts represent an **AND** operation.
Two contacts in parallel represent an **OR** operation.

SEE ALSO
`Coils`
`Power Rails`

Coils

Coils are basic graphic elements of the LD language. A coil is associated to a boolean variable written upon its graphic symbol. A coil performs a change of the associated variable according to the rung state on its left side.

Below are the possible coil symbols and how they change the rung state:

Symbol	State / Action	Description
	Normal	The associated variable is forced to the value of the rung state on the left of the coil.
	Negated	The associated variable is forced to the negation of the rung state on the left of the coil.
	Set	The associated variable is forced to TRUE if the rung state on the left is TRUE. (no action if the rung state is FALSE)
	Reset	The associated variable is forced to FALSE if the rung state on the left is TRUE. (no action if the rung state is FALSE)

When a contact or a coil is selected. You can press the SPACE bar to change its type (normal, negated, pulse...).

EVEN THOUGH COILS ARE COMMONLY CONNECTED TO A POWER RAIL ON THE RIGHT, THE RUNG MAY BE CONTINUED AFTER A COIL. THE RUNG STATE IS NEVER CHANGED BY A COIL SYMBOL.

SEE ALSO
`Contacts`
`Power Rails`

Power Rails

Vertical power rails are used in LD language for representing the limits of a rung.

The power rail on the left represents the **TRUE** value and initiates the rung state. The power rail on the right receives connections from the coils and has no influence on the execution of the program.

Power rails can also be used in FBD language. Only boolean objects can be connected to left and right power rails.

SEE ALSO
Contacts
Coils

Structured Text (ST)

ST is a structured literal programming language. A ST program is a list of statements. Each statement describes an action and must end with a semicolon (“;”).

The presentation of the text has no meaning for a ST program. You can insert blank characters and line breaks where you want in the program text.

COMMENTS
Comment texts can be entered anywhere in a ST program. Comment texts have no meaning for the execution of the program. A comment text must begin with “(*” and end with “*)”. Comments can be entered on several lines (i.e. a comment text may include line breaks). Comment texts cannot be nested.

EXPRESSIONS
Each statement describes an action and may include evaluation of complex expressions. An expression is evaluated:

From the left to the right.

- According to the default priority order of operators.
- The default priority can be changed using parentheses.

Arguments of an expression can be:

- Declared variables
- Constant expressions
- Function calls

STATEMENTS
Below are available basic statements that can be entered in a ST program:

- assignment
- function block calling

Below are the available conditional statements in ST language:

- **IF** / **THEN** / **ELSE** (simple binary switch).

- **CASE** (enumerated switch).

Below are the available statements for describing loops in ST language:

- **WHILE** (with test on loop entry).
- **REPEAT** (with test on loop exit).

FOR (enumeration).

Use of ST expressions in a graphic language

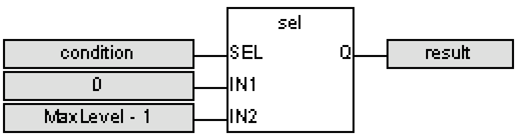
The workbench enables any complex ST expression to be associated with a graphic element in either LD or FBD language. This feature makes possible to simplify LD and FBD diagrams when some trivial calculation has to be entered. It also enables you to use graphic features for representing a main algorithm as text is used for details of implementation.

Expression must be written in ST language. An expression is anything you can imagine between parenthesis in a ST program. Obviously the ST expression must fit the data type required by the diagram (e.g. an expression put on a contact must be boolean).

FBD LANGUAGE

A complex ST expression can be entered in any variable box of a FBD diagram, if the box is not connected on its input.

EXAMPLE



LD LANGUAGE

A complex ST expression can be entered on any kind of contact, and on any input of a function or function block.

Program organization units

An application is a list of programs. Programs are executed sequentially within the target cycle, according to the following model:

```
Begin cycle
| exchange I/Os
| execute first program
| ...
| execute last program
| wait for cycle time to elapse
End Cycle
```

Programs are executed according to the order defined by the user. All SFC programs must be grouped (it is not possible to insert a program in FBD, LD or ST in between two SFC programs). The number of programs in an application is limited to 32767. Each program is entered using a language chosen when the program is created.

Possible languages are:

- Sequential Function Chart (**SFC**),
- Function Block Diagram (**FBD**),
- Ladder Diagram (**LD**),
- Structured Text (**ST**)

Programs must have unique names. The name cannot be a reserved keyword of the programming languages and cannot have the same name as a standard or "C" function or function block. A program should not have the same name as a declared variable. The name of a program should begin by a letter or an underscore ("_") mark, followed by letters, digits or underscore marks. It is not allowed to put two consecutive underscores within a name. Naming is case insensitive. Two names with different cases are considered as the same.

CHILD SFC PROGRAMS

You can define a hierarchy of SFC programs, entered as a tree in the list of programs. A child program is controlled within action blocks of the parent SFC program.

USER DEFINED FUNCTION BLOCKS

The list of programs may be completed by User Defined Function Blocks (UDFBs). UDFBs are described using SFC, FBD, LD or ST language, and can be used as other function blocks in the programs of the application. Input and output parameters plus private variables of a **UDFB** are declared in the variable editor as local variables of the **UDFB**.

There is no restriction using any operation in a **UDFB**. A **UDFB** can call standard functions and function blocks.

A **UDFB** can call another **UDFB**. The called **UDFB** must be declared **before** the calling one in the program list.

Each time a **UDFB** is instantiated, its private variables are duplicated for the declared instance. The code of the **UDFB** is duplicated on each call in parent programs. This leads to higher performances at run time, but consumes code space. It is advised recommended to package small algorithms in UDFBs. Large parts of code should be managed in programs.

A **UDFB** cannot have more than 32 input parameters or 32 output parameters.

SUB-PROGRAMS

The list of programs may be completed by Sub-programs. Sub-programs are described using FBD, LD, ST or IL language, and can be called by the programs of the application. Input and output parameters plus local variables of a sub-program are declared in the variable editor as local variables of the sub-program.

A sub-program may call another sub-program or a **UDFB**.

Unlike **UDFB**, local variables of a sub program are not instantiated. This means that the sub-program always work on the same set of local variables. Local variables of a sub-program keep their value among various calls. The code of a sub-program is not duplicated when called several times by parent programs.

A sub-program cannot have more than 32 input parameters or 32 output parameters.

Data types

BASIC DATA TYPES

Type	Description
BOOL	Boolean (bit) - can be FALSE or TRUE - stored on 1 bit.
SINT (*)	Small signed integer on 8 bits (from -128 to +127).
USINT (*)	Small unsigned integer on 8 bits (from 0 to +255).
BYTE	Same as USINT.
INT (*)	Signed integer on 16 bits (from -32768 to +32767).
UINT (*)	Unsigned integer on 16 bits (from 0 to +65535).
WORD	Same as UINT.
DINT	Signed integer on 32 bits (from -2147483648 to +2147483647).
UDINT (*)	Unsigned integer on 32 bits (from 0 to +4294967295).
DWORD	Same as UDINT.
LINT (*)	Long signed integer on 64 bits.
REAL (*)	Single precision floating point - stored on 32 bits.
LREAL (*)	Double precision floating point - stored on 64 bits.
TIME	Time of day - less than 24h - accuracy is 1ms.
STRING (*)	Variable length string with declared maximum length. The declared maximum length cannot exceed 255 characters.

(*) Some of those data types may be not supported by all targets.

STRUCTURES

A structure is a complex data type defined as a set of members. Members of a structure may have various data types. A member of a structure may have dimensions or may be an instance of another structure.

When a structure is defined, it may be used as other data types to declare variables.

Members of a structure may have an initial value. In that case, corresponding members of all declared variable having this structure type will be initialized with the initial value of the member.

For specifying a member of a structured variable in languages, use the following notation:

```
VariableName.MemberName
```

ENUMERATED DATA TYPES

You can define some new data types that are enumaration of named values. For example:

```
type: LIGHT
values:  GREEN, ORANGE, RED
```

Then in programs, you can use one of the enumerated values, prefixed by the type name:

```
Light1 := LIGHT#RED;
```

Variables having enumerated data types can only be used for assignment, comparison, and SEL/MUX functions.

"BIT FIELD" DATA TYPES

You can define new data types derived from integer data types, that have some readable names for some of their bits. Thus you can use **VarName.BitName** notations in programs. Such data types cannot be derived from the **LINT** type.

Variables

All variables used in programs must be first declared in the variable editor. Each variable belongs to a group and is must be identified by a unique name within its group.

GROUPS

A group is a set of variables. A group either refers to a physical class of variables, or identifies the variables local to a program or user defined function block. Below are the possible groups:

Group	Description
GLOBAL	Internal variables known by all programs.
RETAIN	Non volatile internal variables known by all programs.
%I...	Channels of an input board - variables with same data type linked to a physical input device.
%Q...	Channels of an output board - variables with same data type linked to a physical output device.
PROGRAMxxx	All internal variables local to a program. (the name of the group is the name of the program)
UDFBxxx	All internal variables local to a User Defined Function Block plus its IN and OUT parameters. (the name of the group is the name of the program)

DATA TYPE AND DIMENSION

Each variable must have a valid data type. It can be either a basic data type or a function block. In that case the variable is an instance of the function block. Physical I/Os must have a basic data type. Instances of function blocks can refer either to a standard or "C" embedded block, or to a User Defined Function Block.

If the selected data type is **STRING**, you must specify a maximum length, that cannot exceed 255 characters.

Refer to the list of available data types for more information. Refer to the section describing function blocks for further information about how to use a function instance.

Additionally, you can specify dimension(s) for an internal variable, in order to declare an array. Arrays have at most 3 dimensions. All indexes are 0 based. For instance, in case of single dimension array, the first element is always identified by ArrayName[0]. The total number of items in an array (merging all dimensions) cannot exceed 65535.

NAMING A VARIABLE

A variable must be identified by a unique name within its parent group. The variable name cannot be a reserved keyword of the programming languages and cannot have the same name as a standard or "C" function or function block. A variable should not have the same name as a program or a user defined function block.

The name of a variable should begin by a letter or an underscore ("_") mark, followed by letters, digits or underscore marks. It is not allowed to put two consecutive underscores within a variable name. Naming is case insensitive. Two names with different cases are considered as the same.

NAMING PHYSICAL I/OS

Each I/O channel has a predefined symbol that reflects its physical location. This symbol begins with %I for an input and %Q for an output, followed by a letter identifying the physical size of the data. Then comes the location of the board, expressed on 1 or two numbers, and finally the 0 based index of the channel within the board. All numbers are separated by dots. Below are the possible prefixes for IO symbols:

Prefix	Description
%IX	1 byte input - BOOL or SINT
%QX	1 byte output - BOOL or SINT
%IW	2 bytes input - INT
%QW	2 bytes output - INT
%ID	4 bytes input - DINT or REAL
%QD	4 bytes input - DINT or REAL
%IL	8 bytes input - LINT or LEAL
%QL	8 bytes output - LINT or LEAL
%IS	STRING input
%QS	STRING output

Additionally, you can give an alias (a readable name) to each I/O channel. In that case, either the "%" name or the alias can be used in programs with no difference. The alias must fit to the same rules as a variable name.

ATTRIBUTES OF A VARIABLE

Physical I/Os are marked as either Input or Output. Inputs are read-only variables. For each internal variable, you can select the Read Only.

Parameters of User Defined Function Blocks and sub-programs are marked as either IN or OUT.

PARAMETERS OF SUB-PROGRAMS AND UDFBS

Sub-programs and UDFBs may have parameters on input or ou output. Output parameters cannot be arrays of data structures but only single data. When an array is passed as an inupt parameter to a **UDFB**, it is considered as **INOUT** so the **UDFB** can read or write in it. The support of complex data types for input parameters may depend on selected compiling options.

Arrays

You can specify dimension(s) for internal variables, in order to declare arrays. All indexes are 0 based. For instance, in case of single dimension array, the first element is always identified by ArrayName[0].

To declare an array, enter its dimension in the corresponding column of the variable editor. For a multi-dimension array, enter dimensions separated by comas (ex: 2,10,4).

USE IN ST AND IL LANGUAGES

To specify an item of an array in ST language, enter the mane of the array followed by the index(es) entered between "[" and "]" characters. For multi-dimension arrays, enter indexes separated by comas. Indexes may be either constant or complex expressions.

EXAMPLE

```
TheArray[1,7] := value;

result := SingleArray[i + 2];
```

USE IN FBD AND LD LANGUAGES

In graphical languages, the following blocks are available for managing array elements:

Block	Description
[I]>>	Get value of an item in a single dimension array.
[I,J]>>	Get value of an item in a two dimension array.
[I,J,K]>>	Get value of an item in a three dimension array.
>>[I]	Set value of an item in a single dimention array.
>>[I,J]	Set value of an item in a two dimension array.
>>[I,J,K]	Set value of an item in a three dimension array.

For get blocks, the first input is the array and the output is the value of the item. Other inputs are indexes in the array.

For put blocks, the first input is the forced value and the second input is the array. Other inputs are indexes in the array.

Arrays have at most 3 dimensions.

All indexes are 0 based.

The total number of items in an array (merging all dimensions) cannot exceed 65535.

Constant Expressions

Constant expressions can be used in all languages for assigning a variable with a value. All constant expressions have a well defined data type according to their semantics. If you program an operation between variables and constant expressions having inconsistent data types, it will lead to syntactic errors when the program is compiled. Below are the syntactic rules for constant expressions according to possible data types:

BOOL: BOOLEAN

There are only two possible boolean constant expressions. They are reserved keywords **TRUE** and **FALSE**.

SINT: SMALL (8 BIT) INTEGER

Small integer constant expressions are valid integer values (between -128 and 127) and must be prefixed with **SINT#**. All integer expressions having no prefix are considered as **DINT** integers.

USINT / BYTE: UNSIGNED 8 BIT INTEGER

Unsigned small integer constant expressions are valid integer values (between 0 and 255) and must be prefixed with **USINT#**. All integer expressions having no prefix are considered as **DINT** integers.

INT: 16 BIT INTEGER

16 bit integer constant expressions are valid integer values (between -32768 and 32767) and must be prefixed with **INT#**. All integer expressions having no prefix are considered as **DINT** integers.

UINT / WORD: UNSIGNED 16 BIT INTEGER

Unsigned 16 bit integer constant expressions are valid integer values (between 0 and 255) and must be prefixed with **UINT#**. All integer expressions having no prefix are considered as **DINT** integers.

DINT: 32 BIT (DEFAULT) INTEGER

32 bit integer constant expressions must be valid numbers between -2147483648 to +2147483647. **DINT** is the default size for integers: such constant expressions do not need any prefix. You can use 2#, 8# or 16# prefixes for specifying a number in respectively binary, octal or hexadecimal basis.

UDINT / DWORD: UNSIGNED 32 BIT INTEGER

Unsigned 32 bit integer constant expressions are valid integer values (between 0 and 4294967295) and must be prefixed with **UDINT#**. All integer expressions having no prefix are considered as **DINT** integers.

LINT: LONG (64 BIT) INTEGER

Long integer constant expressions are valid integer values and must be prefixed with **LINT#**. All integer expressions having no prefix are considered as **DINT** integers.

REAL: SINGLE PRECISION FLOATING POINT VALUE

Real constant expressions must be valid number, and must include a dot ("."). If you need to enter a real expression having an integer value, add .0 at the end of the number. You can use F or E separators for specifying the exponent in case of a scientist representation. **REAL** is the default precision for floating points: such expressions do not need any prefix.

LREAL: DOUBLE PRECISION FLOATING POINT VALUE

Real constant expressions must be valid number, and must include a dot ("."), and must be prefixed with **LREAL#**. If you need to enter a real expression having an integer value, add **.0** at the end of the number. You can use **F** or **E** separators for specifying the exponent in case of a scientist representation.

TIME: TIME OF DAY

Time constant expressions represent durations that must be less than 24 hours. Expressions must be prefixed by either **TIME#** or **T#**. They are expressed as a number of hours followed by h, a number of minutes followed by **m**, a number of seconds followed by **s**, and a number of milliseconds followed by **ms**. The order of units (hour, minutes, seconds, milliseconds) must be respected. You cannot insert blank characters in the time expression. There must be at least one valid unit letter in the expression.

STRING: CHARACTER STRING

String expressions must be written between single quote marks. The length of the string cannot exceed 255 characters. You can use the following sequences to represent a special or not printable character within a string:

Sequence	Description
$$	a "$" character
$'	a single quote
$T	a tab stop (ASCII code 9)
$R	a carriage return character (ASCII code 13)
$L	a line feed character (ASCII code 10)
$N	carriage return plus line feed characters (ASCII codes 13 and 10)
$P	a page break character (ASCII code 12)
$xx	any character (xx is the ASCII code expressed on two hexadecimal digits

EXAMPLES OF VALID CONSTANT EXPRESSIONS:

Expression	Description
TRUE	TRUE boolean expression
FALSE	FALSE boolean expression
SINT#127	small integer
INT#2000	16 bit integer
123456	DINT (32 bit) integer
16#abcd	DINT integer in hexadecimal basis
LINT#1	long (64 bit) integer having the value "1"
0.0	0 expressed as a REAL number
1.002E3	1002 expressed as a REAL number in scientist format
LREAL#1E-200	Double precision real number

Expression	Description
TIME#1h34m10s	Time value using TIME#
T#10s123ms	Time value using T#
T#23h59m59s999ms	maximum TIME value
TIME#0s	null TIME value
T#1h123ms	TIME value with some units missing
‘hello’	character string
‘name$Tage’	character string with two words separated by a tab
‘I$’m here’	character string with a quote inside (I’m here)
‘x$00y’	character string with two characters separated by a null character (ASCII code 0)

EXAMPLES OF TYPICAL ERRORS IN CONSTANT EXPRESSIONS:

Expression	Error-Description
BooVar := 1;	0 and 1 cannot be used for Booleans
1a2b	basis prefix (“16#”) omitted
1E-200	“LREAL#” prefix omitted for a double precision float
T#12	Time unit missing
‘I’m here’	quote within a string with “$” mark omitted
hello	quotes omitted around a character string

Conditional Compiling

The compiler supports conditional compiling directives in ST, LD, and FBD languages. Conditional compiling directives condition the inclusion of a part of the program in the generated code. Conditional compiling is an easy way to manage several various configurations and options in a unique application programming.

Conditional compiling uses definitions as conditions. Below is the main syntax:

```
#ifdef CONDITION
    statementsYES...
#else
    statementsNO...
#endif
```

If **CONDITION** has been defined using #define syntax, then the statementsYES part is included in the code, else the statementsNO part is included. The #else statement is optional.

In ST and IL text languages, directives must be entered alone on one line line of text. In FBD language, directives must be entered as the text of network breaks. In LD language, directives must be entered on comment lines.

The condition **__DEBUG** is automatically defined when the application is compiled in **DEBUG** mode. This allows you to incorporate some additional statements (such as trace outputs) in your code that are not included in **RELEASE** mode.

Exception handling

The compiler enables you to write your own exception programs for handling particular system events. The following exceptions can be handled:

- Startup (before the first cycle)
- Shutdown (after the last cycle)
- Division by zero

STARTUP

You can write your own exception program to be executed before the first application cycle is executed:

- Create a new main program that will handle the exception. It cannot be a **SFC** program.
- Add the following global definition:

```
#OnStartup ProgramName
```

THE PROGRAM IS EXECUTED BEFORE ALL OTHER PROGRAMS WITHIN THE FISRT CYCLE. THIS IMPLIES THAT THE CYCLE TIMING MAY BE LONGER DURING THE FIRST CYCLE.

YOU CANNOT PUT BREAKPOINTS IN THE STARTUP PROGRAM.

SHUTDOWN

You can write your own exception program to be executed after the last application cycle when the runtime system is cleanly stopped:

- Create a new main program that will handle the exception. It cannot be a **SFC** program.
- Add the following global definition:

```
#OnShutdown ProgramName
```

YOU CANNOT PUT BREAKPOINTS IN THE SHUTDOWN PROGRAM.

DIVISION BY ZERO

You can write your own exception program for handling the “Division by zero” exception. Below is the procedure you must follow for setting an exception handler:

- Create a new sub-program without any parameter that will handle the exception
- In the editor of global defines (auf Seite 74), insert the following line:

```
#OnDivZero SubProgramName
```

In the sub-program that handles the exception you can perform any safety or trace operation. You then have the selection between the following possibilities:

- Return without any special call. In that case the standard handling will be performed: a system error message is generated, the result of the division is replaced by a maximum value and the application continues.
- Call the FatalStop function. The runtime then stops immediately in Fatal Error mode.
- Call the CycleStop function. The runtime finishes the current program and then turns in cycle setting mode.

Handlers can also be used in **DEBUG** mode for tracking the bad operation. Just put a breakpoint in your handler. When stopped, the call stack will show you the location of the division in the source code of the program.

ARRAY INDEX OUT OF BOUNDS

You can write your own exception program for handling the “Array index out of bounds” exception. Below is the procedure you must follow for setting an exception handler:

- Create a new sub-program without any parameter that will handle the exception
- In the editor of global defines (auf Seite 74), insert the following line:

```
#OnBadArrayIndex SubProgramName
```

This is anyway a fatal error. If the “Check array bounds” compiling option is set, the runtime goes in “fatal error” mode after calling your sub-program.

Variable status bits

The workbench enables you to associate status bits to declared variables. Each variable may have, in addition to its real time value:

- 64 status bits
- a date and time stamp

Status bits and time stamps are generally set by input drivers taking care of hardware inputs, but may also be transported together with the value of the variable on some network protocols. In addition, the IEC 61131-3 programs may access to the status bits of variables.

STATUS BIT MANAGEMENT MAY BE NOT AVAILABLE ON SOME TARGETS. PLEASE REFER TO OEM INSTRUCTIONS FOR

FURTHER DETAILS ABOUT AVAILABLE FEATURES.

STATUS BIT MANAGEMENT IS CPU AND MEMORY CONSUMING AND MAY REDUCE THE PERFORMANCES OF YOUR APPLICATIONS.

ENABLING STATUS BITS

In order to enable the management of status bits and time/date stamps by the runtime, you must check the following option in the list of compiler options from the Project Settings wizard:

- Allocate status flags for variables with embedded properties

Only variables having some properties defined (either a profile attached or embedded symbol) will get status bits. Status bits are available only for global scope variables (global, retain, IOs...) with a single data type (cannot be array or structure).

READING AND WRITING STATUS FROM PROGRAMS

The following functions are available for managing status information in the programs:

Name	Description
vsiGetBit	get a status bit of a variable
vsiGetDate	get the date stamp of a variable
vsiGetTime	get the time stamp of a variable
vsiSetBit	set a status bit of a variable
vsiSetDate	set the date stamp of a variable
vsiSetTime	set the time stamp of a variable
vsiStamp	update the stamp of a variable according to the current time

SYNTAX

```
bBit := vsiGetBit ( variable, bitID );
iDate := vsiGetDate ( variable );
iTime := vsiGetTime ( variable );
bOK := vsiSetBit ( variable, bitID, bBit );
bOK := vsiSetDate ( variable, iDate );
bOK := vsiSetTime ( variable, iTime );
bOK := vsiStamp ( variable );
```

The functions use the following arguments:

Argument	Description
variable	Variable having embedded profile or symbol.
bitID : DINT	ID of a status bit (see list of IDs).
bBit : BOOL	Value of the status bit.

Argument	Description
iDate : DINT	Date stamp according to real time clock functions conventions.
iTime : DINT	Time stamp according to real time clock functions conventions.
bOK : BOOL	TRUE if successful.

See the description of real time clock functions (auf Seite 2-47) for further information about time and date stamps.

DRIVERS SUPPORTING STATUS BITS

Below are runtime drivers taking care of status bits and date/time stamping:

Driver	Description
Variable binding (ETHERNET)	Binding (spontaneous protocol) is used for real time exchange of variable values among runtimes over ETHERNET. The protocol takes care of carrying status bits. The protocol updates the date and time stamps of variables updated by the network.
MODBUS Master	The MODBUS master protocols (RTU / TCP / UDP) takes care of updating the date and time stamp of all variables updated by the network. The MODBUS stack also sets the _VSB_I_BIT status bits of received variables according to the exchange error status.
MODBUS Slave	The MODBUS slave protocols (RTU / TCP / UDP) takes care of updating the date and time stamp of all variables updated by the network.
IEC 60870-5 Slave	The IEC 60870-5-101 and IEC 60870-5-104 slave protocols send the _VSB_I_BIT, _VSB_OV_BIT, _VSB_BL_BIT, _VSB_SP_BIT and _VSB_NT_BIT in the protocol telegrams for points and measures. Update of date/time stamp included.
IEC 61850 Server	Variable status bits are not supported by the IEC 61850 Server.
IEC 61850 Client	The IEC 61850 Client can read the _VSB_I_BIT from the IEC 61850 Server. Update of date/time stamp included.
zenon RT to straton connection	The zenon RT to straton connection can read all 64 status bits. Update of date/ time stamp included.

SEE ALSO

Variable Status Bit List

LIST OF VARIABLE STATUS BITS

Below is the list of available status bits. Identifiers (**_VSB_...**) are predefined in the compiler and can be directly used in the programs:

Bit	Identifier	Description
0	**_VSB_ST_M1**	user defined status
1	**_VSB_ST_M2**	user defined status
2	**_VSB_ST_M3**	user defined status
3	**_VSB_ST_M4**	user defined status
4	**_VSB_ST_M5**	user defined status
5	**_VSB_ST_M6**	user defined status
6	**_VSB_ST_M7**	user defined status
7	**_VSB_ST_M8**	user defined status
8	**_VSB_SELEC**	Select
9	**_VSB_REV**	Revision
10	**_VSB_DIREC**	Desired direction
11	**_VSB_RTE**	Runtime exceeded
12	**_VSB_MVALUE**	Manual value
13	**_VSB_ST_14**	user defined status
14	**_VSB_ST_15**	user defined status
15	**_VSB_ST_16**	user defined status
16	**_VSB_GR**	General request
17	**_VSB_SPONT**	Spontaneous
18	**_VSB_I_BIT**	Invalid
19	**_VSB_SUWI**	Summer/Winter time announcement
20	**_VSB_N_UPD**	Switched off
21	**_VSB_RT_E**	Realtime external
22	**_VSB_RT_I**	Realtime internal
23	**_VSB_NSORT**	Not sortable
24	**_VSB_DM_TR**	Default message trafo value
25	**_VSB_RM_TR**	Run message trafo value

Bit	Identifier	Description
26	_VSB_INFO	Info for variable
27	_VSB_AVALUE	Alternative value
28	_VSB_RES28	reserved
29	_VSB_ACTUAL	Not updated
30	_VSB_WINTER	Winter time
31	_VSB_RES31	reserved
32	_VSB_TCB0	Transmission cause
33	_VSB_TCB1	Transmission cause
34	_VSB_TCB2	Transmission cause
35	_VSB_TCB3	Transmission cause
36	_VSB_TCB4	Transmission cause
37	_VSB_TCB5	Transmission cause
38	_VSB_PN_BIT	P/N bit
39	_VSB_T_BIT	Test bit
40	_VSB_WR_ACK	Acknoledge writing
41	_VSB_WR_SUC	Writing successful
42	_VSB_NORM	Normal status
43	_VSB_ABNORM	Deviation normal status
44	_VSB_BL_BIT	IEC status: blocked
45	_VSB_SP_BIT	IEC status: substituted
46	_VSB_NT_BIT	IEC status: not typical
47	_VSB_OV_BIT	IEC status: overflow
48	_VSB_SE_BIT	IEC status: select
49	not defined	
50	not defined	
51	not defined	
52	not defined	
53	not defined	
54	not defined	
55	not defined	
56	not defined	

Bit	Identifier	Description
57	**not defined**	
58	**not defined**	
59	**not defined**	
60	**not defined**	
61	**not defined**	
62	**not defined**	
63	**not defined**	

Basic Operations

LANGUAGE FEATURES - BASIC DATA MANIPULATION

Variable assignment

Bit access

Parentheses

Calling a function

Calling a function block

Calling a sub-program

BASIC DATA MANIPULATION FUNCTIONS

Name	Description
MOVEBLOCK	Copying/moving array items
COUNTOF	Number of items in an array
INC	Increase a variable
DEC	decrease a variable

LANGUAGE FEATURES - CONTROLLING PROGRAM EXECUTION

Labels

Jumps

RETURN

STRUCTURED STATEMENTS- CONTROLLING PROGRAM EXECUTION

Statement	Description
IF	Conditional execution of statements.
WHILE	Repeat statements while a condition is TRUE.
REPEAT	Repeat statements until a condition is TRUE.
FOR	Execute iterations of statements.
CASE	Switch to one of various possible statements.
EXIT	Exit from a loop instruction.
WAIT	Delay program execution.
ON	Conditional execution.

Access to bits of an integer

You can directly specify a bit within n integer variable in expressions and diagrams, using the following notation:

`Variable.BitNo`

Where:

Variable	is the name of an integer variable.
BitNo	is the number of the bit in the integer.

The variable can have one of the following data types:

SINT, **USINT**, **BYTE** (8 bits from .0 to .7)
INT, **UINT**, **WORD** (16 bits from .0 to .15)
DINT, **UDINT**, **DWORD** (32 bits from .0 to 31)
LINT (from 0 to 63)

BitNo = 0 always represents the less significant bit.

Calling a function

A function calculates a result according to the current value of its inputs. Unlike a function block, a function has no internal data and is not linked to declared instances. A function has only one output: the result of the function. A function can be:

- Astandard function (**SHL**, **SIN**...).
- A function written in "C" language and embedded on the target.

ST LANGUAGE

To call a function block in ST, you have to enter its name, followed by the input parameters written between parenthesis and separated by comas. The function call may be inserted into any complex expression. a function call can be used as an input parameter of another function. The following example demonstrates a call to ODD and SEL functions:

EXAMPLE

```
(* The following statement converts any odd integer value into the nearest even integer: *)

    iEvenVal := SEL ( ODD( iValue ), iValue, iValue+1 );
```

FBD AND LD LANGUAGES

To call a function block in FBD or LD languages, you just need to insert the function in the diagram and to connect its inputs and output.

IL LANGUAGE

To call a function block in IL language, you must load its first input parameter before the call, and then use the function name as an instruction, followed by the other input parameters, separated by comas. The result of the function is then the current result. The following example demonstrates a call to ODD and SEL functions:

EXAMPLE

```
(* The following statement converts any odd integer into 0: *)

Op1: LD    iValue
     ODD
     SEL   iValue, 0
     ST    iResult
```

Calling a function block

CAL CALC CALNC CALCN

A function block groups an algorithm and a set of private data. It has inputs and outputs. A function block can be:

1. A standard function block (**RS**, **TON**...).
2. A block written in "C" language and embedded on the target.
3. A User Defined Function Block (**UDFB**) written in **ST**, **FBD**, **LD** or **IL**.

To use a function block, you have to declare an instance of the block as a variable, identified by a unique name. Each instance of a function block as its own set of private data and can be called separately. A call to a function block instance processes the block algorithm on the private data of the instance, using the specified input parameters.

ST LANGUAGE

To call a function block in ST, you have to specify the name of the instance, followed by the input parameters written between parenthesis and separated by comas. To have access to an output parameter, use the name of the instance followed by a dot '.' and the name of the wished parameter. The following example demonstrates a call to an instance of TON function block (MyTimer is declared as an instance of TON):

EXAMPLE

```
MyTimer (bTrig, t#2s);
TimerOutput := MyTimer.Q;
ElapsedTime := MyTimer.ET;
```

FBD AND LD LANGUAGES

To call a function block in FBD or LD languages, you just need to insert the block in the diagram and to connect its inputs and outputs. The name of the instance must be specified upon the rectangle of the block.

IL LANGUAGE

To call a function block in IL language, you must use the CAL instruction, and use a declared instance of the function block. The instance name is the operand of the CAL instruction, followed by the input parameters written between parenthesis and separated by comas. Alternatively the **CALC**, **CALCN** or **CALNC** conditional instructions can be used:

Name	Description
CAL	Calls the function block.
CALC	Calls the function block if the current result is TRUE.
CALNC	Calls the function block if the current result is FALSE.
CALCN	same as CALNC.

The following example demonstrates a call to an instance of TON function block (MyTimer is declared as an instance of TON):

EXAMPLE

```
Op1: CAL   MyTimer (bTrig, t#2s)
     LD    MyTimer.Q
     ST    TimerOutput
     LD    MyTimer.ET
     ST    ElapsedTimer

Op2: LD    bCond
     CALC  MyTimer (bTrig, t#2s)  (* called only if bCond is TRUE *)
Op3: LD    bCond
     CALNC MyTimer (bTrig, t#2s)  (* called only if bCond is FALSE *)
```

Calling a sub-program

A sub-program is called by another program. Unlike function blocks, local variables of a sub-program are not instantiated, and thus you do not need to declare instances. A call to a sub-program processes the block algorithm using the specified input parameters. Output parameters can then be accessed.

ST LANGUAGE

To call a sub-program in ST, you have to specify its name, followed by the input parameters written between parenthesis and separated by comas. To have access to an output parameter, use the name of the sub-program followed by a dot '.' and the name of the wished parameter:

```
MySubProg (i1, i2); (* calls the sub-program *)
Res1 := MySubProg.Q1;
Res2 := MySubProg.Q2;
```

Alternatively, if a sub-program has one and only one output parameter, it can be called as a function in ST language:

```
Res := MySubProg (i1, i2);
```

FBD AND LD LANGUAGES

To call a sub-program in FBD or LD languages, you just need to insert the block in the diagram and to connect its inputs and outputs.

IL LANGUAGE

To call a sub-program in IL language, you must use the CAL instruction with the name of the sub-program, followed by the input parameters written between parenthesis and separated by comas. Alternatively the **CALC**, **CALCN** or **CALNC** conditional instructions can be used:

Name	Description
CAL	Calls the sub-program.
CALC	Calls the sub-program if the current result is **TRUE**.
CALNC	Calls the sub-program if the current result is **FALSE**.
CALCN	same as CALNC.

EXAMPLE

```
    Op1: CAL   MySubProg (i1, i2)
         LD    MySubProg.Q1
         ST    Res1
         LD    MySubProg.Q2
         ST    Res2
```

:= Assignment

OPERATOR

Variable assignment.

INPUTS

Name	Type	Description
IN	ANY	Any variable or complex expression

OUTPUTS

Name	Type	Description
Q	ANY	Forced variable

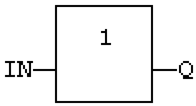
REMARKS

The output variable and the input expression must have the same type. The forced variable cannot have the read only attribute. In LD and FBD languages, the 1 block is available to perform a “1 gain” data copy. In LD language, the input rung (EN) enables the assignment, and the output rung keeps the state of the input rung. In IL language, the LD instruction loads the first operand, and the ST instruction stores the current result into a variable. The current result and the operand of ST must have the same type. Both LD and ST instructions can be modified by N in case of a boolean operand for performing a boolean negation.

ST LANGUAGE

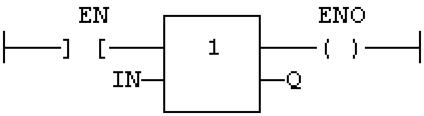
```
Q := IN; (* copy IN into variable Q *)
Q := (IN1 + (IN2 / IN 3)) * IN4; (* assign the result of a complex expression
*)
result := SIN (angle); (* assign a variable with the result of a function *)
time := MyTon.ET; (* assign a variable with an output parameter of a function
block *)
```

FBD LANGUAGE



LD LANGUAGE

The copy is executed only if EN is **TRUE**.
ENO has the same value as EN.



IL LANGUAGE

```
Op1: LD IN (* current result is: IN *)
ST Q (* Q is: IN *)
LDN IN1 (* current result is: NOT (IN1) *)
ST Q (* Q is: NOT (IN1) *)
LD IN2 (* current result is: IN2 *)
STN Q (* Q is: NOT (IN2) *)
```

SEE ALSO

Parentheses

CASE OF ELSE END_CASE

STATEMENT

Switch between enumerated statements.

SYNTAX

```
CASE <DINT expression> OF
<value> :
    <statements>
<value> , <value> :
```

```
        <statements>;
    <value> .. <value> :
        <statements>;
    ELSE
        <statements>
    END_CASE;
```

REMARKS

All enumerated values correspond to the evaluation of the **DINT** expression and are possible cases in the execution of the statements. The statements specified after the **ELSE** keyword are executed if the expression takes a value that is not enumerated in the switch. For each case, you must specify either a value, or a list of possible values separated by comas (“,”) or a range of values specified by a “min .. max” interval. You must enter space characters before and after the “..” separator.

ST LANGUAGE

EXAMPLE

This example checks the first prime numbers:

```
CASE iNumber OF
0 :
   Alarm := TRUE;
   AlarmText := ‘0 gives no result’;
1 .. 3, 5 :
   bPrime := TRUE;
4, 6 :
   bPrime := FALSE;
ELSE
   Alarm := TRUE;
   AlarmText := ‘I don’t know after 6 !’;
END_CASE;
```

FBD LANGUAGE

Not available.

LD LANGUAGE

Not available.

IL LANGUAGE

Not available.

SEE ALSO

IF
WHILE
REPEAT
FOR
EXIT

CountOf

FUNCTION
Returns the number of items in an array.

INPUTS

Name	Type	Description
ARR	ANY	Declared array.

OUTPUTS

Name	Type	Description
Q	DINT	Total number of items in the array.

REMARKS
The input must be an array and can have any data type. This function is particularly useful to avoid writing directly the actual size of an array in a program, and thus keep the program independent from the declaration.

EXAMPLE

```
FOR i := 1 TO CountOf (MyArray) DO
   MyArray[i-1] := 0;
END_FOR;
```

In LD language, the operation is executed only if the input rung (EN) is **TRUE**. The output rung (ENO) keeps the same value as the input rung.

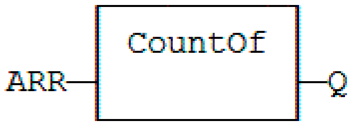
EXAMPLE

Array	Return
Arr1 [0..9]	10
Arr2 [0..4 , 0..9]	50

ST LANGUAGE

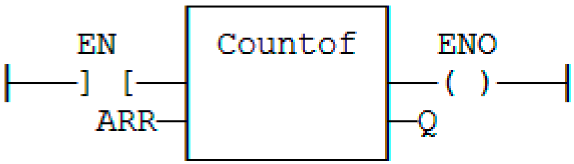
```
Q := CountOf (ARR);
```

FBD LANGUAGE



LD LANGUAGE

The function is executed only if EN is **TRUE**.
ENO keeps the same value as EN.



IL LANGUAGE

Not available.

DEC

FUNCTION

Decrease a numerical variable.

INPUTS

Name	Type	Description
IN	ANY	Numerical variable (increased after call).

OUTPUTS

Name	Type	Description
Q	ANY	Decreased value.

REMARKS

When the function is called, the variable connected to the IN input is decreased and copied to Q. All data types are supported except **BOOL** and **STRING**: for these types, the output is the copy of IN.

For real values, variable is decreased by 1.0. For time values, variable is decreased by 1ms.

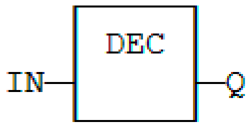
The IN input must be directly connected to a variable, and cannot be a constant or complex expression.

This function is particularly designed for ST language. It allows simplified writing as assigning the result of the function is not mandatory.

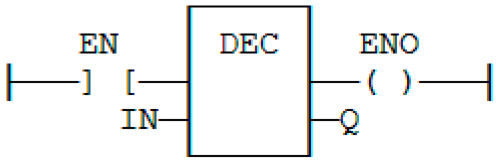
ST LANGUAGE

```
IN := 2;
Q := DEC (IN);
(* now: IN = 1 ; Q = 1 *)

DEC (IN);       (* simplified call *)
```

FBD LANGUAGE



LD LANGUAGE



IL LANGUAGE

Not available.

EXIT

STATEMENT

Exit from a loop statement.

REMARKS

The **EXIT** statement indicates that the current loop (**WHILE**, **REPEAT** or FOR) must be finished. The execution continues after the **END_WHILE**, **END_REPEAT** or **END_FOR** keyword or the loop where the **EXIT** is. **EXIT** quits only one loop and cannot be used to exit at the same time several levels of nested loops.

Loop instructions may lead to infinite loops that block the target cycle.

ST LANGUAGE

This program searches for the first non null item of an array:

```
iFound = -1; (* means: not found *)
FOR iPos := 0 TO (iArrayDim - 1) DO
   IF iPos <> 0 THEN
      iFound := iPos;
      EXIT;
```

```
        END_IF;
    END_FOR;
```

FBD LANGUAGE
Not available.

LD LANGUAGE
Not available.

IL LANGUAGE
Not available.

SEE ALSO
IF
WHILE
REPEAT
FOR
CASE

FOR TO BY END_FOR

STATEMENT
Iteration of statement execution.

SYNTAX
```
FOR <index> := <minimum> TO <maximum> BY <step> DO
    <statements>
END_FOR;
```
Where:

Index	DINT internal variable used as index.
minimum	DINT expression: initial value for index.
maximum	DINT expression: maximum allowed value for index.
step	DINT expression: increasing step of index after each iteration (default is 1).

REMARKS
The BY <step> statement can be omitted. The default value for the step is 1.

ST LANGUAGE
```
    iArrayDim := 10;

    (* resets all items of the array to 0 *)
    FOR iPos := 0 TO (iArrayDim - 1) DO
```

```
      MyArray[iPos] := 0;
   END_FOR;

   (* set all items with odd index to 1 *)
   FOR iPos := 1 TO 9 BY 2 DO
      MyArray[ipos] := 1;
   END_FOR;
```

FBD LANGUAGE
Not available.

LD LANGUAGE
Not available.

IL LANGUAGE
Not available.

SEE ALSO
IF
WHILE
REPEAT
CASE
EXIT

IF THEN ELSE ELSIF END_IF

STATEMENT
Conditional execution of statements.

SYNTAX
```
IF <BOOL expression> THEN
    <statements>
ELSIF <BOOL expression> THEN
    <statements>
ELSE
    <statements>
END_IF;
```

REMARKS
The IF statement is available in ST only. The execution of the statements is conditioned by a boolean expression. **ELSIF** and **ELSE** statements are optional. There can be several **ELSIF** statements.

ST LANGUAGE
```
   (* simple condition *)
   IF bCond THEN
      Q1 := IN1;
```

```
        Q2 := TRUE;
    END_IF;

    (* binary selection *)
    IF bCond THEN
        Q1 := IN1;
        Q2 := TRUE;
    ELSE
        Q1 := IN2;
        Q2 := FALSE;
    END_IF;

    (* enumerated conditions *)
    IF bCond1 THEN
        Q1 := IN1;
    ELSIF bCond2 THEN
        Q1 := IN2;
    ELSIF bCond3 THEN
        Q1 := IN3;
    ELSE
        Q1 := IN4;
    END_IF;
```

FBD LANGUAGE

Not available.

LD LANGUAGE

Not available.

IL LANGUAGE

Not available.

SEE ALSO

WHILE
REPEAT
FOR
CASE
EXIT

INC

FUNCTION

Increase a numerical variable:

INPUTS

Name	Type	Description
IN	ANY	Numerical variable (increased after call).

OUTPUTS

Name	Type	Description
Q	ANY	Increased value.

REMARKS

When the function is called, the variable connected to the IN input is increased and copied to Q. All data types are supported except **BOOL** and **STRING**: for these types, the output is the copy of IN.

For **REAL** values, variable is increased by 1.0. For **TIME** values, variable is increased by 1ms.

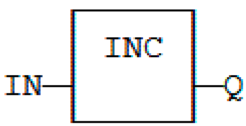
The IN input must be directly connected to a variable, and cannot be a constant or complex expression.

This function is particularly designed for ST language. It allows simplified writing as assigning the result of the function is not mandatory.

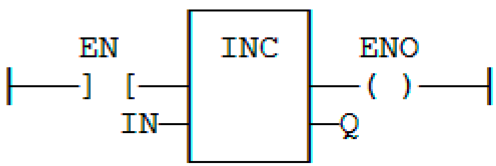
ST LANGUAGE

```
IN := 1;
Q := INC (IN);
(* now: IN = 2 ; Q = 2 *)

INC (IN); (* simplified call *)
```

FBD LANGUAGE



LD LANGUAGE



IL LANGUAGE

Not available.

Jumps JMP JMPC JMPNC JMPCN

STATEMENT

Jump to a label.

REMARKS

A jump to a label branches the execution of the program after the specified label. Labels and jumps cannot be used in structured ST language. In FBD language, a jump is represented by the >> symbol followed by the label name. The input of the >> symbol must be connected to a valid boolean signal. The jump is performed only if the input is **TRUE**. In LD language, the >> symbol, followed by the target label name, is used as a coil at the end of a rung. The jump is performed only if the rung state is **TRUE**. In IL language, JMP, **JMPC**, **JMPCN** and **JMPNC** instructions are used to specify a jump. The destination label is the operand of the jump instruction.

BACKWARD JUMPS MAY LEAD TO INFINITE LOOPS THAT BLOCK THE TARGET CYCLE.

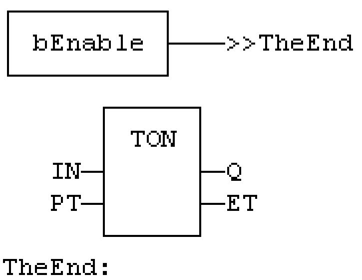
ST LANGUAGE

Not available.

FBD LANGUAGE

In this example the TON block will not be called if bEnable is **TRUE**:

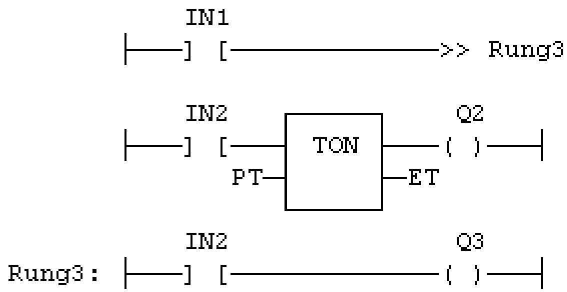
EXAMPLE



LD LANGUAGE

In this example the second rung will be skipped if IN1 is **TRUE**:

EXAMPLE



IL LANGUAGE

JUMP INSTRUCTIONS

Name	Description
JMP	Jump always
JMPC	Jump if the current result is TRUE
JMPNC	Jump if the current result is FALSE
JMPCN	Same as JMPNC

EXAMPLE

```
Start:    LD   IN1
          JMPC TheRest   (* Jump to “TheRest” if IN1 is TRUE *)

          LD   IN2       (* these three instructions are not executed *)
          ST   Q2        (* if IN1 is TRUE *)
          JMP  TheEnd    (* unconditional jump to “TheEnd” *)

TheRest:  LD   IN3
          ST   Q3
TheEnd:
```

SEE ALSO

Labels

RETURN

Labels

STATEMENT

Destination of a Jump instruction.

REMARKS

Labels are used as a destination of a jump instruction in FDB, LD or IL language. Labels and jumps cannot be used in structured ST language. A label must be represented by a unique name, followed by a colon (":"). In FBD language, labels can be inserted anywhere in the diagram, and are connected to nothing. In LD language, a label must identify a rung, and is shown on the left side of the rung. In IL language, labels are destination for JMP, **JMPC**, **JMPCN** and **JMPNC** instructions. They must be written before the instruction at the beginning of the line, and should index the beginning of a valid IL statement: LD (load) instruction, or unconditional instructions such as CAL, JMP or RET. The label can also be written alone on a line before the indexed instruction. In all languages, it is not mandatory that a label be a target of a jump instruction. You can also use label for marking parts of the programs in order to increase its readability.

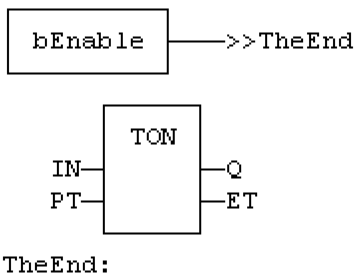
ST LANGUAGE

Not available.

FBD LANGUAGE

In this example the TON block will not be called if bEnable is **TRUE**:

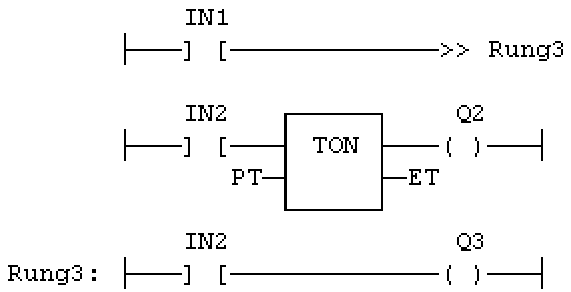
EXAMPLE



LD LANGUAGE

In this example the second rung will be skipped if IN1 is **TRUE**:

EXAMPLE



IL LANGUAGE

```
Start:    LD   IN1      (* unused label - just for readability *)
          JMPC TheRest (* Jump to “TheRest” if IN1 is TRUE *)

          LD   IN2      (* these two instructions are not executed *)
          ST   Q2       (* if IN1 is TRUE *)

TheRest:  LD   IN3      (* label used as the jump destination *)
          ST   Q3
```

SEE ALSO

Jumps

RETURN

MOVEBLOCK

FUNCTION

Move/Copy items of an array.

INPUTS

Name	Type	Description
SRC	ANY (*)	Array containing the source of the copy.
DST	ANY (*)	Array containing the destination of the copy.
PosSRC	DINT	Index of the first character in SRC.
PosDST	DINT	Index of the destination in DST.
NB	DINT	Number of items to be copied.

(*) SRC/DST cannot be a **STRING**.

OUTPUTS

Name	Type	Description
OK	BOOL	TRUE if successful.

REMARKS

Arrays of string are not supported by this function.

In LD language, the operation is executed only if the input rung (EN) is **TRUE**. The function is not available in IL language.

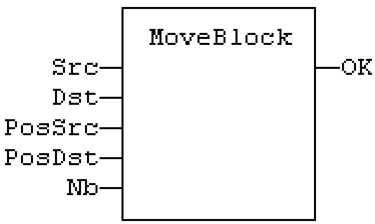
The function copies NB consecutive items starting at the PosSRC index in SRC array to PosDST position in DST array. SRC and DST can be the same array. In that case, the function avoids lost items when source and destination areas overlap.

This function checks array bounds and is always safe. The function returns **TRUE** if successful. It returns **FALSE** if input positions and number do not fit the bounds of SRC and DST arrays.

ST LANGUAGE

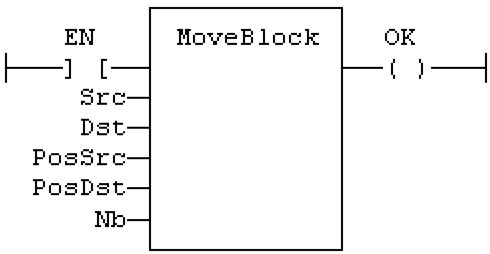
```
OK := MOVEBLOCK (SRC, DST, PosSRS, PosDST, NB);
```

FBD LANGUAGE



LD LANGUAGE

The function is executed only if EN is **TRUE**:



IL LANGUAGE

Not available.

Parentheses ( )

OPERATOR
Force the evaluation order in a complex expression.

REMARKS
Parenthesis are used in ST and IL language for changing the default evaluation order of various operations within a complex expression. For instance, the default evaluation of "2 * 3 + 4" expression in ST language gives a result of 10 as "*" operator has highest priority. Changing the expression as "2 * (3 + 4)" gives a result of 14. Parenthesis can be nested in a complex expression.

Below is the default evaluation order for ST language operations (first is highest priority):

Order	Description	Operators
1	Unary operators	- NOT
2	Multiply/Divide	* /
3	Add/Subtract	+ -
4	Comparisons	< > <= >= = <>
5	Boolean And	& AND
6	Boolean Or	OR
7	Exclusive OR	XOR

In IL language, the default order is the sequence of instructions. Each new instruction modifies the current result sequentially. In IL language, the opening parenthesis "(" is written between the instruction and its operand. The closing parenthesis ")" must be written alone as an instruction without operand.

ST LANGUAGE
```
Q := (IN1 + (IN2 / IN 3)) * IN4;
```

FBD LANGUAGE
Not available.

LD LANGUAGE
Not available.

IL LANGUAGE
```
Op1: LD(   IN1
     ADD(  IN2
     MUL   IN3
     )
     SUB   IN4
     )
     ST    Q    (* Q is: (IN1 + (IN2 * IN3) - IN4) *)
```

SEE ALSO
`Assignment`

REPEAT UNTIL END_REPEAT

STATEMENT
Repeat a list of statements.

SYNTAX
```
REPEAT
    <statements>
UNTIL <BOOL expression> END_REPEAT;
```

REMARKS
The statements between **REPEAT** and **UNTIL** are executed until the boolean expression is **TRUE**. The condition is evaluated after the statements are executed. Statements are executed at least once.

LOOP INSTRUCTIONS MAY LEAD TO INFINITE LOOPS THAT BLOCK THE TARGET CYCLE. NEVER TEST THE STATE OF AN INPUT IN THE CONDITION AS THE INPUT WILL NOT BE REFRESHED BEFORE THE NEXT CYCLE.

ST LANGUAGE
```
iPos := 0;
REPEAT
   MyArray[iPos] := 0;
   iNbCleared := iNbCleared + 1;
   iPos := iPos + 1;
UNTIL iPos = iMax END_REPEAT;
```

FBD LANGUAGE
Not available.

LD LANGUAGE
Not available.

IL LANGUAGE
Not available.

SEE ALSO
`IF`
`WHILE`
`FOR`
`CASE`
`EXIT`

RETURN RET RETC RETNC RETCN

STATEMENT
Jump to the end of the program.

REMARKS
The **RETURN** statement jumps to the end of the program. In FBD language, the return statement is represented by the "<**RETURN**>" symbol. The input of the symbol must be connected to a valid boolean signal. The jump is performed only if the input is **TRUE**. In LD language, the "<**RETURN**>" symbol is used as a coil at the end of a rung. The jump is performed only if the rung state is **TRUE.** In IL language, RET, **RETC**, **RETCN** and **RETNC** instructions are used.

When used within an action block of a SFC step, the **RETURN** statement jumps to the end of the action block.

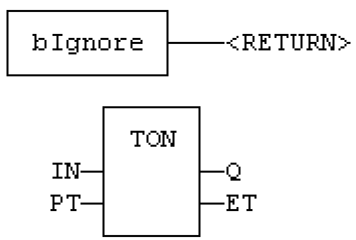
ST LANGUAGE

```
IF NOT bEnable THEN
   RETURN;
END_IF;
```

The rest of the program will not be executed if bEnabled is **FALSE**.

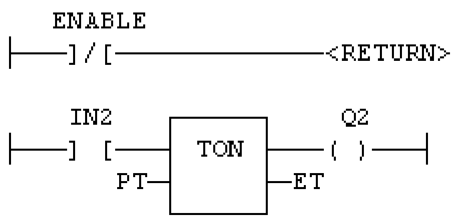
FBD LANGUAGE

EXAMPLE
In this example the TON block will not be called if bIgnore is **TRUE**:



LD LANGUAGE

EXAMPLE
In this example the second rung will be skipped if **ENABLE** is **FALSE**:



IL LANGUAGE

Below is the meaning of possible instructions: Name	Description
RET	Jump to the end always.
RETC	Jump to the end if the current result is TRUE.
RETNC	Jump to the end if the current result is FALSE.
RETCN	Same as RETNC.

EXAMPLE

```
Start: LD   IN1
       RETC            (* Jump to the end if IN1 is TRUE *)

       LD   IN2        (* these instructions are not executed *)
       ST   Q2         (* if IN1 is TRUE *)
       RET             (* Jump to the end unconditionally *)

       LD   IN3        (* these instructions are never executed *)
       ST   Q3
```

SEE ALSO

Labels

Jumps

WHILE DO END_WHILE

STATEMENT

Repeat a list of statements.

SYNTAX

```
WHILE <BOOL expression> DO
    <statements>
END_WHILE ;
```

REMARKS

The statements between DO and **END_WHILE** are executed while the boolean expression is **TRUE**. The condition is evaluated before the statements are executed. If the condition is **FALSE** when **WHILE** is first reached, statements are never executed.

LOOP INSTRUCTIONS MAY LEAD TO INFINITE LOOPS THAT BLOCK THE TARGET CYCLE. NEVER TEST THE STATE OF AN INPUT IN THE CONDITION AS THE INPUT WILL NOT BE REFRESHED BEFORE THE NEXT CYCLE.

ST LANGUAGE

```
    iPos := 0;
    WHILE iPos < iMax DO
       MyArray[iPos] := 0;
       iNbCleared := iNbCleared + 1;
    END_WHILE;
```

FBD LANGUAGE

Not available.

LD LANGUAGE

Not available.

IL LANGUAGE

Not available.

SEE ALSO

IF

REPEAT

FOR

CASE

EXIT

ON

STATEMENT

Conditional execution of statements.

SYNTAX

```
ON <BOOL expression> DO
    <statements>
END_DO;
```

REMARKS

Statements within the ON structure are executed only when the boolean expression rises from **FALSE** to **TRUE**. The ON instruction avoids systematic use of the **R_TRIG** function block or other “last state” flags.

The ON syntax is available in any program, sub-program or **UDFB**. It is available in both T5 p-code or native code compilation modes.

This statement is an extension to the standard and is not IEC61131-3 compliant.

ST LANGUAGE

```
(* This example counts the rising edges of variable bIN *)
ON bIN DO
   diCount := diCount + 1;
END_DO;
```

WAIT / WAIT_TIME

STATEMENT

Suspend the execution of a ST program.

SYNTAX

```
WAIT <BOOL expression> ;

WAIT_TIME <TIME expression> ;
```

REMARKS

The **WAIT** statement checks the attached boolean expression and does the following:

- If the expression is **TRUE**, the program continues normally.

- If the expression is **FALSE**, then the execution of the program is suspended up to the next PLC cycle. The boolean expression will be checked again during next cycles until it becomes **TRUE**. The execution of other programs is not affected.

The **WAIT_TIME** statement suspends the execution of the program for the specified duration. The execution of other programs is not affected.

These instructions are available in ST language only and has no correspondence in other languages. These instructions cannot be called in a User Defined Function Block (**UDFB**). The use of **WAIT** or **WAIT_TIME** in a **UDFB** provokes a compile error.

WAIT and **WAIT_TIME** instructions can be called in a sub-program. However, this may lead to some unsafe situation if the same sub program is called from various programs. Re-entrancy is not supported by **WAIT** and **WAIT_TIME** instructions. Avoiding this situation is the responsibility of the programmer. The compiler outputs some warning messages if a sub-program containing a **WAIT** or **WAIT_TIME** instruction is called from more than one program.

These instructions should not be called from ST parts of SFC programs. This makes no sense as SFC is already a state machine. The use of **WAIT** or **WAIT_TME** in SFC or in a sub-program called from SFC provokes a compile error.

These instructions are not available when the code is compiled through a "C" compiler. Using "C" code generation with a program containing a **WAIT** or **WAIT_TIME** instruction provokes an error during post-compiling.

These statement are extensions to the standard and are not IEC61131-3 compliant.

ST LANGUAGE

```
(* use of WAIT with different kinds of BOOL expressions *)
WAIT BoolVariable;
WAIT (diLevel > 100) AND NOT bAlarm;

WAIT SubProgCall ();


(* use of WAIT_TIME with different kinds of TIME expressions *)
WAIT_TIME t#2s;
WAIT_TIME TimeVariable;
```

Boolean Operations

STANDARD OPERATORS FOR MANAGING BOOLEANS:

Operator	Description
AND	performs a boolean AND
OR	performs a boolean OR
XOR	performs an exclusive OR
NOT	performs a boolean negation of its input
S	force a boolean output to TRUE
R	force a boolean output to FALSE
QOR	Qualified OR

AVAILABLE BLOCKS FOR MANAGING BOOLEAN SIGNALS:

Block	Description
RS	reset dominant bistable
SR	set dominant bistable
R_TRIG	rising pulse detection
F_TRIG	falling pulse detection
SEMA	semaphore
FLIPFLOP	Flipflop/bistable

AND ANDN &

OPERATOR
Performs a logical AND of all inputs.

INPUTS
IN1 : **BOOL** First boolean input.
IN2 : **BOOL** Second boolean input.

OUTPUTS
Q : **BOOL** Boolean AND of all inputs.

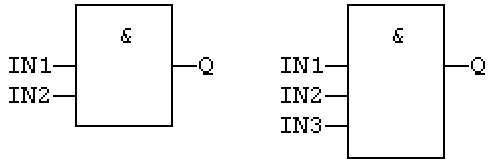
TRUTH TABLE
AND

IN1	IN2	Q
0	0	0
0	1	0
1	0	0
1	1	1

REMARKS
In FBD language, the block may have up to 16 inputs. The block is called “&” in FBD language. In LD language, an AND operation is represented by serialized contacts. In IL language, the AND instruction performs a logical AND between the current result and the operand. The current result must be boolean. The **ANDN** instruction performs an AND between the current result and the boolean negation of the input operand. In ST and IL languages, “&” can be used instead of “AND”.

ST LANGUAGE
```
Q := IN1 AND IN2;
Q := IN1 & IN2 & IN3;
```

FBD LANGUAGE
The block may have up to 16 inputs:



LD LANGUAGE

SERIALIZED CONTACTS:

```
     IN1      IN2       Q
|----] [------] [------( )----|
```

IL LANGUAGE

```
Op1: LD   IN1
     &    IN2   (* "&" or "AND" can be used *)
     ST   Q     (* Q is equal to: IN1 AND IN2 *)
Op2: LD   IN1
     AND  IN2
     &N   IN3   (* "&N" or "ANDN" can be used *)
     ST   Q     (* Q is equal to: IN1 AND IN2 AND (NOT IN3) *)
```

SEE ALSO

OR
XOR
NOT

FLIPFLOP

FUNCTION BLOCK

Flipflop bistable.

INPUTS

Name	Type	Description
IN	BOOL	Swap command (on rising edge).
RST	BOOL	Reset to FALSE.

OUTPUTS

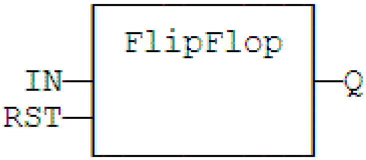
Name	Type	Description
Q	BOOL	Output.

REMARKS
The output is systematically reset to **FALSE** if RST is **TRUE**.
The output changes on each rising edge of the IN input, if RST is **FALSE**.

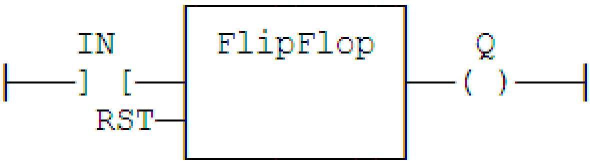
ST LANGUAGE
MyFlipFlop is declared as an instance of **FLIPFLOP** function block:

```
MyFlipFlop (IN, RST);
Q := MyFlipFlop.Q;
```

FBD LANGUAGE



LD LANGUAGE
The IN command is the rung - the rung is the output:



IL LANGUAGE
MyFlipFlop is declared as an instance of **FLIPFLOP** function block:

```
Op1: CAL MyFlipFlop (IN, RST)
     LD  MyFlipFlop.Q
     ST  Q1
```

SEE ALSO
R
S
SR

F_TRIG

FUNCTION BLOCK
Falling pulse detection.

INPUTS

Name	Type	Description
CLK	BOOL	Boolean signal.

OUTPUTS

Name	Type	Description
Q	BOOL	TRUE when the input changes from TRUE to FALSE.

TRUTH TABLE

CLK	CLK prev	Q
0	0	0
0	1	1
1	0	0
1	1	0

REMARKS

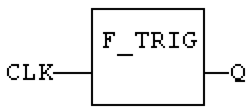
Although]P[an]N[contacts may be used in LD language, it is recommended to use declared instances of **R_TRIG** or **F_TRIG** function blocks in order to avoid unexpected behaviour during an On Line change.

ST LANGUAGE

MyTrigger is declared as an instance of **F_TRIG** function block:

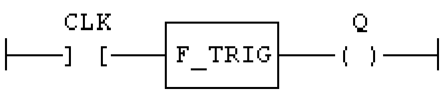
```
MyTrigger (CLK);
Q := MyTrigger.Q;
```

FBD LANGUAGE



LD LANGUAGE

The input signal is the rung - the rung is the output:



IL LANGUAGE

MyTrigger is declared as an instance of **F_TRIG** function block:

```
Op1: CAL MyTrigger (CLK)
     LD  MyTrigger.Q
```

```
        ST  Q
```

SEE ALSO
HYPERLINK "Bool-R_TRIG.docx" R_TRIG

NOT

OPERATOR
Performs a boolean negation of the input.

INPUTS
IN : **BOOL** Boolean value.

OUTPUTS
Q : **BOOL** Boolean negation of the input.

TRUTH TABLE

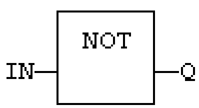
IN	Q
0	1
1	0

REMARKS
In FBD language, the block NOT can be used. Alternatively, you can use a link terminated by a o negation. In LD language, negated contacts and coils can be used. In IL language, the N modifier can be used with instructions LD, AND, OR, XOR and ST. It represents a negation of the operand. In ST language, NOT can be followed by a complex boolean expression between parenthesis.

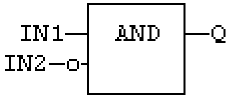
ST LANGUAGE

```
    Q := NOT IN;
    Q := NOT (IN1 OR IN2);
```

FBD LANGUAGE



Explicit use of the NOT block:
Use of a negated link: Q is IN1 AND NOT IN2:



LD LANGUAGE

Negated contact: Q is: IN1 AND NOT IN2:

```
     IN1     IN2      Q
|---] [-----]\[-----( )----|
```

Negated coil: Q is NOT (IN1 AND IN2):

```
     IN1     IN2      Q
|---] [-----] [-----(\)----|
```

IL LANGUAGE

```
Op1: LDN IN1
     OR  IN2
     ST  Q    (* Q is equal to: (NOT IN1) OR IN2 *)
Op2: LD  IN1
     AND IN2
     STN Q    (* Q is equal to: NOT (IN1 AND IN2) *)
```

SEE ALSO

AND

OR

XOR

OR ORN

OPERATOR

Performs a logical OR of all inputs.

INPUTS

Name	Type	Description
IN1	BOOL	First boolean input.
IN2	BOOL	Second boolean input.

OUTPUTS

Name	Type	Description
Q	BOOL	Boolean OR of all inputs.

TRUTH TABLE

IN1	IN2	Q
0	0	0
0	1	1
1	0	1
1	1	1

REMARKS

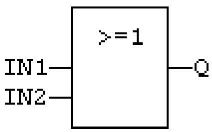
In FBD language, the block may have up to 16 inputs. The block is called >=1 in FBD language. In LD language, an OR operation is represented by contacts in parallel. In IL language, the OR instruction performs a logical OR between the current result and the operand. The current result must be boolean. The ORN instruction performs an OR between the current result and the boolean negation of the operand.

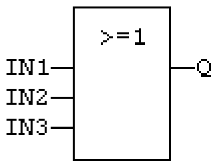
ST LANGUAGE

```
Q := IN1 OR IN2;
Q := IN1 OR IN2 OR IN3;
```

FBD LANGUAGE

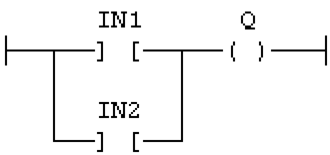
The block may have up to 16 inputs:





LD LANGUAGE

Parallel contacts:



IL LANGUAGE

```
Op1: LD   IN1
     OR   IN2
     ST   Q    (* Q is equal to: IN1 OR IN2 *)
Op2: LD   IN1
     ORN IN2
     ST Q      (* Q is equal to: IN1 OR (NOT IN2) *)
```

SEE ALSO

AND

XOR

NOT

QOR

OPERATOR

Count the number of **TRUE** inputs.

INPUTS

Name	Type	Description
IN1 .. Inn	BOOL	Boolean inputs

OUTPUTS

Name	Type	Description
Q	DINT	Number of inputs being **TRUE**

REMARKS

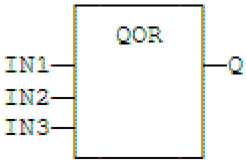
The block accept a non fixed number of inputs.

ST LANGUAGE

```
Q := QOR (IN1, IN2);
Q := QOR (IN1, IN2, IN3, IN4, IN5, IN6);
```

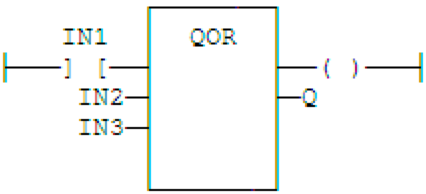
FBD LANGUAGE

The block may have up to 16 inputs:



LD LANGUAGE

The block may have up to 16 inputs:



IL LANGUAGE

```
    Op1: LD  IN1
         QOR IN2, IN3
         ST  Q
```

R

OPERATOR

Force a boolean output to **FALSE**.

INPUTS

Name	Type	Description
RESET	BOOL	Condition.

OUTPUTS

Name	Type	Description
Q	BOOL	Output to be forced.

TRUTH TABLE

RESET	Q prev	Q
0	0	0
0	1	1

RESET	Q prev	Q
1	0	0
1	1	0

REMARKS
S and R operators are available as standard instructions in the IL language. In LD languages they are represented by (S) and (R) coils. In FBD language, you can use (S) and (R) coils, but you should prefer RS and SR function blocks. Set and reset operations are not available in ST language.

ST LANGUAGE
Not available.

FBD LANGUAGE
Not available. Use RS or SR function blocks.

LD LANGUAGE
Use of “R” coil:

```
   RESET    Q
|---] [-----(R)----|
```

IL LANGUAGE

```
    Op1: LD   RESET
         R    Q     (* Q is forced to FALSE if RESET is TRUE *)
                    (* Q is unchanged if RESET is FALSE *)
```

SEE ALSO
S
RS
SR

RS

FUNCTION BLOCK
Reset dominant bistable.

INPUTS

Name	Type	Description
SET	BOOL	Condition for forcing to TRUE.
RESET1	BOOL	Condition for forcing to FALSE (highest priority command).

OUTPUTS

Name	Type	Description
Q1	BOOL	Output to be forced.

TRUTH TABLE

SET	RESET1	Q1 prev	Q1
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

REMARKS

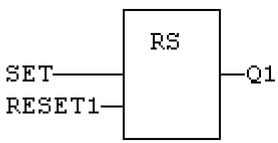
The output is unchanged when both inputs are **FALSE**. When both inputs are **TRUE**, the output is forced to **FALSE** (reset dominant).

ST LANGUAGE

MyRS is declared as an instance of RS function block:

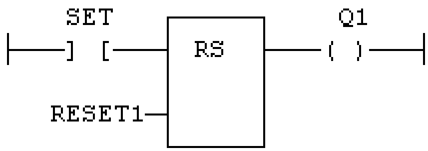
```
MyRS (SET, RESET1);
Q1 := MyRS.Q1;
```

FBD LANGUAGE



LD LANGUAGE

The SET command is the rung - the rung is the output:



IL LANGUAGE

MyRS is declared as an instance of RS function block:

```
Op1: CAL MyRS (SET, RESET1)
     LD  MyRS.Q1
     ST  Q1
```

SEE ALSO

R

S

SR

R_TRIG

FUNCTION BLOCK

Rising pulse detection.

INPUTS

Name	Type	Description
CLK	BOOL	Boolean signal.

OUTPUTS

Name	Type	Description
Q	BOOL	TRUE when the input changes from FALSE to TRUE.

TRUTH TABLE

CLK	CLK prev	Q
0	0	0
0	1	0
1	0	1
1	1	0

REMARKS

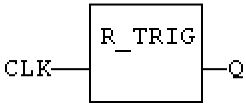
Although]P[an]N[contacts may be used in LD language, it is recommended to use declared instances of **R_TRIG** or **F_TRIG** function blocks in order to avoid unexpected behaviour during an On Line change.

ST LANGUAGE

MyTrigger is declared as an instance of **R_TRIG** function block:

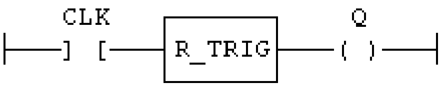
```
MyTrigger (CLK);
Q := MyTrigger.Q;
```

FBD LANGUAGE



LD LANGUAGE

The input signal is the rung - the rung is the output:



IL LANGUAGE

MyTrigger is declared as an instance of **R_TRIG** function block:

```
Op1: CAL MyTrigger (CLK)
     LD  MyTrigger.Q
     ST  Q
```

SEE ALSO

HYPERLINK “Bool-F_TRIG.docx” F_TRIG

S

OPERATOR

Force a boolean output to **TRUE**.

INPUTS

Name	Type	Description
SET	BOOL	Condition.

OUTPUTS

Name	Type	Description
Q	BOOL	Output to be forced.

TRUTH TABLE

SET	Q prev	Q
0	0	0
0	1	1
1	0	1
1	1	1

REMARKS

S and R operators are available as standard instructions in the IL language. In LD languages they are represented by (S) and (R) coils In FBD language, you can use (S) and (R) coils, but you should prefer RS and SR function blocks. Set and reset operations are not available in ST language.

ST LANGUAGE

Not available.

FBD LANGUAGE

Not available. Use RS or SR function blocks.

LD LANGUAGE

Use of S coil:

```
     SET       Q
|----] [-----(S)----|
```

IL LANGUAGE

```
    Op1:  LD   SET
          S    Q     (* Q is forced to TRUE if SET is TRUE *)
                      (* Q is unchanged if SET is FALSE *)
```

SEE ALSO

R
RS
SR

SEMA

FUNCTION BLOCK

Semaphore.

INPUTS

Name	Type	Description
CLAIM	BOOL	Takes the semaphore.
RELEASE	BOOL	Releases the semaphore.

OUTPUTS

Name	Type	Description
BUSY	BOOL	TRUE if semaphore is busy.

REMARKS

The function block implements the following algorithm:

```
BUSY := mem;
if CLAIM then
   mem := TRUE;
else if RELEASE then
   BUSY := FALSE;
   mem := FALSE;
end _ if;
```

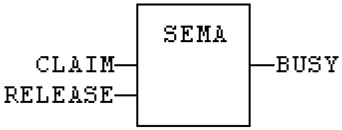
In LD language, the input rung is the **CLAIM** command. The output rung is the **BUSY** output signal.

ST LANGUAGE

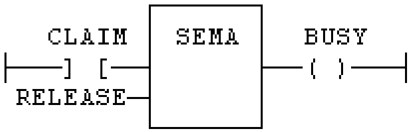
MySema is a declared instance of **SEMA** function block:

```
MySema (CLAIM, RELEASE);
BUSY := MyBlinker.BUSY;
```

FBD LANGUAGE



LD LANGUAGE



IL LANGUAGE

MySema is a declared instance of **SEMA** function block:

```
Op1: CAL MySema (CLAIM, RELEASE)
     LD  MyBlinker.BUSY
     ST  BUSY
```

SR

FUNCTION BLOCK

Set dominant bistable.

INPUTS

Name	Type	Description
SET1	BOOL	Condition for forcing to TRUE (highest priority command).
RESET	BOOL	Condition for forcing to FALSE.

OUTPUTS

Name	Type	Description
Q1	BOOL	Output to be forced.

TRUTH TABLE

SET1	RESET	Q1 prev	Q1
0	0	0	0
0	0	1	1
0	1	0	0

SET1	RESET	Q1 prev	Q1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

REMARKS

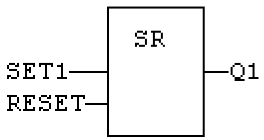
The output is unchanged when both inputs are **FALSE**. When both inputs are **TRUE**, the output is forced to **TRUE** (set dominant).

ST LANGUAGE

MySR is declared as an instance of SR function block:

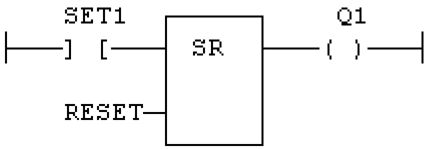
```
MySR (SET1, RESET);
Q1 := MySR.Q1;
```

FBD LANGUAGE



LD LANGUAGE

The SET1 command is the rung - the rung is the output:



IL Language

MySR is declared as an instance of SR function block:

```
Op1: CAL MySR (SET1, RESET)
     LD  MySR.Q1
     ST  Q1
```

SEE ALSO

R

S

RS

XOR XORN

OPERATOR
Performs an exclusive OR of all inputs.

INPUTS

Name	Type	Description
IN1	BOOL	First boolean input.
IN2	BOOL	Second boolean input.

OUTPUTS

Name	Type	Description
Q	BOOL	Exclusive OR of all inputs.

TRUTH TABLE

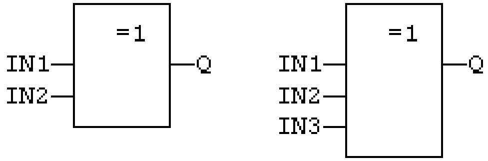
IN1	IN2	Q
0	0	0
0	1	1
1	0	1
1	1	0

REMARKS
The block is called =1 in FBD and LD languages. In IL language, the XOR instruction performs an exclusive OR between the current result and the operand. The current result must be boolean. The **XORN** instruction performs an exclusive between the current result and the boolean negation of the operand.

ST LANGUAGE

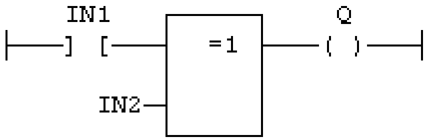
```
Q := IN1 XOR IN2;
Q := IN1 XOR IN2 XOR IN3;
```

FBD LANGUAGE



LD LANGUAGE

First input is the rung. The rung is the output:



IL Language

```
Op1: LD    IN1
     XOR   IN2
     ST    Q    (* Q is equal to: IN1 XOR IN2 *)
Op2: LD    IN1
     XORN  IN2
     ST    Q    (* Q is equal to: IN1 XOR (NOT IN2) *)
```

SEE ALSO

AND
OR
NOT

Arithmetic Operations

STANDARD OPERATORS

Operator	Reference	Description
+	ADD	Addition
-	SUB	Subtraction (dyadic operator)
*****	MUL	Multiplication
/	DIV	Division
-	NEG	Integer negation (monadic operator)

STANDARD FUNCTIONS

Function	Description
MIN	get the minimum of two values
MAX	get the maximum of two values
LIMIT	bound an integer to low and high limits
MOD	modulo
ODD	test if an integer is odd
SetWithin	force a value when inside an interval

+ ADD

OPERATOR
Performs an addition of all inputs.

INPUTS

Name	Type	Description
IN1	ANY	First input.
IN2	ANY	Second input.

OUTPUTS

Name	Type	Description
Q	ANY	Result: IN1 + IN2.

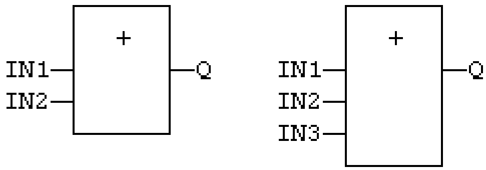
REMARKS
All inputs and the output must have the same type. In FBD language, the block may have up to 16 inputs. In LD language, the input rung (EN) enables the operation, and the output rung keeps the same value as the input rung. In IL language, the ADD instruction performs an addition between the current result and the operand. The current result and the operand must have the same type.

The addition can be used with strings. The result is the concatenation of the input strings.

ST LANGUAGE

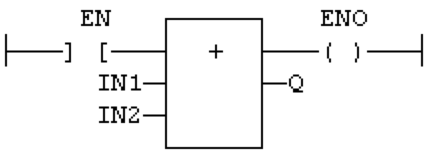
```
Q := IN1 + IN2;
MyString := ‘He’ + ‘ll ‘ + ‘o’;   (* MyString is equal to ‘Hello’ *)
```

FBD LANGUAGE
The block may have up to 16 inputs:



LD LANGUAGE

The addition is executed only if EN is **TRUE**.
ENO is equal to EN.



IL LANGUAGE:

```
Op1: LD   IN1
     ADD IN2
     ST   Q    (* Q is equal to: IN1 + IN2 *)
Op2: LD   IN1
     ADD IN2
     ADD IN3
     ST   Q    (* Q is equal to: IN1 + IN2 + IN3 *)
```

SEE ALSO

- (SUB)
*** (MUL)**
/ (DIV)

/ DIV

OPERATOR

Performs a division of inputs.

INPUTS

Name	Type	Description
IN1	ANY_NUM	First input.
IN2	ANY_NUM	Second input.

OUTPUTS

Name	Type	Description
Q	ANY_NUM	Result: IN1 / IN2.

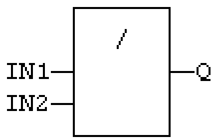
REMARKS

All inputs and the output must have the same type. In LD language, the input rung (EN) enables the operation, and the output rung keeps the same value as the input rung. In IL language, the DIV instruction performs a division between the current result and the operand. The current result and the operand must have the same type.

ST LANGUAGE

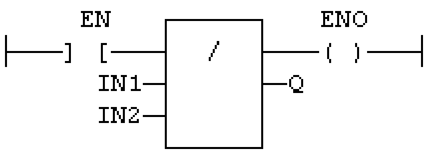
```
Q := IN1 / IN2;
```

FBD LANGUAGE



LD LANGUAGE

The division is executed only if EN is **TRUE**.
ENO is equal to EN.



IL LANGUAGE

```
Op1: LD   IN1
     DIV IN2
     ST   Q    (* Q is equal to: IN1 / IN2 *)
Op2: LD   IN1
     DIV IN2
     DIV IN3
     ST   Q    (* Q is equal to: IN1 / IN2 / IN3 *)
```

SEE ALSO

+ (ADD)
- (SUB)
*** (MUL)**

- NEG

OPERATOR
Performs an integer negation of the input.

INPUTS

Name	Type	Description
IN	DINT	Integer value.

OUTPUTS

Name	Type	Description
Q	DINT	Integer negation of the input.

TRUTH TABLE (EXAMPLES)

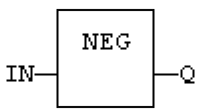
IN	Q
0	0
1	-1
-123	123

REMARKS
In FBD and LD language, the block NEG can be used. In LD language, the operation is executed only if the input rung (EN) is **TRUE**. The output rung (ENO) keeps the same value as the input rung. This feature is not available in IL language. In ST language, “-” can be followed by a complex boolean expression between parenthesis.

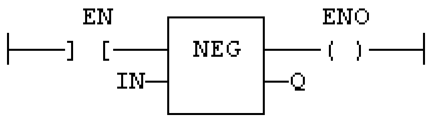
ST LANGUAGE

```
Q := -IN;
Q := - (IN1 + IN2);
```

FBD LANGUAGE



LD LANGUAGE
The negation is executed only if EN is **TRUE**.
ENO keeps the same value as EN.



IL LANGUAGE
Not available.

LIMIT

FUNCTION
Bounds an integer between low and high limits.

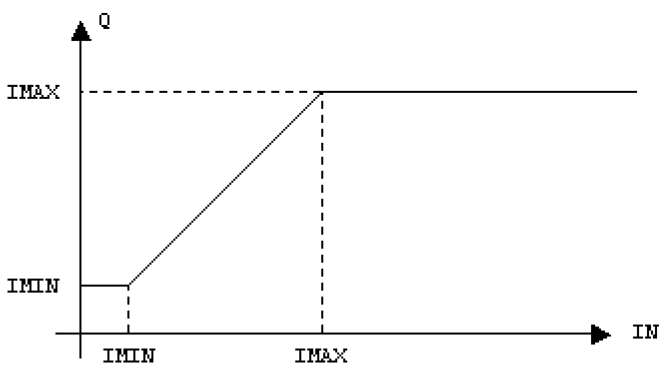
INPUTS

Name	Type	Description
IMIN	DINT	Low bound.
IN	DINT	Input value.
IMAX	DINT	High bound.

OUTPUTS

Name	Type	Description
Q	DINT	IMIN if IN < IMIN; IMAX if IN > IMAX; IN otherwise.

FUNCTION DIAGRAM



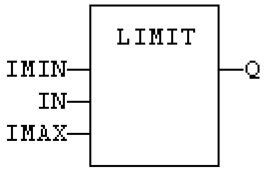
REMARKS
In LD language, the input rung (EN) enables the operation, and the output rung keeps the state of the input

rung. In IL language, the first input must be loaded before the function call. Other inputs are operands of the function, separated by a coma.

ST LANGUAGE

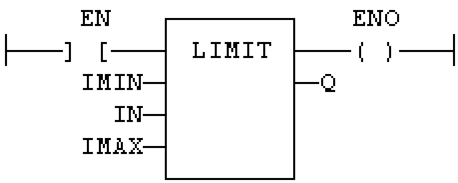
```
Q := LIMIT (IMIN, IN, IMAX);
```

FBD LANGUAGE



LD LANGUAGE

The comparison is executed only if EN is **TRUE**.
ENO has the same value as EN.



IL LANGUAGE

```
Op1: LD    IMIN
     LIMIT IN, IMAX
     ST    Q
```

SEE ALSO

MIN
MAX
MOD
ODD

MAX

FUNCTION

Get the maximum of two values.

INPUTS

Name	Type	Description
IN1	ANY	First input.
IN2	ANY	Second input.

OUTPUTS

Name	Type	Description
Q	ANY	IN1 if IN1 > IN2; IN2 otherwise.

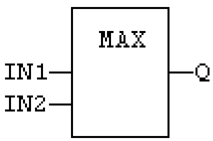
REMARKS

In LD language, the input rung (EN) enables the operation, and the output rung keeps the state of the input rung. In IL language, the first input must be loaded before the function call. The second input is the operand of the function.

ST LANGUAGE

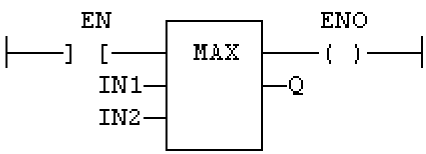
```
Q := MAX (IN1, IN2);
```

FBD LANGUAGE



LD LANGUAGE

The comparison is executed only if EN is TRUE.
ENO has the same value as EN.



IL LANGUAGE

```
Op1: LD  IN1
     MAX IN2
     ST  Q    (* Q is the maximum of IN1 and IN2 *)
```

SEE ALSO

MIN
LIMIT

MOD
ODD

MIN

FUNCTION
Get the minimum of two values.

INPUTS

Name	Type	Description
IN1	ANY	First input.
IN2	ANY	Second input.

OUTPUTS

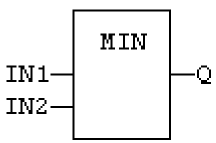
Name	Type	Description
Q	ANY	IN1 if IN1 < IN2; IN2 otherwise.

REMARKS
In LD language, the input rung (EN) enables the operation, and the output rung keeps the state of the input rung. In IL language, the first input must be loaded before the function call. The second input is the operand of the function.

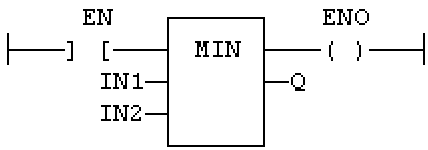
ST LANGUAGE

```
Q := MIN (IN1, IN2);
```

FBD LANGUAGE



LD LANGUAGE
The comparison is executed only if EN is TRUE.
ENO has the same value as EN.



IL LANGUAGE

```
Op1: LD  IN1
     MIN IN2
     ST  Q    (* Q is the minimum of IN1 and IN2 *)
```

SEE ALSO

MAX
LIMIT
MOD
ODD

MOD / MODR / MODLR

FUNCTION

Calculation of modulo.

INPUTS

Name	Type	Description
IN	DINT/REAL/LREAL	Input value.
BASE	DINT/REAL/LREAL	Base of the modulo.

OUTPUTS

Name	Type	Description
Q	DINT/REAL/LREAL	Modulo: rest of the integer division (IN / BASE).

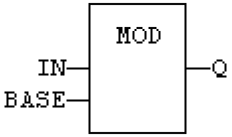
REMARKS

In LD language, the input rung (EN) enables the operation, and the output rung keeps the state of the input rung. In IL language, the first input must be loaded before the function call. The second input is the operand of the function.

ST LANGUAGE

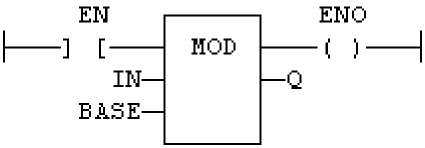
```
Q := MOD (IN, BASE);
```

FBD LANGUAGE



LD LANGUAGE

The comparison is executed only if EN is TRUE.
ENO has the same value as EN.



IL LANGUAGE

```
    Op1: LD   IN
         MOD BASE
         ST   Q     (* Q is the rest of integer division: IN / BASE *)
```

SEE ALSO

MIN
MAX
LIMIT
ODD

* MUL

OPERATOR

Performs a multiplication of all inputs.

INPUTS

Name	Type	Description
IN1	ANY_NUM	First input.
IN2	ANY_NUM	Second input

OUTPUTS

Name	Type	Description
Q	ANY_NUM	Result: IN1 * IN2.

REMARKS

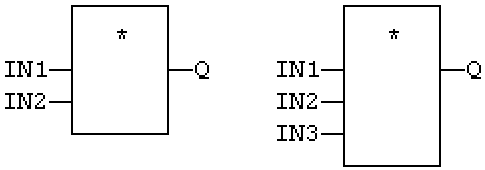
All inputs and the output must have the same type. In FBD language, the block may have up to 16 inputs. In LD language, the input rung (EN) enables the operation, and the output rung keeps the same value as the input rung. In IL language, the MUL instruction performs a multiplication between the current result and the operand. The current result and the operand must have the same type.

ST LANGUAGE

```
Q := IN1 * IN2;
```

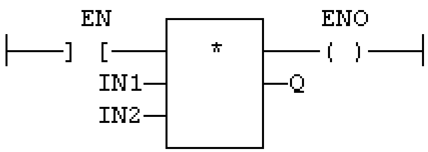
FBD LANGUAGE

The block may have up to 16 inputs:



LD Language

The multiplication is executed only if EN is **TRUE**.
ENO is equal to EN.



IL LANGUAGE

```
Op1: LD  IN1
     MUL IN2
     ST  Q    (* Q is equal to: IN1 * IN2 *)
Op2: LD  IN1
     MUL IN2
     MUL IN3
     ST  Q    (* Q is equal to: IN1 * IN2 * IN3 *)
```

SEE ALSO

+ (ADD)
- (SUB)
/ (DIV)

ODD

FUNCTION
Test if an integer is odd.

INPUTS

Name	Type	Description
IN	DINT	Input value.

OUTPUTS

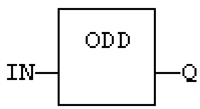
Name	Type	Description
Q	BOOL	TRUE if IN is odd. FALSE if IN is even.

REMARKS
In LD language, the input rung (EN) enables the operation, and the output rung is the result of the function.
In IL language, the input must be loaded before the function call.

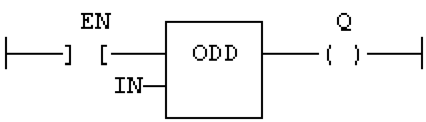
ST LANGUAGE

```
Q := ODD (IN);
```

FBD LANGUAGE



LD LANGUAGE
The function is executed only if EN is TRUE.



IL LANGUAGE

```
Op1: LD  IN
     ODD
     ST  Q    (* Q is TRUE if IN is odd *)
```

SEE ALSO
MIN
MAX

LIMIT
MOD

SetWithin

FUNCTION
Force a value when inside an interval

INPUTS

Name	Type	Description
IN	ANY	Input
MIN	ANY	Low limit of the interval
MAX	ANY	High limit of the interval
VAL	ANY	Value to apply when inside the interval

OUTPUTS

Name	Type	Description
Q	BOOL	Result

TRUTH TABLE

IN	Q
IN < MIN	IN
IN > MAX	IN
MIN < IN < MAX	VAL

REMARKS
The output is forced to VAL when the IN value is within the [MIN .. MAX] interval. It is set to IN when outside the interval.

- SUB

OPERATOR
Performs a subtraction of inputs.

INPUTS

Name	Type	Description
IN1	ANY_NUM / TIME	First input.
IN2	ANY_NUM / TIME	Second input.

OUTPUTS

Name	Type	Description
Q	ANY_NUM / TIME	Result: IN1 - IN2.

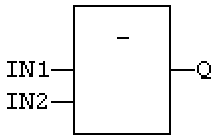
REMARKS

All inputs and the output must have the same type. In LD language, the input rung (EN) enables the operation, and the output rung keeps the same value as the input rung. In IL language, the SUB instruction performs a subtraction between the current result and the operand. The current result and the operand must have the same type.

ST LANGUAGE

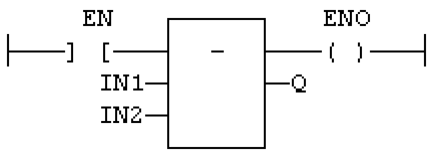
```
Q := IN1 - IN2;
```

FBD LANGUAGE



LD LANGUAGE

The subtraction is executed only if EN is **TRUE**.
ENO is equal to EN.



IL LANGUAGE

```
Op1: LD  IN1
     SUB IN2
     ST  Q    (* Q is equal to: IN1 - IN2 *)
Op2: LD  IN1
     SUB IN2
     SUB IN3
     ST  Q    (* Q is equal to: IN1 - IN2 - IN3 *)
```

SEE ALSO
+ (ADD)
*** (MUL)**
/ (DIV)

Comparison Operations

STANDARD OPERATORS AND BLOCKS THAT PERFORM COMPARISONS:

Operator	Ref	Meaning
<	LT	less than
>	GT	greater than
<=	LE	less or equal
>=	GE	greater or equal
=	EQ	is equal
<>	NE	is not equal
CMP	CMP	Detailed comparison

CMP

FUNCTION BLOCK
Comparison with detailed outputs for integer inputs.

INPUTS

Name	Type	Description
IN1	DINT	First value.
IN2	DINT	Second value.

OUTPUTS

Name	Type	Description
LT	BOOL	TRUE if IN1 < IN2
EQ	BOOL	TRUE if IN1 = IN2
GT	BOOL	TRUE if IN1 > IN2

REMARKS

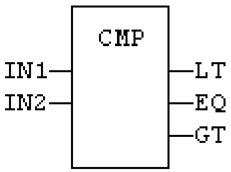
In LD language, the rung input (EN) validates the operation. The rung output is the result of LT (lower than comparison).

ST LANGUAGE

MyCmp is declared as an instance of CMP function block:

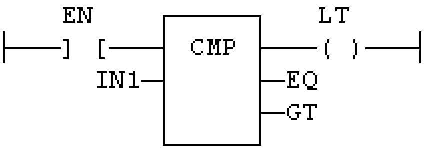
```
MyCMP (IN1, IN2);
bLT := MyCmp.LT;
bEQ := MyCmp.EQ;
bGT := MyCmp.GT;
```

FBD LANGUAGE



LD LANGUAGE

The comparison is performed only if EN is **TRUE**:



IL LANGUAGE

MyCmp is declared as an instance of CMP function block:

```
Op1: CAL MyCmp (IN1, IN2)
     LD  MyCmp.LT
     ST  bLT
     LD  MyCmp.EQ
     ST  bEQ
     LD  MyCmp.GT
     ST  bGT
```

SEE ALSO

> GT
< LT
>= GE
<= LE
= EQ
<> NE

>= GE

OPERATOR
Test if first input is greater than or equal to second input.

INPUTS

Name	Type	Description
IN1	ANY	First input.
IN2	ANY	Second input.

OUTPUTS

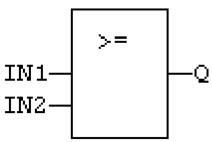
Name	Type	Description
Q	BOOL	TRUE if IN1 >= IN2.

REMARKS
Both inputs must have the same type. In LD language, the input rung (EN) enables the operation, and the output rung is the result of the comparison. In IL language, the GE instruction performs the comparison between the current result and the operand. The current result and the operand must have the same type.

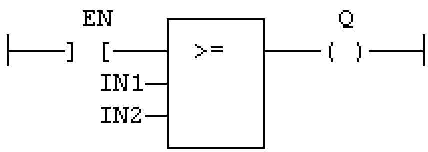
Comparisons can be used with strings. In that case, the lexical order is used for comparing the input strings. For instance, “ABC” is less than “ZX” ; “**ABCD**” is greater than “ABC”.

ST LANGUAGE
```
Q := IN1 >= IN2;
```

FBD LANGUAGE



LD LANGUAGE
The comparison is executed only if EN is **TRUE**.



IL LANGUAGE

```
Op1: LD   IN1
     GE   IN2
     ST   Q     (* Q is true if IN1 >= IN2 *)
```

SEE ALSO

> GT
< LT
<= LE
= EQ
<> NE
CMP

> GT

OPERATOR

Test if first input is greater than second input.

INPUTS

Name	Type	Description
IN1	ANY	First input.
IN2	ANY	Second input.

OUTPUTS

Name	Type	Description
Q	BOOL	TRUE if IN1 > IN2.

REMARKS

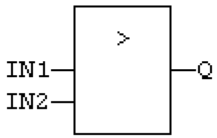
Both inputs must have the same type. In LD language, the input rung (EN) enables the operation, and the output rung is the result of the comparison. In IL language, the GT instruction performs the comparison between the current result and the operand. The current result and the operand must have the same type.

Comparisons can be used with strings. In that case, the lexical order is used for comparing the input strings. For instance, “ABC” is less than “ZX” ; “**ABCD**” is greater than “ABC”.

ST LANGUAGE

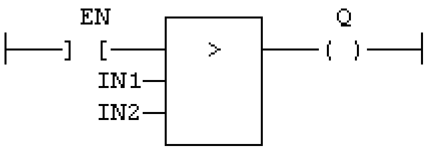
```
Q := IN1 > IN2;
```

FBD LANGUAGE



LD LANGUAGE

The comparison is executed only if EN is **TRUE**.



IL LANGUAGE

```
Op1: LD  IN1
     GT  IN2
     ST  Q    (* Q is true if IN1 > IN2 *)
```

SEE ALSO

< LT
>= GE
<= LE
= EQ
<> NE
CMP

= EQ

OPERATOR

Test if first input is equal to second input.

INPUTS

Name	Type	Description
IN1	ANY	First input.
IN2	ANY	Second input.

OUTPUTS

Name	Type	Description
Q	BOOL	TRUE if IN1 = IN2.

REMARKS

Both inputs must have the same type. In LD language, the input rung (EN) enables the operation, and the output rung is the result of the comparison. In IL language, the EQ instruction performs the comparison between the current result and the operand. The current result and the operand must have the same type.

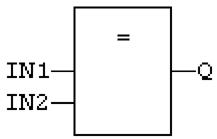
Comparisons can be used with strings. In that case, the lexical order is used for comparing the input strings. For instance, “ABC” is less than “ZX” ; “**ABCD**” is greater than “ABC”.

Equality comparisons cannot be used with **TIME** variables. The reason why is that the timer actually has the resolution of the target cycle and test may be unsafe as some values may never be reached.

ST LANGUAGE

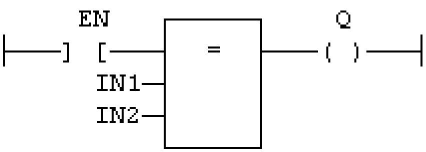
```
Q := IN1 = IN2;
```

FBD LANGUAGE



LD LANGUAGE

The comparison is executed only if EN is **TRUE**:



IL LANGUAGE

```
Op1: LD  IN1
     EQ  IN2
     ST  Q    (* Q is true if IN1 = IN2 *)
```

SEE ALSO

> GT

< LT

>= GE

<= LE

<> NE

CMP

<> NE

OPERATOR

Test if first input is not equal to second input.

INPUTS

Name	Type	Description
IN1	ANY	First input.
IN2	ANY	Second input.

OUTPUTS

Name	Type	Description
Q	BOOL	TRUE if IN1 is not equal to IN2.

REMARKS

Both inputs must have the same type. In LD language, the input rung (EN) enables the operation, and the output rung is the result of the comparison. In IL language, the NE instruction performs the comparison between the current result and the operand. The current result and the operand must have the same type.

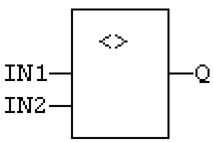
Comparisons can be used with strings. In that case, the lexical order is used for comparing the input strings. For instance, “ABC” is less than “ZX” ; “**ABCD**” is greater than “ABC”.

Equality comparisons cannot be used with **TIME** variables. The reason why is that the timer actually has the resolution of the target cycle and test may be unsafe as some values may never be reached

ST LANGUAGE

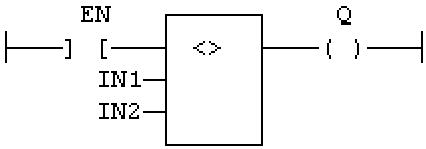
```
Q := IN1 <> IN2;
```

FBD LANGUAGE



LD LANGUAGE

The comparison is executed only if EN is **TRUE**:



IL LANGUAGE

```
Op1: LD   IN1
     NE   IN2
     ST   Q     (* Q is true if IN1 is not equal to IN2 *)
```

SEE ALSO

> GT
< LT
>= GE
<= LE
= EQ
CMP

<= LE

OPERATOR

Test if first input is less than or equal to second input.

INPUTS

Name	Type	Description
IN1	ANY	First input.
IN2	ANY	Second input.

OUTPUTS

Name	Type	Description
Q	BOOL	TRUE if IN1 <= IN2.

REMARKS

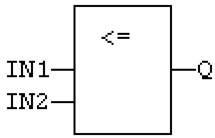
Both inputs must have the same type. In LD language, the input rung (EN) enables the operation, and the output rung is the result of the comparison. In IL language, the LE instruction performs the comparison between the current result and the operand. The current result and the operand must have the same type.

Comparisons can be used with strings. In that case, the lexical order is used for comparing the input strings. For instance, “ABC” is less than “ZX” ; “**ABCD**” is greater than “ABC”.

ST LANGUAGE

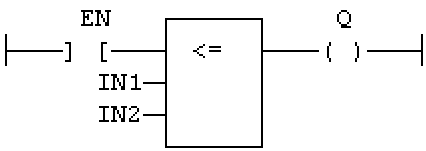
```
Q := IN1 <= IN2;
```

FBD LANGUAGE



LD LANGUAGE

The comparison is executed only if EN is **TRUE**:



IL LANGUAGE

```
Op1: LD  IN1
     LE  IN2
     ST  Q    (* Q is true if IN1 <= IN2 *)
```

SEE ALSO

> GT
< LT
>= GE
= EQ
<> NE
CMP

< LT

OPERATOR

Test if first input is less than second input.

INPUTS

Name	Type	Description
IN1	ANY	First input.
IN2	ANY	Second input.

OUTPUTS

Name	Type	Description
Q	BOOL	TRUE if IN1 < IN2.

REMARKS

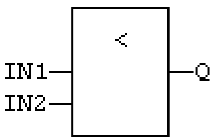
Both inputs must have the same type. In LD language, the input rung (EN) enables the operation, and the output rung is the result of the comparison. In IL language, the LT instruction performs the comparison between the current result and the operand. The current result and the operand must have the same type.

Comparisons can be used with strings. In that case, the lexical order is used for comparing the input strings. For instance, “ABC” is less than “ZX” ; “**ABCD**” is greater than “ABC”.

ST LANGUAGE

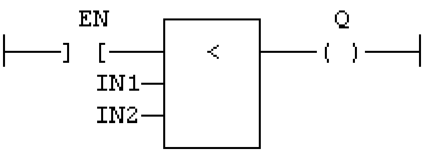
```
Q := IN1 < IN2;
```

FBD LANGUAGE



LD LANGUAGE

The comparison is executed only if EN is **TRUE**:



IL LANGUAGE

```
Op1: LD  IN1
     LT  IN2
     ST  Q    (* Q is true if IN1 < IN2 *)
```

SEE ALSO

> GT

>= GE

<= LE

= EQ

<> NE

CMP

Type Conversion Functions

STANDARD FUNCTIONS FOR CONVERTING A DATA ELEMENT INTO ANOTHER DATA TYPE:

Function	Conversion
ANY_TO_BOOL	converts to boolean
ANY_TO_SINT	converts to small (8 bit) integer
ANY_TO_INT	converts to 16 bit integer
ANY_TO_DINT	converts to integer (32 bit - default)
ANY_TO_LINT	converts to long (64 bit) integer
ANY_TO_REAL	converts to real
ANY_TO_LREAL	converts to double precicion real
ANY_TO_TIME	converts to time
ANY_TO_STRING	converts to character string
NUM_TO_STRING	converts a number to a string

STANDARD FUNCTIONS PERFORMING CONVERSIONS IN BCD FORMAT (*):

Function	Conversion
BIN_TO_BCD	converts a binary value to a DCB value
BCD_TO_BIN	converts a BCD value to a binary value

(*) BCD conversion functions may not be supported by all targets.

ANY_TO_BOOL

OPERATOR
Converts the input into boolean value.

INPUTS

Name	Type	Description
IN	ANY	Input value.

OUTPUTS

Name	Type	Description
Q	BOOL	Value converted to boolean.

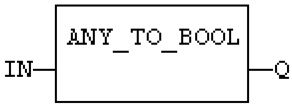
REMARKS

For **DINT**, **REAL** and **TIME** input data types, the result is **FALSE** if the input is 0. The result is **TRUE** in all other cases. For **STRING** inputs, the output is **TRUE** if the input string is not empty, and **FALSE** if the string is empty. In LD language, the conversion is executed only if the input rung (EN) is **TRUE**. The output rung is the result of the conversion. In IL Language, the **ANY_TO_BOOL** function converts the current result.

ST LANGUAGE

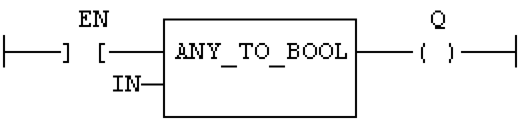
```
Q := ANY_TO_BOOL (IN);
```

FBD LANGUAGE



LD LANGUAGE

The conversion is executed only if EN is **TRUE**.
The output rung is the result of the conversion.
The output rung is **FALSE** if the EN is **FALSE**.



IL LANGUAGE

```
Op1: LD   IN
     ANY_TO_BOOL
     ST   Q
```

SEE ALSO

ANY_TO_SINT
ANY_TO_INT
ANY_TO_DINT
ANY_TO_LINT
ANY_TO_REAL
ANY_TO_LREAL
ANY_TO_TIME
ANY_TO_STRING

ANY_TO_DINT / ANY_TO_UDINT

OPERATOR
Converts the input into integer value.

INPUTS

Name	Type	Description
IN	ANY	Input value.

OUTPUTS

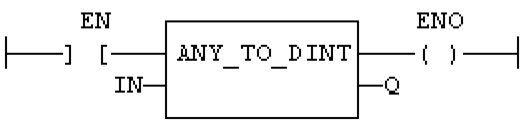
Name	Type	Description
Q	DINT	Value converted to integer.

REMARKS
For **BOOL** input data types, the output is 0 or 1. For **REAL** input data type, the output is the integer part of the input real. For **TIME** input data types, the result is the number of milliseconds. For **STRING** inputs, the output is the number represented by the string, or 0 if the string does not represent a valid number. In LD language, the conversion is executed only if the input rung (EN) is **TRUE**. The output rung (ENO) keeps the same value as the input rung. In IL Language, the **ANY_TO_DINT** function converts the current result.

ST LANGUAGE
```
Q := ANY_TO_DINT (IN);
```

FBD LANGUAGE

LD LANGUAGE
The conversion is executed only if EN is **TRUE**.
ENO keeps the same value as EN.



IL LANGUAGE
```
Op1: LD  IN
     ANY_TO_DINT
     ST  Q
```

SEE ALSO

ANY_TO_BOOL
ANY_TO_SINT
ANY_TO_INT
ANY_TO_LINT
ANY_TO_REAL
ANY_TO_LREAL
ANY_TO_TIME
ANY_TO_STRING

ANY_TO_INT / ANY_TO_UINT

OPERATOR

Converts the input into 16 bit integer value.

INPUTS

Name	Type	Description
IN	ANY	Input value.

OUTPUTS

Name	Type	Description
Q	INT	Value converted to 16 bit integer.

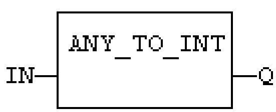
REMARKS

For **BOOL** input data types, the output is 0 or 1. For **REAL** input data type, the output is the integer part of the input real. For **TIME** input data types, the result is the number of milliseconds. For **STRING** inputs, the output is the number represented by the string, or 0 if the string does not represent a valid number. In LD language, the conversion is executed only if the input rung (EN) is **TRUE**. The output rung (ENO) keeps the same value as the input rung. In IL Language, the **ANY_TO_INT** function converts the current result.

ST LANGUAGE

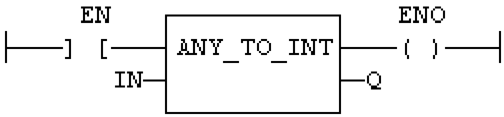
```
Q := ANY_TO_INT (IN);
```

FBD LANGUAGE



LD LANGUAGE

The conversion is executed only if EN is **TRUE**.
ENO keeps the same value as EN.



IL LANGUAGE

```
Op1: LD  IN
     ANY_TO_INT
     ST  Q
```

SEE ALSO

ANY_TO_BOOL
ANY_TO_SINT
ANY_TO_DINT
ANY_TO_LINT
ANY_TO_REAL
ANY_TO_LREAL
ANY_TO_TIME
ANY_TO_STRING

ANY_TO_LINT

OPERATOR

Converts the input into long (64 bit) integer value.

INPUTS

Name	Type	Description
IN	ANY	Input value.

OUTPUTS

Name	Type	Description
Q	LINT	Value converted to long (64 bit) integer.

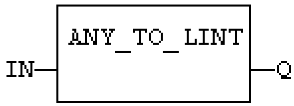
REMARKS

For **BOOL** input data types, the output is 0 or 1. For **REAL** input data type, the output is the integer part of the input real. For **TIME** input data types, the result is the number of milliseconds. For **STRING** inputs, the output is the number represented by the string, or 0 if the string does not represent a valid number. In LD language, the conversion is executed only if the input rung (EN) is **TRUE**. The output rung (ENO) keeps the same value as the input rung. In IL Language, the **ANY_TO_LINT** function converts the current result.

ST LANGUAGE

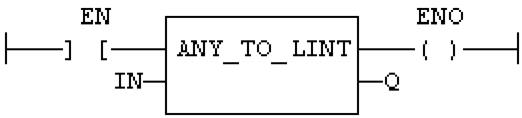
```
Q := ANY_TO_LINT (IN);
```

FBD LANGUAGE



LD LANGUAGE

The conversion is executed only if EN is **TRUE**.
ENO keeps the same value as EN.



IL LANGUAGE

```
Op1: LD   IN
     ANY_TO_LINT
     ST   Q
```

SEE ALSO

ANY_TO_BOOL
ANY_TO_SINT
ANY_TO_INT
ANY_TO_DINT
ANY_TO_REAL
ANY_TO_LREAL
ANY_TO_TIME
ANY_TO_STRING

ANY_TO_LREAL

OPERATOR

Converts the input into double precision real value.

INPUTS

Name	Type	Description
IN	ANY	Input value.

OUTPUTS

Name	Type	Description
Q	LREAL	Value converted to double precision real.

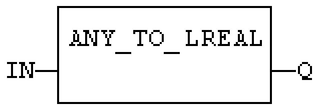
REMARKS

For **BOOL** input data types, the output is 0.0 or 1.0. For **DINT** input data type, the output is the same number. For **TIME** input data types, the result is the number of milliseconds. For **STRING** inputs, the output is the number represented by the string, or 0.0 if the string does not represent a valid number. In LD language, the conversion is executed only if the input rung (EN) is **TRUE**. The output rung (ENO) keeps the same value as the input rung. In IL Language, the **ANY_TO_LREAL** function converts the current result.

ST LANGUAGE

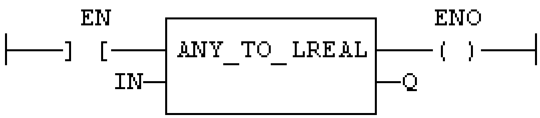
```
Q := ANY_TO_LREAL (IN);
```

FBD LANGUAGE



LD LANGUAGE

The conversion is executed only if EN is **TRUE**.
ENO keeps the same value as EN.



IL LANGUAGE

```
Op1: LD   IN
     ANY_TO_LREAL
     ST   Q
```

SEE ALSO

ANY_TO_BOOL
ANY_TO_SINT
ANY_TO_INT
ANY_TO_DINT
ANY_TO_LINT
ANY_TO_REAL
ANY_TO_TIME
ANY_TO_STRING

ANY_TO_REAL

OPERATOR
Converts the input into real value.

INPUTS

Name	Type	Description
IN	ANY	Input value.

OUTPUTS

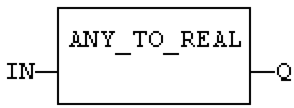
Name	Type	Description
Q	REAL	Value converted to real.

REMARKS
For **BOOL** input data types, the output is 0.0 or 1.0. For **DINT** input data type, the output is the same number. For **TIME** input data types, the result is the number of milliseconds. For **STRING** inputs, the output is the number represented by the string, or 0.0 if the string does not represent a valid number. In LD language, the conversion is executed only if the input rung (EN) is **TRUE**. The output rung (ENO) keeps the same value as the input rung. In IL Language, the **ANY_TO_REAL** function converts the current result.

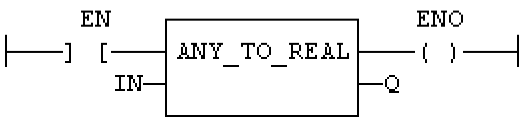
ST LANGUAGE

```
Q := ANY_TO_REAL (IN);
```

FBD LANGUAGE



LD LANGUAGE
The conversion is executed only if EN is **TRUE**.
ENO keeps the same value as EN.



IL LANGUAGE

```
Op1: LD  IN
     ANY_TO_REAL
     ST  Q
```

SEE ALSO

ANY_TO_BOOL
ANY_TO_SINT
ANY_TO_INT
ANY_TO_DINT
ANY_TO_LINT
ANY_TO_LREAL
ANY_TO_TIME
ANY_TO_STRING

ANY_TO_SINT

OPERATOR

Converts the input into a small (8 bit) integer value.

INPUTS

Name	Type	Description
IN	ANY	Input value.

OUTPUTS

Name	Type	Description
Q	SINT	Value converted to a small (8 bit) integer.

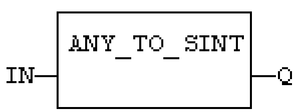
REMARKS

For **BOOL** input data types, the output is 0 or 1. For **REAL** input data type, the output is the integer part of the input real. For **TIME** input data types, the result is the number of milliseconds. For **STRING** inputs, the output is the number represented by the string, or 0 if the string does not represent a valid number. In LD language, the conversion is executed only if the input rung (EN) is **TRUE**. The output rung (ENO) keeps the same value as the input rung. In IL Language, the **ANY_TO_SINT** function converts the current result.

ST LANGUAGE

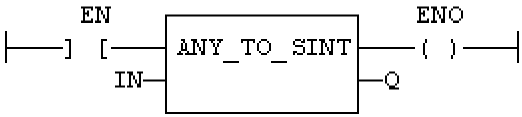
```
Q := ANY_TO_SINT (IN);
```

FBD LANGUAGE



LD LANGUAGE

The conversion is executed only if EN is **TRUE**.
ENO keeps the same value as EN.



IL LANGUAGE

```
Op1: LD   IN
     ANY_TO_SINT
     ST   Q
```

SEE ALSO

ANY_TO_BOOL
ANY_TO_INT
ANY_TO_DINT
ANY_TO_LINT
ANY_TO_REAL
ANY_TO_LREAL
ANY_TO_TIME
ANY_TO_STRING

ANY_TO_STRING

OPERATOR

Converts the input into string value.

INPUTS

Name	Type	Description
IN	ANY	Input value.

OUTPUTS

Name	Type	Description
Q	STRING	Value converted to string.

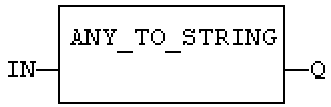
REMARKS

For **BOOL** input data types, the output is 1 or 0 for **TRUE** and **FALSE** respectively. For **DINT**, **REAL** or **TIME** input data types, the output is the string representation of the input number. This is a number of milliseconds for **TIME** inputs. In LD language, the conversion is executed only if the input rung (EN) is **TRUE**. The output rung (ENO) keeps the same value as the input rung. In IL language, the **ANY_TO_STRING** function converts the current result.

ST LANGUAGE

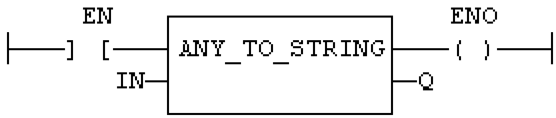
```
Q := ANY_TO_STRING (IN);
```

FBD LANGUAGE



LD LANGUAGE

The conversion is executed only if EN is **TRUE**.
ENO keeps the same value as EN.



IL LANGUAGE

```
    Op1: LD   IN
         ANY_TO_STRING
         ST   Q
```

SEE ALSO

ANY_TO_BOOL
ANY_TO_SINT
ANY_TO_INT
ANY_TO_DINT
ANY_TO_LINT
ANY_TO_REAL
ANY_TO_LREAL
ANY_TO_TIME

ANY_TO_TIME

OPERATOR

Converts the input into time value.

INPUTS

Name	Type	Description
IN	ANY	Input value.

OUTPUTS

Name	Type	Description
Q	TIME	Value converted to time.

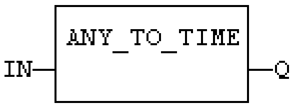
REMARKS

For **BOOL** input data types, the output is t#0ms or t#1ms. For **DINT** or **REAL** input data type, the output is the time represented by the input number as a number of milliseconds. For **STRING** inputs, the output is the time represented by the string, or t#0ms if the string does not represent a valid time. In LD language, the conversion is executed only if the input rung (EN) is **TRUE**. The output rung (ENO) keeps the same value as the input rung. In IL Language, the **ANY_TO_TIME** function converts the current result.

ST LANGUAGE

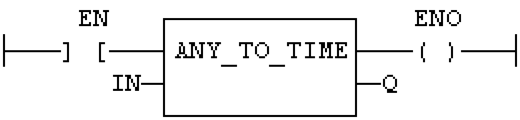
```
Q := ANY_TO_TIME (IN);
```

FBD LANGUAGE



LD LANGUAGE

The conversion is executed only if EN is **TRUE**.
ENO keeps the same value as EN.



IL LANGUAGE

```
Op1: LD   IN
     ANY_TO_TIME
     ST   Q
```

SEE ALSO

ANY_TO_BOOL
ANY_TO_SINT
ANY_TO_INT
ANY_TO_DINT
ANY_TO_LINT
ANY_TO_REAL
ANY_TO_LREAL
ANY_TO_STRING

BCD_TO_BIN

FUNCTION
Converts a BCD (Binary Coded Decimal) value to a binary value.

INPUTS

Name	Type	Description
IN	DINT	Integer value in BCD.

OUTPUTS

Name	Type	Description
Q	DINT	Value converted to integer or 0 if IN is not a valid positive BCD value.

TRUTH TABLE (EXAMPLES)

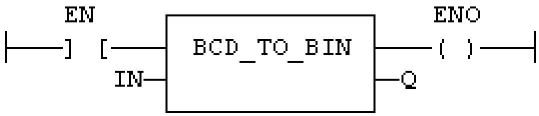
IN	Q
-2	0 (invalid)
0	0
16 (16#10)	10
15 (16#0F)	0 (invalid)

REMARKS
The input must be positive and must represent a valid BCD value. In LD language, the operation is executed only if the input rung (EN) is **TRUE**. The output rung (ENO) keeps the same value as the input rung. In IL, the input must be loaded in the current result before calling the function.

ST LANGUAGE
```
Q := BCD_TO_BIN (IN);
```

FBD LANGUAGE

LD LANGUAGE
The function is executed only if EN is **TRUE**.
ENO keeps the same value as EN.



IL LANGUAGE

```
Op1: LD          IN
     BCD_TO_BIN
     ST          Q
```

SEE ALSO

BIN_TO_BCD

BIN_TO_BCD

FUNCTION

Converts a binary value to a BCD (Binary Coded Decimal) value.

INPUTS

Name	Type	Description
IN	DINT	Integer value

OUTPUTS

Name	Type	Description
Q	DINT	Value converted to BCD or 0 if IN is less than 0

TRUTH TABLE (EXAMPLES)

IN	Q
-2	0 (invalid)
0	0
10	16 (16#10)
22	34 (16#34)

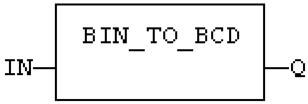
REMARKS

The input must be positive. In LD language, the operation is executed only if the input rung (EN) is **TRUE**. The output rung (ENO) keeps the same value as the input rung. In IL, the input must be loaded in the current result before calling the function.

ST LANGUAGE

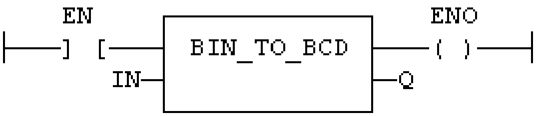
```
Q := BIN_TO_BCD (IN);
```

FBD LANGUAGE



LD LANGUAGE

The function is executed only if EN is **TRUE**.
ENO keeps the same value as EN.



IL LANGUAGE

```
Op1: LD         IN
     BIN_TO_BCD
     ST         Q
```

SEE ALSO

BCD_TO_BIN

NUM_TO_STRING

FUNCTION

Converts a number into string value.

INPUTS

Name	Type	Description
IN	ANY	Input number.
WIDTH	DINT	Wished length for the output string (see remarks)
DIGITS	DINT	Number of digits after decimal point

OUTPUTS

Name	Type	Description
Q	STRING	Value converted to string.

REMARKS

This function converts any numerical value to a string. Unlike the **ANY_TO_STRING** function, it allows you to specify a wished length and a number of digits after the decimal points.

If **WIDTH** is 0, the string is formatted with the necessary length.

If **WIDTH** is greater than 0, the string is completed with heading blank characters in order to match the value of **WIDTH**.

If **WIDTH** is greater than 0, the string is completed with trailing blank characters in order to match the absolute value of **WIDTH**.

If **DIGITS** is 0 then neither decimal part nor point are added.

If **DIGITS** is greater than 0, the corresponding number of decimal digits are added. '0' digits are added if necessary

If the value is too long for the specified width, then the string is filled with '*' characters.

EXAMPLES

```
Q := NUM_TO_STRING (123.4, 8, 2);        (* Q is '  123.40' *)
Q := NUM_TO_STRING (123.4, -8, 2);       (* Q is '123.40  ' *)
Q := NUM_TO_STRING (1.333333, 0, 2);     (* Q is '1.33' *)
Q := NUM_TO_STRING (1234, 3, 0);          (* Q is '***' *)
```

Selectors

STANDARD FUNCTIONS THAT PERFORM DATA SELECTION:

Function	Description
SEL	2 integer inputs
MUX4	4 integer input
MUX8	8 integer input

MUX4

FUNCTION

Select one of the inputs - 4 inputs.

INPUTS

Name	Type	Description
SELECT	DINT	Selection command.
IN1	ANY	First input.
IN2	ANY	Second input.
...		

Name	Type	Description
IN4	ANY	Last input.

OUTPUTS

Name	Type	Description
Q	ANY	IN1 or IN2 ... or IN4 depending on SELECT (see truth table).

TRUTH TABLE

SELECT	Q
0	IN1
1	IN2
2	IN3
3	IN4
other	0

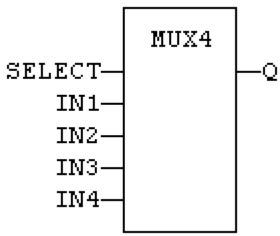
REMARKS

In LD language, the input rung (EN) enables the selection. The output rung keeps the same state as the input rung. In IL language, the first parameter (selector) must be loaded in the current result before calling the function. Other inputs are operands of the function, separated by comas.

ST LANGUAGE

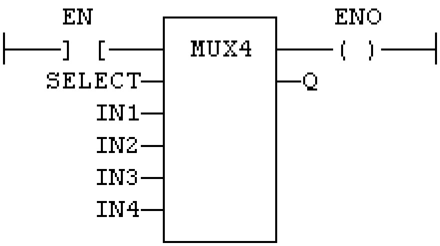
```
Q := MUX4 (SELECT, IN1, IN2, IN3, IN4);
```

FBD LANGUAGE



LD LANGUAGE

The selection is performed only if EN is **TRUE**.
ENO has the same value as EN .



IL LANGUAGE

```
Op1: LD    SELECT
     MUX4  IN1, IN2, IN3, IN4
     ST    Q
```

SEE ALSO

SEL

MUX8

MUX8

FUNCTION

Select one of the inputs - 8 inputs.

INPUTS

Name	Type	Description
SELECT	DINT	Selection command.
IN1	ANY	First input.
IN2	ANY	Second input.
...		
IN8	ANY	Last input.

OUTPUTS

Name	Type	Description
Q	ANY	IN1 or IN2 ... or IN8 depending on SELECT (see truth table).

TRUTH TABLE

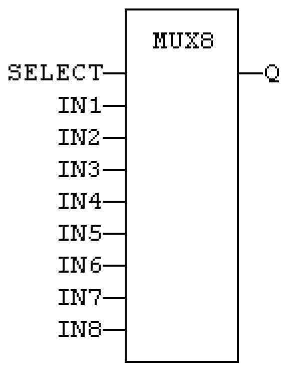
SELECT	Q
0	IN1

SELECT	Q
1	IN2
2	IN3
3	IN4
4	IN5
5	IN6
6	IN7
7	IN8
other	0

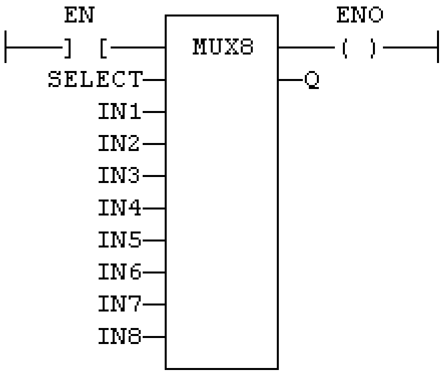
REMARKS
In LD language, the input rung (EN) enables the selection. The output rung keeps the same state as the input rung. In IL language, the first parameter (selector) must be loaded in the current result before calling the function. Other inputs are operands of the function, separated by comas.

ST LANGUAGE
```
Q := MUX8 (SELECT, IN1, IN2, IN3, IN4, IN5, IN6, IN7, IN8);
```

FBD LANGUAGE



LD LANGUAGE
The selection is performed only if EN is **TRUE**.
ENO has the same value as EN.



IL LANGUAGE

```
    Op1: LD    SELECT
         MUX8 IN1, IN2, IN3, IN4, IN5, IN6, IN7, IN8
         ST    Q
```

SEE ALSO

SEL

MUX4

SEL

FUNCTION

Select one of the inputs - 2 inputs.

INPUTS

Name	Type	Description
SELECT	DINT	Selection command.
IN1	ANY	First input.
IN2	ANY	Second input.

OUTPUTS

Name	Type	Description
Q	ANY	IN1 if SELECT is FALSE; IN2 if SELECT is TRUE

TRUTH TABLE

SELECT	Q
0	IN1
1	IN2

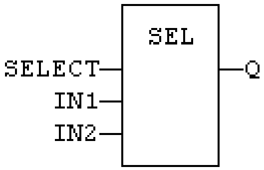
REMARKS

In LD language, the selector command is the input rung. The output rung keeps the same state as the input rung. In IL language, the first parameter (selector) must be loaded in the current result before calling the function. Other inputs are operands of the function, separated by comas.

ST LANGUAGE

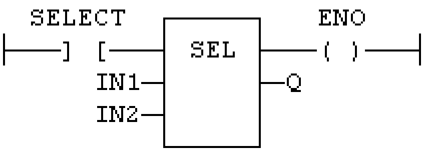
```
Q := SEL (SELECT, IN1, IN2);
```

FBD LANGUAGE



LD LANGUAGE

The input rung is the selector.
ENO has the same value as **SELECT**.



IL LANGUAGE

```
Op1: LD  SELECT
     SEL IN1, IN2
     ST  Q
```

SEE ALSO

MUX4
MUX8

Registers

STANDARD FUNCTIONS FOR MANAGING 8 BIT TO 32 BIT REGISTERS:

Function	Description
SHL	shift left
SHR	shift right
ROL	rotate left
ROR	rotate right

ADVANCED FUNCTIONS FOR REGISTER MANIPULATION:

Function	Description
MBShift	multibyte shift / rotate

BIT TO BIT OPERATIONS ON A 8 BIT TO 32 BIT INTEGERS:

Function	Description
AND_MASK	boolean AND
OR_MASK	boolean OR
XOR_MASK	exclusive OR
NOT_MASK	boolean negation

PACK/UNPACK 8, 16 AND 32 BIT REGISTERS

Function	Description
LOBYTE	Get the lowest byte of a word.
HIBYTE	Get the highest byte of a word.
LOWORD	Get the lowest word of a double word.
HIWORD	Get the highest word of a double word.
MAKEWORD	Pack bytes to a word.
MAKEDWORD	Pack words to a double word.
PACK8	Pack bits in a byte.
UNPACK8	Extract bits from a byte.

BIT ACCESS IN 8 BIT TO 32 BIT INTEGERS:

Function	Description
SETBIT	Set a bit in a register.
TESTBIT	Test a bit of a register.

The following functions are kept for compatibility, but you should use the functions above:

```
AND_DINT, AND_UDINT, AND_DWORD, NOT_DINT, NOT_UDINT, NOT_DWORD
OR_DINT, OR_UDINT, OR_DWORD, XOR_DINT, XOR_UDINT, XOR_DWORD
AND_INT, AND_UINT, AND_WORD, NOT_INT, NOT_UINT, NOT_WORD
OR_INT, OR_UINT, OR_WORD, XOR_INT, XOR_UINT, XOR_WORD
AND_SINT, AND_USINT, AND_BYTE, NOT_SINT, NOT_USINT, NOT_BYTE
OR_SINT, OR_USINT, OR_BYTE, XOR_SINT, XOR_USINT, XOR_BYTE
ROLw, RORw, SHLw, SHRw, ROLb, RORrb, SHLb, SHRb
ROL_DINT, ROR_DINT, SHL_DINT, SHR_DINT
ROL_UDINT, ROR_UDINT, SHL_UDINT, SHR_UDINT
ROL_DWORD, ROR_DWORD, SHL_DWORD, SHR_DWORD
ROL_INT, ROR_INT, SHL_INT, SHR_INT
ROL_UINT, ROR_UINT, SHL_UINT, SHR_UINT
ROL_WORD, ROR_WORD, SHL_WORD, SHR_WORD
ROL_SINT, ROR_SINT, SHL_SINT, SHR_SINT
ROL_USINT, ROR_USINT, SHL_USINT, SHR_USINT
ROL_BYTE, ROR_BYTE, SHL_BYTE, SHR_BYTE
```

AND_MASK

FUNCTION
Performs a bit to bit AND between two integer values

INPUTS

Name	Type	Description
IN	ANY	First input.
MSK	ANY	Second input (AND mask).

OUTPUTS

Name	Type	Description
Q	ANY	AND mask between IN and MSK inputs.

REMARKS
Arguments can be signed or unsigned integers from 8 to 32 bits.

In LD language, the input rung (EN) enables the operation, and the output rung keeps the same value as the input rung. In IL language, the first parameter (IN) must be loaded in the current result before calling the function. The other input is the operands of the function.

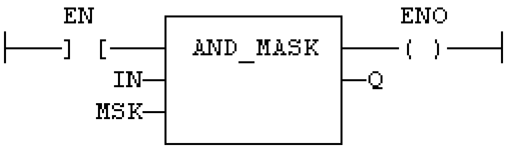
ST LANGUAGE

```
Q := AND_MASK (IN, MSK);
```

FBD LANGUAGE

LD LANGUAGE

The function is executed only if EN is **TRUE**.
ENO is equal to EN.



IL LANGUAGE

```
Op1: LD       IN
     AND_MASK MSK
     ST       Q
```

SEE ALSO

OR_MASK
XOR_MASK
NOT_MASK

HIBYTE

FUNCTION

Get the most significant byte of a word

INPUTS

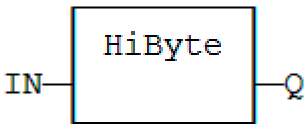
Name	Type	Description
IN	UINT	16 bit register.

OUTPUTS

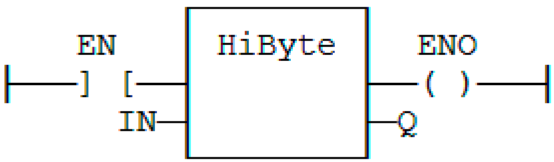
Name	Type	Description
Q	USINT	Most significant byte.

REMARKS
In LD language, the operation is executed only if the input rung (EN) is **TRUE**. The output rung (ENO) keeps the same value as the input rung. In IL, the input must be loaded in the current result before calling the function.

ST LANGUAGE
```
Q := HIBYTE (IN);
```

FBD LANGUAGE


LD LANGUAGE
The function is executed only if EN is **TRUE**.
ENO keeps the same value as EN.



IL LANGUAGE
```
Op1: LD   IN
     HIBYTE
     ST   Q
```

SEE ALSO
LOBYTE
LOWORD
HIWORD
MAKEWORD
MAKEDWORD

LOBYTE

FUNCTION
Get the less significant byte of a word.

INPUTS

Name	Type	Description
IN	UINT	16 bit register.

OUTPUTS

Name	Type	Description
Q	USINT	Lowest significant byte.

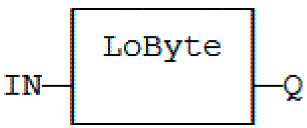
REMARKS

In LD language, the operation is executed only if the input rung (EN) is **TRUE**. The output rung (ENO) keeps the same value as the input rung. In IL, the input must be loaded in the current result before calling the function.

ST LANGUAGE

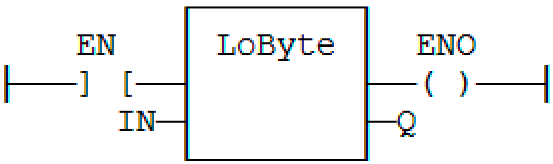
```
Q := LOBYTE (IN);
```

FBD LANGUAGE



LD LANGUAGE

The function is executed only if EN is **TRUE**.
ENO keeps the same value as EN.



IL LANGUAGE

```
Op1: LD   IN
     LOBYTE
     ST   Q
```

SEE ALSO

HIBYTE
LOWORD
HIWORD
MAKEWORD
MAKEDWORD

HIWORD

FUNCTION
Get the most significant word of a double word.

INPUTS

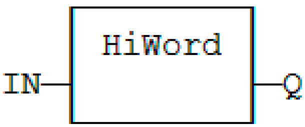
Name	Type	Description
IN	UDINT	32 bit register.

OUTPUTS

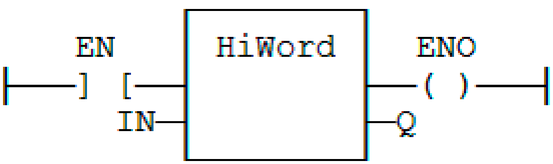
Name	Type	Description
Q	UINT	Most significant word.

REMARKS
In LD language, the operation is executed only if the input rung (EN) is **TRUE**. The output rung (ENO) keeps the same value as the input rung. In IL, the input must be loaded in the current result before calling the function.

ST LANGUAGE
```
Q := HIWORD (IN);
```

FBD LANGUAGE


LD LANGUAGE
The function is executed only if EN is **TRUE**.
ENO keeps the same value as EN.



IL LANGUAGE
```
Op1: LD   IN
     HIWORD
     ST   Q
```

SEE ALSO
LOBYTE
HIBYTE
LOWORD
MAKEWORD
MAKEDWORD

LOWORD

FUNCTION
Get the less significant word of a double word.

INPUTS

Name	Type	Description
IN	UDINT	32 bit register.

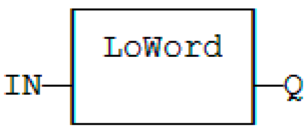
OUTPUTS

Name	Type	Description
Q	UINT	Lowest significant word.

REMARKS
In LD language, the operation is executed only if the input rung (EN) is **TRUE**. The output rung (ENO) keeps the same value as the input rung. In IL, the input must be loaded in the current result before calling the function.

ST LANGUAGE
```
Q := LOWORD (IN);
```

FBD LANGUAGE



LD LANGUAGE
The function is executed only if EN is **TRUE**.
ENO keeps the same value as EN.

IL LANGUAGE
```
Op1: LD  IN
     LOWORD
     ST  Q
```

SEE ALSO
LOBYTE
HIBYTE
HIWORD
MAKEWORD
MAKEDWORD

MAKEDWORD

FUNCTION
Builds a double word as the concatenation of two words.

INPUTS

Name	Type	Description
HI	USINT	Highest significant word.
LO	USINT	Lowest significant word.

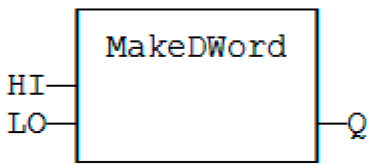
OUTPUTS

Name	Type	Description
Q	UINT	32 bit register.

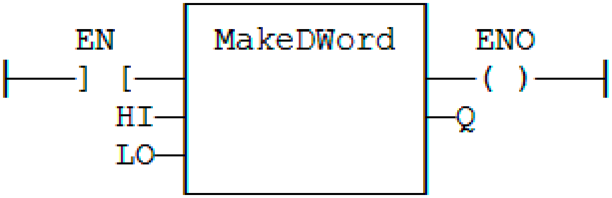
REMARKS
In LD language, the operation is executed only if the input rung (EN) is **TRUE**. The output rung (ENO) keeps the same value as the input rung. In IL, the first input must be loaded in the current result before calling the function.

ST LANGUAGE
```
Q := MAKEDWORD (HI, LO);
```

FBD LANGUAGE



LD LANGUAGE
The function is executed only if EN is **TRUE**.
ENO keeps the same value as EN.



IL LANGUAGE

```
Op1: LD        HI
     MAKEDWORD LO
     ST        Q
```

SEE ALSO

LOBYTE
HIBYTE
LOWORD
HIWORD
MAKEWORD

MAKEWORD

FUNCTION

Builds a word as the concatenation of two bytes.

INPUTS

Name	Type	Description
HI	USINT	Highest significant byte.
LO	USINT	Lowest significant byte.

OUTPUTS

Name	Type	Description
Q	UINT	16 bit register.

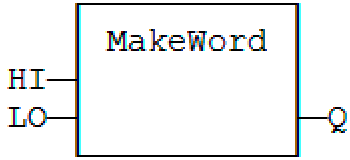
REMARKS

In LD language, the operation is executed only if the input rung (EN) is **TRUE**. The output rung (ENO) keeps the same value as the input rung. In IL, the first input must be loaded in the current result before calling the function.

ST LANGUAGE

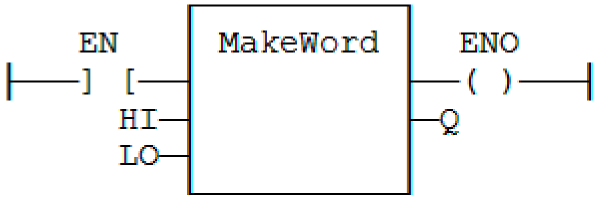
```
Q := MAKEWORD (HI, LO);
```

FBD LANGUAGE



LD LANGUAGE

The function is executed only if EN is **TRUE**.
ENO keeps the same value as EN.



IL LANGUAGE

```
Op1: LD       HI
     MAKEWORD LO
     ST       Q
```

SEE ALSO

LOBYTE
HIBYTE
LOWORD
HIWORD
MAKEDWORD

MBSHIFT

FUNCTION

Multibyte shift / rotate.

INPUTS

Name	Type	Description
Buffer	SINT/USINT	Array of bytes.
Pos	DINT	Base position in the array.
NbByte	DINT	Number of bytes to be shifted/rotated.

Name	Type	Description
NbShift	DINT	Number of shifts or rotations.
ToRight	BOOL	TRUE for right / FALSE for left.
Rotate	BOOL	TRUE for rotate / FALSE for shift.
InBit	BOOL	Bit to be introduced in a shift.

OUTPUTS

Name	Type	Description
Q	BOOL	TRUE if successful.

REMARKS

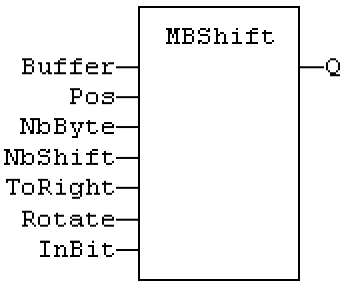
Use the ToRight argument to specify a shift to the left (**FALSE**) or to the right (**TRUE**). Use the Rotate argument to specify either a shift (**FALSE**) or a rotation (**TRUE**). In case of a shift, the InBit argument specifies the value of the bit that replaces the last shifted bit.

In LD language, the rung input (EN) validates the operation. The rung output is the result (Q).

ST LANGUAGE

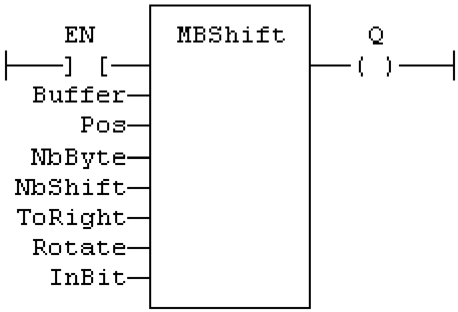
```
Q := MBShift (Buffer, Pos, NbByte, NbShift, ToRight, Rotate, InBit);
```

FBD LANGUAGE



LD LANGUAGE

The function is called only if EN is **TRUE**:



IL LANGUAGE
Not available.

NOT_MASK

FUNCTION
Performs a bit to bit negation of an integer value.

INPUTS

Name	Type	Description
IN	ANY	Integer input.

OUTPUTS

Name	Type	Description
Q	ANY	Bit to bit negation of the input.

REMARKS
Arguments can be signed or unsigned integers from 8 to 32 bits.

In LD language, the input rung (EN) enables the operation, and the output rung keeps the same value as the input rung. In IL language, the parameter (IN) must be loaded in the current result before calling the function.

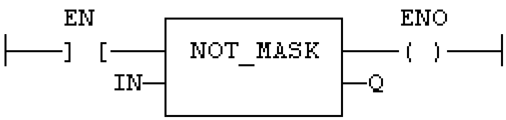
ST LANGUAGE

```
Q := NOT_MASK (IN);
```

FBD LANGUAGE

LD LANGUAGE

The function is executed only if EN is **TRUE**.
ENO is equal to EN.



IL LANGUAGE

```
Op1:  LD        IN
      NOT_MASK
      ST        Q
```

SEE ALSO

AND_MASK
OR_MASK
XOR_MASK

OR_MASK

FUNCTION

Performs a bit to bit OR between two integer values.

INPUTS

Name	Type	Description
IN	ANY	First input.
MSK	ANY	Second input (OR mask).

OUTPUTS

Name	Type	Description
Q	ANY	OR mask between IN and MSK inputs.

REMARKS

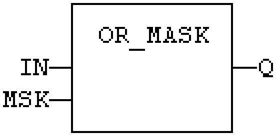
Arguments can be signed or unsigned integers from 8 to 32 bits.

In LD language, the input rung (EN) enables the operation, and the output rung keeps the same value as the input rung. In IL language, the first parameter (IN) must be loaded in the current result before calling the function. The other input is the operands of the function.

ST LANGUAGE

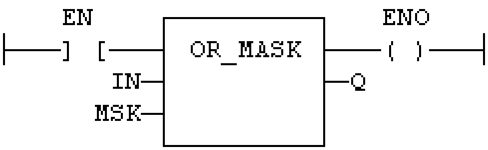
```
Q := OR_MASK (IN, MSK);
```

FBD LANGUAGE



LD LANGUAGE

The function is executed only if EN is **TRUE**.
ENO is equal to EN.



IL LANGUAGE

```
Op1: LD      IN
     OR_MASK MSK
     ST      Q
```

SEE ALSO

AND_MASK
XOR_MASK
NOT_MASK

PACK8

FUNCTION

Builds a byte with bits.

INPUTS

Name	Type	Description
IN0	BOOL	Less significant bit.
...		
IN7	BOOL	Most significant bit.

OUTPUTS

Name	Type	Description
Q	USINT	Byte built with input bits.

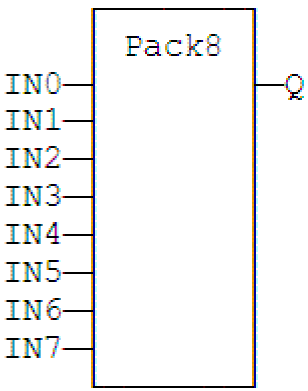
REMARKS

In LD language, the input rung is the IN0 input. The output rung (ENO) keeps the same value as the input rung. In IL, the input must be loaded in the current result before calling the function.

ST LANGUAGE

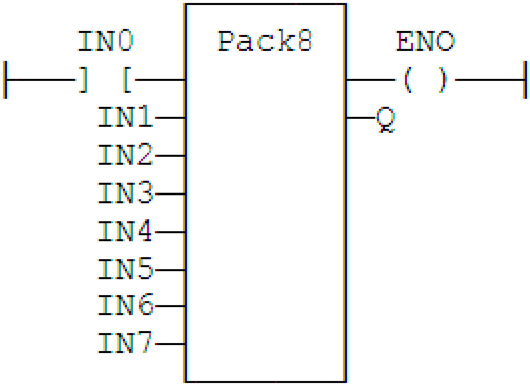
```
Q := PACK8 (IN0, IN1, IN2, IN3, IN4, IN5, IN6, IN7);
```

FBD LANGUAGE



LD LANGUAGE

ENO keeps the same value as EN.



IL LANGUAGE

```
Op1: LD    IN0
     PACK8 IN1, IN2, IN3, IN4, IN5, IN6, IN7
     ST    Q
```

SEE ALSO

UNPACK8

ROL

FUNCTION

Rotate bits of a register to the left.

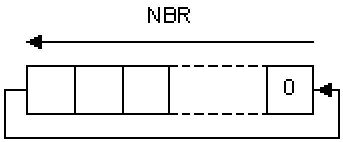
INPUTS

Name	Type	Description
IN	ANY	register.
NBR	DINT	Number of rotations (each rotation is 1 bit).

OUTPUTS

Name	Type	Description
Q	ANY	Rotated register.

DIAGRAM



REMARKS

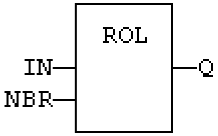
Arguments can be signed or unsigned integers from 8 to 32 bits.

In LD language, the input rung (EN) enables the operation, and the output rung keeps the state of the input rung. In IL language, the first input must be loaded before the function call. The second input is the operand of the function.

ST LANGUAGE

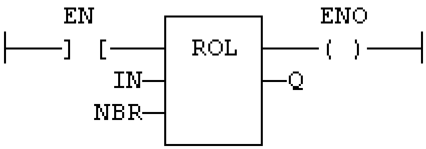
```
Q := ROL (IN, NBR);
```

FBD LANGUAGE



LD LANGUAGE

The rotation is executed only if EN is **TRUE**.
ENO has the same value as EN.



IL LANGUAGE

```
Op1: LD  IN
     ROL NBR
     ST  Q
```

SEE ALSO

SHL

SHR

ROR

ROR

FUNCTION
Rotate bits of a register to the right.

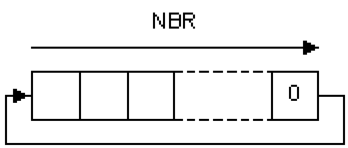
INPUTS

Name	Type	Description
IN	ANY	register.
NBR	ANY	Number of rotations (each rotation is 1 bit).

OUTPUTS

Name	Type	Description
Q	ANY	Rotated register.

DIAGRAM



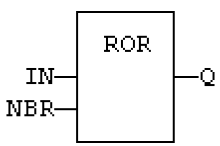
REMARKS
Arguments can be signed or unsigned integers from 8 to 32 bits.

In LD language, the input rung (EN) enables the operation, and the output rung keeps the state of the input rung. In IL language, the first input must be loaded before the function call. The second input is the operand of the function.

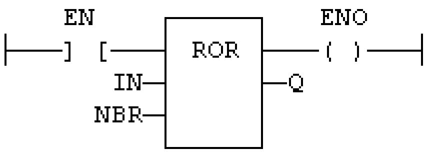
ST LANGUAGE

```
Q := ROR (IN, NBR);
```

FBD LANGUAGE



LD LANGUAGE
The rotation is executed only if EN is **TRUE**.
ENO has the same value as EN.



IL LANGUAGE

```
Op1: LD   IN
     ROR  NBR
     ST   Q
```

SEE ALSO

SHL

SHR

ROL

SETBIT

FUNCTION

Set a bit in an integer register.

INPUTS

Name	Type	Description
IN	ANY	8 to 32 bit integer register.
BIT	DINT	Bit number (0 = less significant bit).
VAL	BOOL	Bit value to apply.

OUTPUTS

Name	Type	Description
Q	ANY	Modified register.

REMARKS

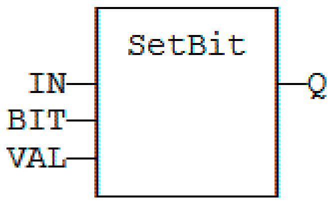
Types **LINT**, **REAL**, **LREAL**, **TIME** and **STRING** are not supported for IN and Q. IN and Q must have the same type. In case of invalid arguments (bad bit number or invalid input type) the function returns the value of IN without modification.

In LD language, the operation is executed only if the input rung (EN) is **TRUE**. The output rung (ENO) keeps the same value as the input rung.

ST LANGUAGE

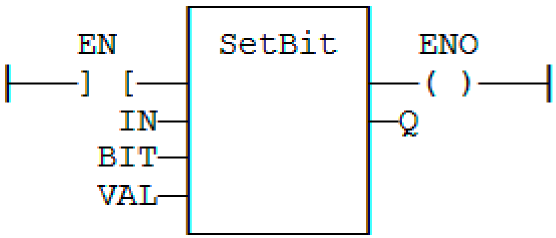
```
Q := SETBIT (IN, BIT, VAL);
```

FBD LANGUAGE



LD LANGUAGE

The function is executed only if EN is **TRUE**.
ENO keeps the same value as EN.



IL LANGUAGE

Not available.

SEE ALSO

TESTBIT

SHL

FUNCTION

Shift bits of a register to the left.

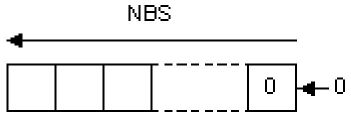
INPUTS

Name	Type	Description
IN	ANY	register.
NBS	ANY	Number of shifts (each shift is 1 bit).

OUTPUTS

Name	Type	Description
Q	ANY	Shifted register.

DIAGRAM



REMARKS

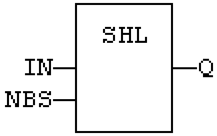
Arguments can be signed or unsigned integers from 8 to 32 bits.

In LD language, the input rung (EN) enables the operation, and the output rung keeps the state of the input rung. In IL language, the first input must be loaded before the function call. The second input is the operand of the function.

ST LANGUAGE

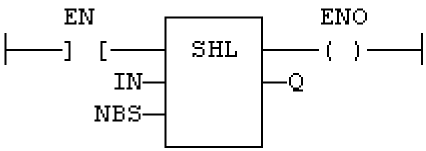
```
Q := SHL (IN, NBS);
```

FBD LANGUAGE



LD LANGUAGE

The shift is executed only if EN is **TRUE**.
ENO has the same value as EN.



IL LANGUAGE

```
Op1: LD   IN
     SHL  NBS
     ST   Q
```

SEE ALSO

SHR

ROL

ROR

SHR

FUNCTION
Shift bits of a register to the right.

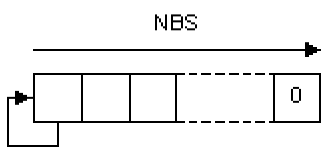
INPUTS

Name	Type	Description
IN	ANY	register.
NBS	ANY	Number of shifts (each shift is 1 bit).

OUTPUTS

Name	Type	Description
Q	ANY	Shifted register.

DIAGRAM



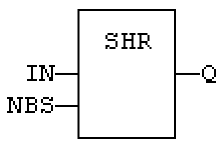
REMARKS
Arguments can be signed or unsigned integers from 8 to 32 bits.

In LD language, the input rung (EN) enables the operation, and the output rung keeps the state of the input rung. In IL language, the first input must be loaded before the function call. The second input is the operand of the function.

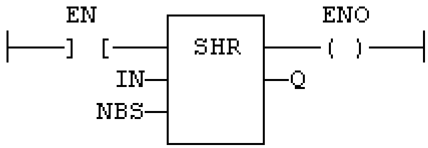
ST LANGUAGE

```
Q := SHR (IN, NBS);
```

FBD LANGUAGE



LD LANGUAGE
The shift is executed only if EN is **TRUE**.
ENO has the same value as EN.



IL LANGUAGE

```
Op1: LD  IN
     SHR NBS
     ST  Q
```

SEE ALSO

SHL

ROL

ROR

TESTBIT

FUNCTION

Test a bit of an integer register.

INPUTS

Name	Type	Description
IN	ANY	8 to 32 bit integer register.
BIT	DINT	Bit number (0 = less significant bit).

OUTPUTS

Name	Type	Description
Q	BOOL	Bit value.

REMARKS

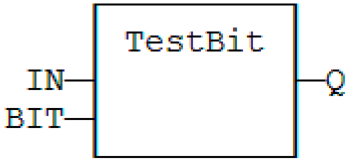
Types **LINT**, **REAL**, **LREAL**, **TIME** and **STRING** are not supported for IN and Q. IN and Q must have the same type. In case of invalid arguments (bad bit number or invalid input type) the function returns **FALSE**.

In LD language, the operation is executed only if the input rung (EN) is **TRUE**. The output rung is the output of the function.

ST LANGUAGE

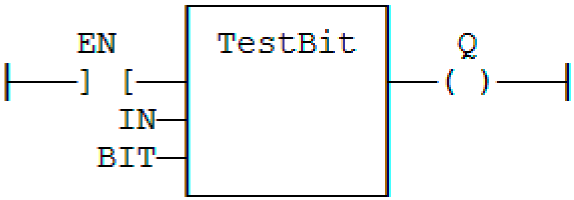
```
Q := TESTBIT (IN, BIT);
```

FBD LANGUAGE



LD LANGUAGE

The function is executed only if EN is **TRUE**.



IL LANGUAGE

Not available.

SEE ALSO

SETBIT

UNPACK8

FUNCTION BLOCK

Extract bits of a byte.

INPUTS

Name	Type	Description
IN	USINT	8 bit register.

OUTPUTS

Name	Type	Description
Q0	BOOL	Less significant bit.
...		
Q7	BOOL	Most significant bit.

REMARKS

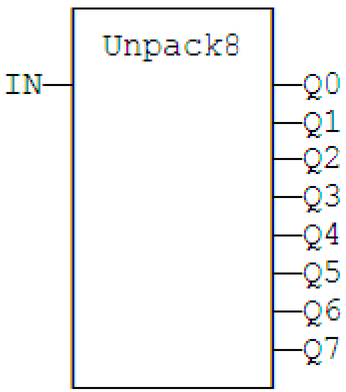
In LD language, the output rung is the Q0 output. The operation is executed only in the input rung (EN) is **TRUE**.

ST LANGUAGE

MyUnpack is a declared instance of the UNPACK8 function block.

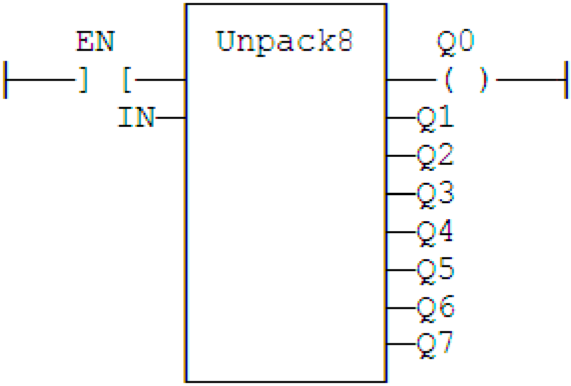
```
MyUnpack (IN);

Q0 := MyUnpack.Q0;
Q1 := MyUnpack.Q1;
Q2 := MyUnpack.Q2;
Q3 := MyUnpack.Q3;
Q4 := MyUnpack.Q4;
Q5 := MyUnpack.Q5;
Q6 := MyUnpack.Q6;
Q7 := MyUnpack.Q7;
```

FBD LANGUAGE



LD LANGUAGE

The operation is performed if EN = **TRUE**:



IL LANGUAGE

MyUnpack is a declared instance of the UNPACK8 function block.

```
Op1: CAL MyUnpack (IN)
     LD  MyUnpack.Q0
     ST  Q0
     (* ... *)
     LD  MyUnpack.Q7
     ST  Q7
```

SEE ALSO

PACK8

XOR_MASK

FUNCTION

Performs a bit to bit exclusive OR between two integer values

INPUTS

Name	Type	Description
IN	ANY	First input.
MSK	ANY	Second input (XOR mask).

OUTPUTS

Name	Type	Description
Q	ANY	Exclusive OR mask between IN and MSK inputs.

REMARKS

Arguments can be signed or unsigned integers from 8 to 32 bits.

In LD language, the input rung (EN) enables the operation, and the output rung keeps the same value as the input rung. In IL language, the first parameter (IN) must be loaded in the current result before calling the function. The other input is the operands of the function.

ST LANGUAGE

```
Q := XOR_MASK (IN, MSK);
```

FBD LANGUAGE

LD LANGUAGE

The function is executed only if EN is **TRUE**.
ENO is equal to EN.

IL LANGUAGE

```
Op1: LD       IN
     XOR_MASK MSK
     ST       Q
```

SEE ALSO

AND_MASK
OR_MASK
NOT_MASK

Counters

Standard blocks for managing counters:

Name	Description
CTU	Up counter
CTD	Down counter
CTUD	Up / Down counter

CTD / CTDr

FUNCTION BLOCK
Down counter.

INPUTS

Name	Type	Description
CD	BOOL	Enable counting. Counter is decreased on each call when CU is TRUE.
LOAD	BOOL	Re-load command. Counter is set to PV when called with LOAD to TRUE.
PV	DINT	Programmed maximum value.

OUTPUTS

Name	Type	Description
Q	BOOL	TRUE when counter is empty, i.e. when CV = 0.
CV	DINT	Current value of the counter.

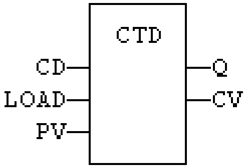
REMARKS
The counter is empty (CV = 0) when the application starts. The counter does not include a pulse detection for CD input. Use **R_TRIG** or **F_TRIG** function block for counting pulses of CD input signal. In LD language, CD is the input rung. The output rung is the Q output.

CTUr, CTDr, CTUDr function blocks operate exactly as other counters, except that all boolean inputs (CU, CD, **RESET**, **LOAD**) have an implicit rising edge detection included. Not that these counters may be not supported on some target systems.

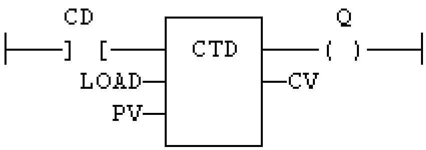
ST LANGUAGE
MyCounter is a declared instance of CTD function block.

```
MyCounter (CD, LOAD, PV);
Q := MyCounter.Q;
CV := MyCounter.CV;
```

FBD LANGUAGE



LD LANGUAGE



IL LANGUAGE

MyCounter is a declared instance of CTD function block.

```
Op1: CAL    MyCounter (CD, LOAD, PV)
     LD     MyCounter.Q
     ST     Q
     LD     MyCounter.CV
     ST     CV
```

SEE ALSO

CTU
CTUD

CTU / CTUr

FUNCTION BLOCK

Up counter.

INPUTS

Name	Type	Description
CU	BOOL	Enable counting. Counter is increased on each call when CU is TRUE.
RESET	BOOL	Reset command. Counter is reset to 0 when called with RESET to TRUE.
PV	DINT	Programmed maximum value.

OUTPUTS

Name	Type	Description
Q	BOOL	TRUE when counter is full, i.e. when CV = PV.
CV	DINT	Current value of the counter.

REMARKS

The counter is empty (CV = 0) when the application starts. The counter does not include a pulse detection for CU input. Use **R_TRIG** or **F_TRIG** function block for counting pulses of CU input signal. In LD language, CU is the input rung. The output rung is the Q output.

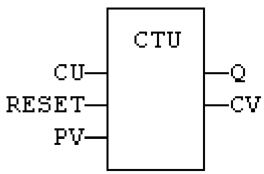
CTUr, CTDr, CTUDr function blocks operate exactly as other counters, except that all boolean inputs (CU, CD, **RESET**, **LOAD**) have an implicit rising edge detection included. Not that these counters may be not supported on some target systems.

ST LANGUAGE

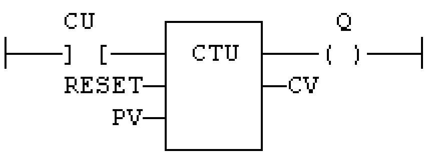
MyCounter is a declared instance of CTU function block.

```
MyCounter (CU, RESET, PV);
Q := MyCounter.Q;
CV := MyCounter.CV;
```

FBD LANGUAGE



LD LANGUAGE



IL LANGUAGE

MyCounter is a declared instance of CTU function block.

```
Op1: CAL    MyCounter (CU, RESET, PV)
     LD     MyCounter.Q
     ST     Q
     LD     MyCounter.CV
     ST     CV
```

SEE ALSO

CTD

CTUD / CTUDr

FUNCTION BLOCK

Up/down counter.

INPUTS

Name	Type	Description
CU	BOOL	Enable counting. Counter is increased on each call when CU is TRUE.
CD	BOOL	Enable counting. Counter is decreased on each call when CD is TRUE.
RESET	BOOL	Reset command. Counter is reset to 0 called with RESET to TRUE.
LOAD	BOOL	Re-load command. Counter is set to PV when called with LOAD to TRUE.
PV	DINT	Programmed maximum value.

OUTPUTS

Name	Type	Description
QU	BOOL	TRUE when counter is full, i.e. when CV = PV.
QD	BOOL	TRUE when counter is empty, i.e. when CV = 0.
CV	DINT	Current value of the counter.

REMARKS

The counter is empty (CV = 0) when the application starts. The counter does not include a pulse detection for CU and CD inputs. Use **R_TRIG** or **F_TRIG** function blocks for counting pulses of CU or CD input signals. In LD language, CU is the input rung. The output rung is the QU output.

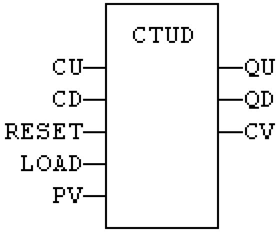
CTUr, CTDr, CTUDr function blocks operate exactly as other counters, except that all boolean inputs (CU, CD, **RESET**, **LOAD**) have an implicit rising edge detection included. Not that these counters may be not supported on some target systems.

ST LANGUAGE

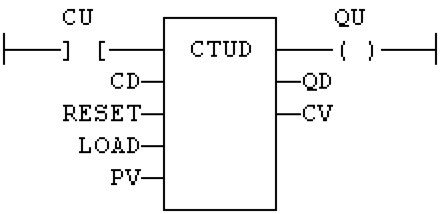
MyCounter is a declared instance of **CTUD** function block.

```
MyCounter (CU, CD, RESET, LOAD, PV);
QU := MyCounter.QU;
QD := MyCounter.QD;
CV := MyCounter.CV;
```

FBD LANGUAGE



LD LANGUAGE



IL LANGUAGE

MyCounter is a declared instance of **CTUD** function block.

```
Op1: CAL     MyCounter (CU, CD, RESET, LOAD, PV)
     LD      MyCounter.QU
     ST      QU
     LD      MyCounter.QD
     ST      QD
     LD      MyCounter.CV
     ST      CV
```

SEE ALSO

CTU

CTD

Timers

STANDARD FUNCTIONS FOR MANAGING TIMERS:

Function	Effect
TON	On timer.
TOF	Off timer.
TP	Pulse timer.
BLINK	Blinker.

Function	Effect
BLINKA	Asymetric blinker.
PLS	Pulse signal generator.
TMU	Up-counting stop-timer.
TMD	Down-counting stop-timer.

BLINK

FUNCTION BLOCK

Blinker.

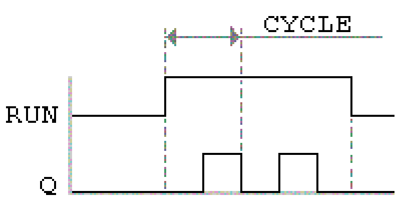
INPUTS

Name	Type	Description
RUN	BOOL	Enabling command.
CYCLE	TIME	Blinking period.

OUTPUTS

Name	Type	Description
Q	BOOL	Output blinking signal.

TIME DIAGRAM



REMARKS

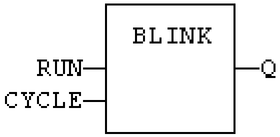
The output signal is **FALSE** when the RUN input is **FALSE**. The **CYCLE** input is the complete period of the blinking signal. In LD language, the input rung is the IN command. The output rung is the Q output signal.

ST LANGUAGE

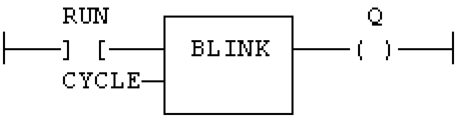
MyBlinker is a declared instance of **BLINK** function block.

```
MyBlinker (RUN, CYCLE);
Q := MyBlinker.Q;
```

FBD LANGUAGE



LD LANGUAGE



IL LANGUAGE

MyBlinker is a declared instance of **BLINK** function block.

```
Op1: CAL MyBlinker (RUN, CYCLE)
     LD  MyBlinker.Q
     ST  Q
```

SEE ALSO

TON
TOF
TP

BLINKA

FUNCTION BLOCK

Asymetric blinker.

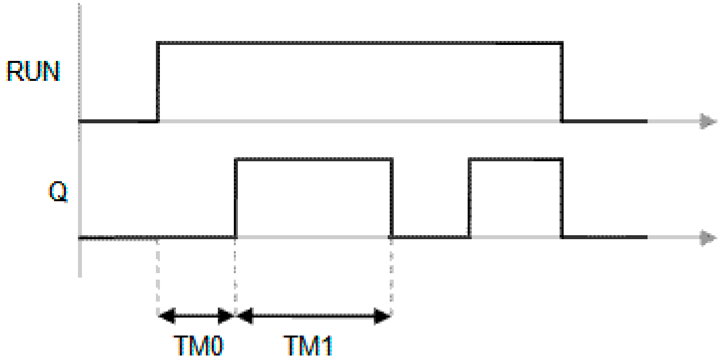
INPUTS

Name	Type	Description
RUN	BOOL	Enabling command.
TM0	TIME	Duration of FALSE state on output.
TM1	TIME	Duration of TRUE state on output.

OUTPUTS

Name	Type	Description
Q	BOOL	Output blinking signal.

TIME DIAGRAM



REMARKS

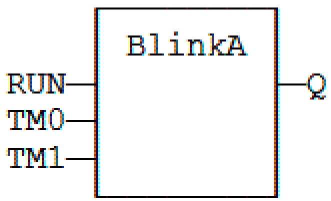
The output signal is **FALSE** when the RUN input is **FALSE**. In LD language, the input rung is the IN command. The output rung is the Q output signal.

ST LANGUAGE

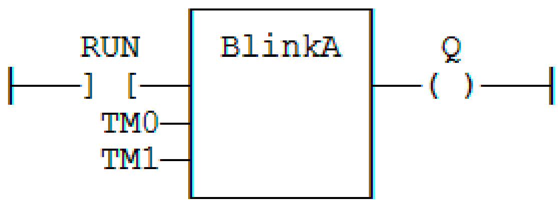
MyBlinker is a declared instance of **BLINKA** function block.

```
MyBlinker (RUN, TM0, TM1);
Q := MyBlinker.Q;
```

FBD LANGUAGE



LD LANGUAGE



IL LANGUAGE

MyBlinker is a declared instance of **BLINKA** function block.

```
Op1: CAL MyBlinker (RUN, TM0, TM1)
     LD  MyBlinker.Q
     ST  Q
```

SEE ALSO

TON

TOF

TP

PLS

FUNCTION BLOCK

Pulse signal generator:

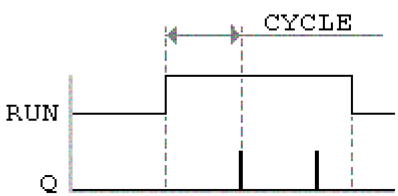
INPUTS

Name	Type	Description
RUN	BOOL	Enabling command.
CYCLE	TIME	Signal period.

OUTPUTS

Name	Type	Description
Q	BOOL	Output pulse signal.

TIME DIAGRAM



REMARKS

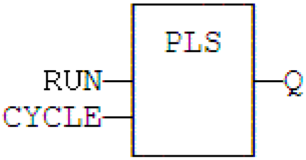
On every period, the output is set to **TRUE** during one cycle only. In LD language, the input rung is the IN command. The output rung is the Q output signal.

ST LANGUAGE

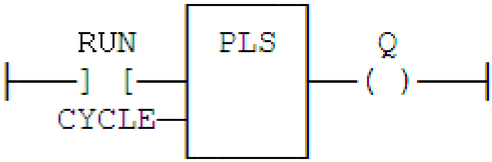
MyPLS is a declared instance of PLS function block:

```
MyPLS (RUN, CYCLE);
Q := MyPLS.Q;
```

FBD LANGUAGE



LD LANGUAGE



IL LANGUAGE

MyPLS is a declared instance of PLS function block:

```
Op1: CAL MyPLS (RUN, CYCLE)
     LD  MyPLS.Q
     ST  Q
```

SEE ALSO

TON

TOF

TP

TMD

FUNCTION BLOCK

Down-counting stop timer.

INPUTS

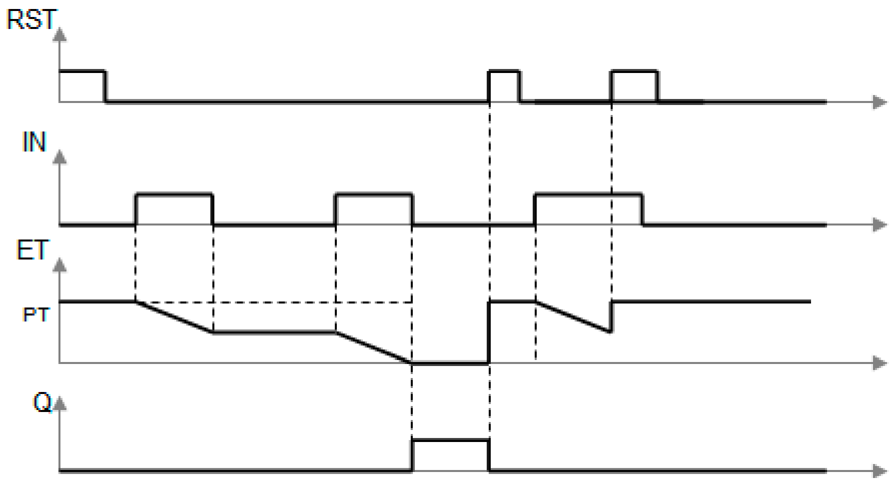
Name	Type	Description
IN	BOOL	The time counts when this input is TRUE.
RST	BOOL	Timer is reset to PT when this input is TRUE.
PT	TIME	Programmed time.

OUTPUTS

Name	Type	Description
Q	BOOL	Timer elapsed output signal.

Name	Type	Description
ET	TIME	Elapsed time.

TIME DIAGRAM



REMARKS

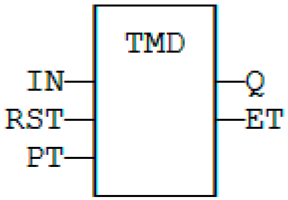
The timer counts up when the IN input is **TRUE**. It stops when the programmed time is elapsed. The timer is reset when the RST input is **TRUE**. It is not reset when IN is false.

ST LANGUAGE

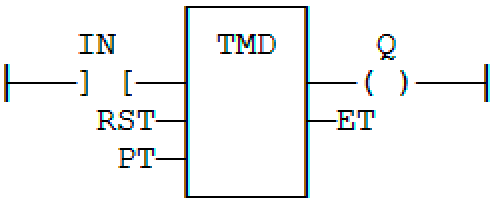
MyTimer is a declared instance of TMD function block.

```
MyTimer (IN, RST, PT);
Q := MyTimer.Q;
ET := MyTimer.ET;
```

FBD LANGUAGE



LD LANGUAGE



IL LANGUAGE

MyTimer is a declared instance of TMD function block.

```
Op1: CAL MyTimer (IN, RST, PT)
     LD  MyTimer.Q
     ST  Q
     LD  MyTimer.ET
     ST  ET
```

SEE ALSO

TMU

TMU

FUNCTION BLOCK

Up-counting stop watch.

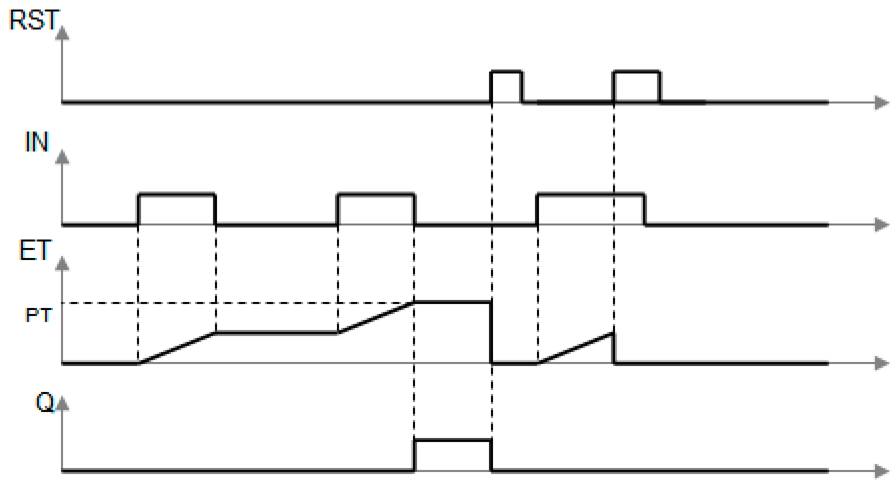
INPUTS

Name	Type	Description
IN	BOOL	The time counts when this input is TRUE.
RST	BOOL	Timer is reset to 0 when this input is TRUE.
PT	TIME	Programmed time.

OUTPUTS

Name	Type	Description
Q	BOOL	Timer elapsed output signal.
ET	TIME	Elapsed time.

TIME DIAGRAM



REMARKS

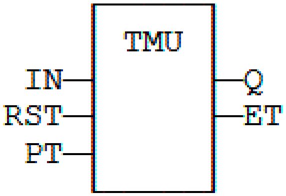
The timer counts up when the IN input is **TRUE**. It stops when the programmed time is elapsed. The timer is reset when the RST input is **TRUE**. It is not reset when IN is false.

ST LANGUAGE

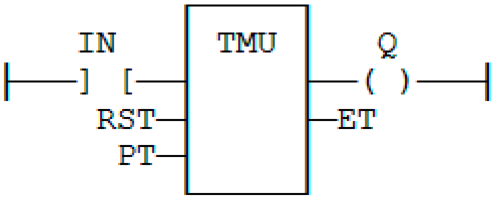
MyTimer is a declared instance of TMU function block.

```
MyTimer (IN, RST, PT);
Q := MyTimer.Q;
ET := MyTimer.ET;
```

FBD LANGUAGE



LD LANGUAGE



IL LANGUAGE

MyTimer is a declared instance of TMU function block.

```
Op1: CAL MyTimer (IN, RST, PT)
     LD  MyTimer.Q
     ST  Q
     LD  MyTimer.ET
     ST  ET
```

SEE ALSO

TMD

TOF / TOFR

FUNCTION BLOCK

Off timer.

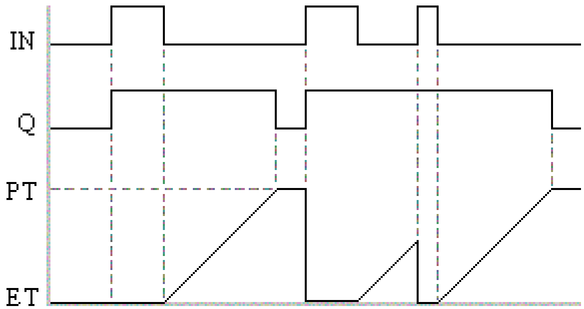
INPUTS

Name	Type	Description
IN	BOOL	Timer command.
PT	TIME	Programmed time.
RST	BOOL	Reset (TOFR only).

OUTPUTS

Name	Type	Description
Q	BOOL	Timer elapsed output signal.
ET	TIME	Elapsed time.

TIME DIAGRAM



REMARKS

The timer starts on a falling pulse of IN input. It stops when the elapsed time is equal to the programmed time. A rising pulse of IN input resets the timer to 0. The output signal is set to **TRUE** on when the IN input rises to **TRUE**, reset to **FALSE** when programmed time is elapsed.

TOFR is same as TOF but has an extra input for resetting the timer.

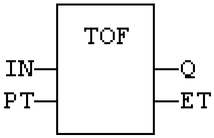
In LD language, the input rung is the IN command. The output rung is Q the output signal.

ST LANGUAGE

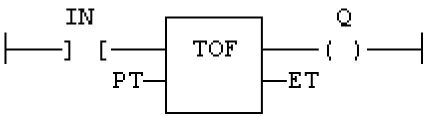
MyTimer is a declared instance of TOF function block.

```
MyTimer (IN, PT);
Q := MyTimer.Q;
ET := MyTimer.ET;
```

FBD LANGUAGE



LD LANGUAGE



IL LANGUAGE

MyTimer is a declared instance of TOF function block.

```
Op1: CAL MyTimer (IN, PT)
     LD  MyTimer.Q
     ST  Q
     LD  MyTimer.ET
     ST  ET
```

SEE ALSO

TON
TP
BLINK

TON

FUNCTION BLOCK

On timer.

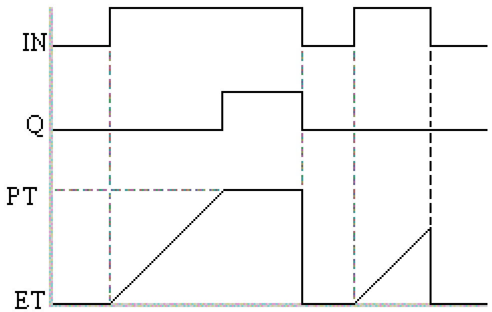
INPUTS

Name	Type	Description
IN	BOOL	Timer command.
PT	TIME	Programmed time.

OUTPUTS

Name	Type	Description
Q	BOOL	Timer elapsed output signal.
ET	TIME	Elapsed time.

TIME DIAGRAM



REMARKS

The timer starts on a rising pulse of IN input. It stops when the elapsed time is equal to the programmed time. A falling pulse of IN input resets the timer to 0. The output signal is set to **TRUE** when programmed time is elapsed, and reset to **FALSE** when the input command falls.

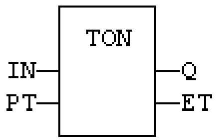
In LD language, the input rung is the IN command. The output rung is Q the output signal.

ST LANGUAGE

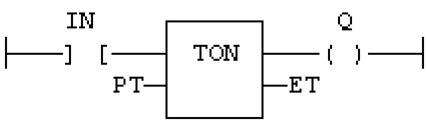
MyTimer is a declared instance of TON function block.

```
MyTimer (IN, PT);
Q := MyTimer.Q;
ET := MyTimer.ET;
```

FBD LANGUAGE



LD LANGUAGE



IL LANGUAGE

MyTimer is a declared instance of TON function block.

```
Op1: CAL MyTimer (IN, PT)
     LD  MyTimer.Q
     ST  Q
     LD  MyTimer.ET
```

ST ET

SEE ALSO
TOF
TP
BLINK

TP / TPR

FUNCTION BLOCK
Pulse timer.

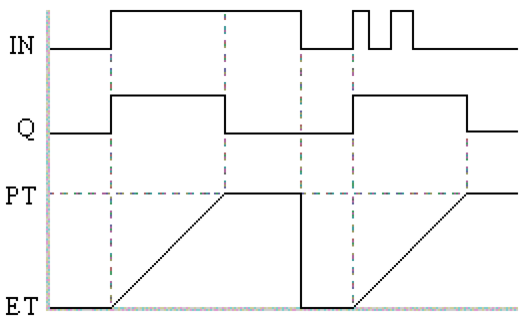
INPUTS

Name	Type	Description
IN	BOOL	Timer command.
PT	TIME	Programmed time.
RST	BOOL	Reset (TPR only).

OUTPUTS

Name	Type	Description
Q	BOOL	Timer elapsed output signal.
ET	TIME	Elapsed time.

TIME DIAGRAM



REMARKS
The timer starts on a rising pulse of IN input. It stops when the elapsed time is equal to the programmed time. A falling pulse of IN input resets the timer to 0, only if the programmed time is elapsed. All pulses of IN while the timer is running are ignored. The output signal is set to **TRUE** while the timer is running.

TPR is same as TP but has an extra input for resetting the timer

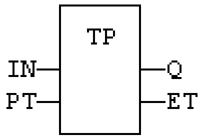
In LD language, the input rung is the IN command. The output rung is Q the output signal.

ST LANGUAGE

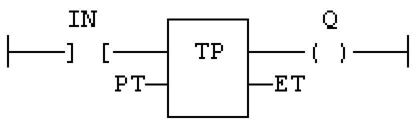
MyTimer is a declared instance of TP function block.

```
MyTimer (IN, PT);
Q := MyTimer.Q;
ET := MyTimer.ET;
```

FBD LANGUAGE



LD LANGUAGE



IL LANGUAGE

MyTimer is a declared instance of TP function block.

```
Op1: CAL MyTimer (IN, PT)
     LD  MyTimer.Q
     ST  Q
     LD  MyTimer.ET
     ST  ET
```

SEE ALSO

TON

TOF

BLINK

Mathematical Operations

STANDARD MATHEMATICAL FUNCTIONS

Name	Description
ABS	Absolute value
TRUNC	Integer part (truncate)

Name	Description
LOG	Logarithm (Base 10)
LN	Natural logarithm
POW	Raise to a power
EXPT	Raise to a power
EXP	Natural power (power of *e*)
SQRT	Square root
ROOT	Root extraction
SCALELIN	scaling - linear conversion

ABS

FUNCTION

Returns the absolute value of the input.

INPUTS

Name	Type	Description
IN	ANY	value.

OUTPUTS

Name	Type	Description
Q	ANY	Result: absolute value of IN.

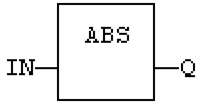
REMARKS

In LD language, the operation is executed only if the input rung (EN) is **TRUE**. The output rung (ENO) keeps the same value as the input rung. In IL, the input must be loaded in the current result before calling the function.

ST LANGUAGE

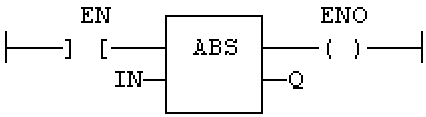
```
Q := ABS (IN);
```

FBD LANGUAGE



LD LANGUAGE

The function is executed only if EN is **TRUE**.
ENO keeps the same value as EN.



IL LANGUAGE

```
Op1: LD  IN
     ABS
     ST  Q     (* Q is: ABS (IN) *)
```

SEE ALSO

TRUNC
LOG
POW
SQRT

EXP / EXPL

FUNCTION

Calculates the natural exponential of the input.

INPUTS

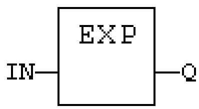
Name	Type	Description
IN	REAL/LREAL	Real value.

OUTPUTS

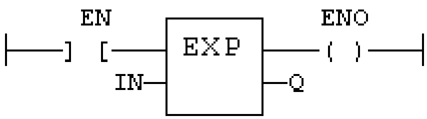
Name	Type	Description
Q	REAL/LREAL	Result: natural exponential of IN.

REMARKS
In LD language, the operation is executed only if the input rung (EN) is **TRUE**. The output rung (ENO) keeps the same value as the input rung. In IL, the input must be loaded in the current result before calling the function.

ST LANGUAGE
```
Q := EXP (IN);
```

FBD LANGUAGE


LD LANGUAGE
The function is executed only if EN is **TRUE**.
ENO keeps the same value as EN.



IL LANGUAGE
```
Op1: LD   IN
     EXP
     ST   Q    (* Q is: EXP (IN) *)
```

SEE ALSO
ABS
TRUNC
POW
SQRT

EXPT

FUNCTION
Calculates a power.

INPUTS

Name	Type	Description
IN	REAL	Real value.

Name	Type	Description
EXP	DINT	Exponent.

OUTPUTS

Name	Type	Description
Q	REAL	Result: IN to the 'EXP' power.

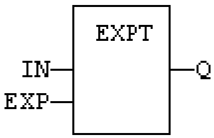
REMARKS

In LD language, the operation is executed only if the input rung (EN) is **TRUE**. The output rung (ENO) keeps the same value as the input rung. In IL, the input must be loaded in the current result before calling the function. The exponent (second input of the function) must be the operand of the function.

ST LANGUAGE

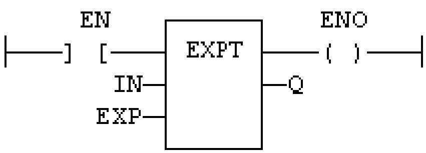
```
Q := EXPT (IN, EXP);
```

FBD LANGUAGE



LD LANGUAGE

The function is executed only if EN is **TRUE**.
ENO keeps the same value as EN.



IL LANGUAGE

```
Op1: LD   IN
     EXPT EXP
     ST   Q    (* Q is: (IN ** EXP) *)
```

SEE ALSO

ABS
TRUNC
LOG
SQRT

LOG

FUNCTION
Calculates the logarithm (base 10) of the input.

INPUTS

Name	Type	Description
IN	REAL	Real value.

OUTPUTS

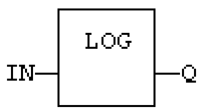
Name	Type	Description
Q	REAL	Result: logarithm (base 10) of IN.

REMARKS
In LD language, the operation is executed only if the input rung (EN) is **TRUE**. The output rung (ENO) keeps the same value as the input rung. In IL, the input must be loaded in the current result before calling the function.

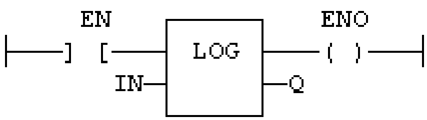
ST LANGUAGE

```
Q := LOG (IN);
```

FBD LANGUAGE



LD LANGUAGE
The function is executed only if EN is **TRUE**.
ENO keeps the same value as EN.



IL LANGUAGE

```
Op1: LD  IN
     LOG
     ST  Q    (* Q is: LOG (IN) *)
```

SEE ALSO
ABS

TRUNC
POW
SQRT

LN

FUNCTION
Calculates the natural logarithm of the input.

INPUTS

Name	Type	Description
IN	REAL/LREAL	Real value.

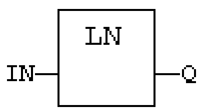
OUTPUTS

Name	Type	Description
Q	REAL/LREAL	Result: natural logarithm of IN.

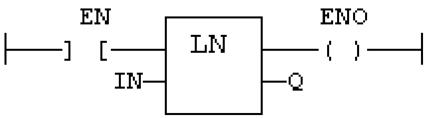
REMARKS
In LD language, the operation is executed only if the input rung (EN) is **TRUE**. The output rung (ENO) keeps the same value as the input rung. In IL, the input must be loaded in the current result before calling the function.

ST LANGUAGE
```
Q := LN (IN);
```

FBD LANGUAGE



LD LANGUAGE
The function is executed only if EN is **TRUE**.
ENO keeps the same value as EN.



IL LANGUAGE

```
Op1: LD   IN
     LN
     ST   Q     (* Q is: LN (IN) *)
```

SEE ALSO

ABS
TRUNC
POW
SQRT

POW ** POWL

FUNCTION

Calculates a power.

INPUTS

Name	Type	Description
IN	REAL/LREAL	Real value.
EXP	REAL/LREAL	Exponent.

OUTPUTS

Name	Type	Description
Q	REAL/LREAL	Result: IN at the ‘EXP’ power.

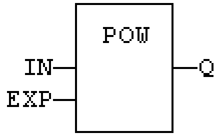
REMARKS

Alternatively, in ST language, the ** operator can be used. In LD language, the operation is executed only if the input rung (EN) is **TRUE**. The output rung (ENO) keeps the same value as the input rung. In IL, the input must be loaded in the current result before calling the function. The exponent (second input of the function) must be the operand of the function.

ST LANGUAGE

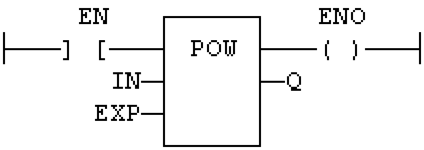
```
Q := POW (IN, EXP);
Q := IN ** EXP;
```

FBD LANGUAGE



LD LANGUAGE

The function is executed only if EN is **TRUE**.
ENO keeps the same value as EN.



IL LANGUAGE

```
Op1: LD  IN
     POW EXP
     ST  Q     (* Q is: (IN ** EXP) *)
```

SEE ALSO

ABS
TRUNC
LOG
SQRT

ROOT

FUNCTION

Calculates the Nth root of the input.

INPUTS

Name	Type	Description
IN	REAL	Real value
N	DINT	Root level

OUTPUTS

Name	Type	Description
Q	REAL	Result: Nth root of IN

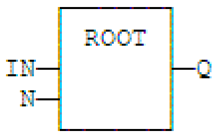
REMARKS

In LD language, the operation is executed only if the input rung (EN) is **TRUE**. The output rung (ENO) keeps the same value as the input rung. In IL, the input must be loaded in the current result before calling the function.

ST LANGUAGE

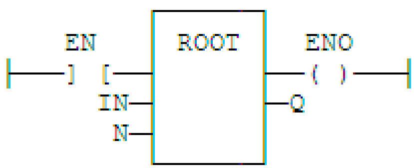
```
Q := ROOT (IN, N);
```

FBD LANGUAGE



LD LANGUAGE

The function is executed only if EN is **TRUE**.
ENO keeps the same value as EN.



IL LANGUAGE

```
Op1: LD   IN
     ROOT N
     ST   Q     (* Q is: ROOT (IN) *)
```

ScaleLin

FUNCTION

Scaling - linear conversion.

INPUTS

Name	Type	Description
IN	REAL	Real value.
IMIN	REAL	Minimum input value.
IMAX	REAL	Maximum input value.
OMIN	REAL	Minimum output value.
OMAX	REAL	Maximum output value.

OUTPUTS

Name	Type	Description
OUT	REAL	Result: OMIN + IN * (OMAX - OMIN) / (IMAX - IMIN).

TRUTH TABLE

Inputs	OUT
IMIN >= IMAX	= IN
IN < IMIN	= IMIN
IN > IMAX	= IMAX
other	= OMIN + IN * (OMAX - OMIN) / (IMAX - IMIN)

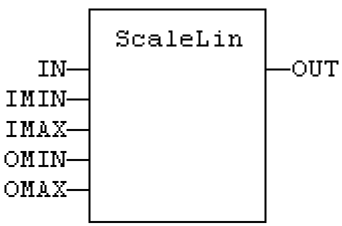
REMARKS

In LD language, the operation is executed only if the input rung (EN) is **TRUE**. The output rung (ENO) keeps the same value as the input rung. In IL, the input must be loaded in the current result before calling the function.

ST LANGUAGE

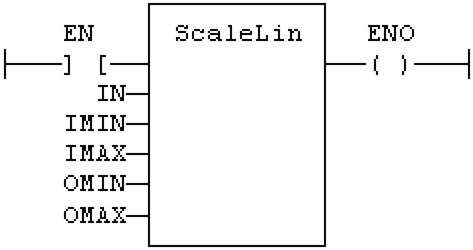
```
OUT := ScaleLin (IN, IMIN, IMAX, OMIN, OMAX);
```

FBD LANGUAGE



LD LANGUAGE

The function is executed only if EN is **TRUE**.
ENO keeps the same value as EN.



IL LANGUAGE

```
Op1: LD       IN
     ScaleLin IMAX, IMIN, OMAX, OMIN
     ST       OUT
```

SQRT / SQRTL

FUNCTION

Calculates the square root of the input.

INPUTS

Name	Type	Description
IN	REAL/LREAL	Real value.

OUTPUTS

Name	Type	Description
Q	REAL/LREAL	Result: square root of IN.

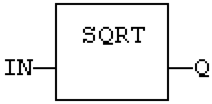
REMARKS

In LD language, the operation is executed only if the input rung (EN) is **TRUE**. The output rung (ENO) keeps the same value as the input rung. In IL, the input must be loaded in the current result before calling the function.

ST LANGUAGE

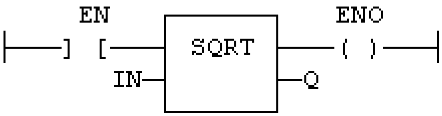
```
Q := SQRT (IN);
```

FBD LANGUAGE



LD LANGUAGE

The function is executed only if EN is **TRUE**.
ENO keeps the same value as EN.



IL LANGUAGE

```
Op1: LD   IN
     SQRT
     ST   Q    (* Q is: SQRT (IN) *)
```

SEE ALSO

ABS
TRUNC
LOG
POW

TRUNC / TRUNCL

FUNCTION

Truncates the decimal part of the input.

INPUTS

Name	Type	Description
IN	REAL/LREAL	Real value.

OUTPUTS

Name	Type	Description
Q	REAL/LREAL	Result: integer part of IN.

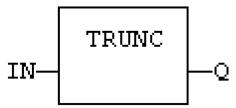
REMARKS

In LD language, the operation is executed only if the input rung (EN) is **TRUE**. The output rung (ENO) keeps the same value as the input rung. In IL, the input must be loaded in the current result before calling the function.

ST LANGUAGE

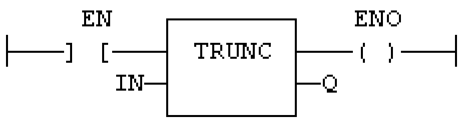
```
Q := TRUNC (IN);
```

FBD LANGUAGE



LD LANGUAGE

The function is executed only if EN is **TRUE**.
ENO keeps the same value as EN.



IL LANGUAGE

```
Op1: LD  IN
     TRUNC
     ST  Q    (* Q is the integer part of IN *)
```

SEE ALSO

ABS
LOG
POW
SQRT

Trigonometric Functions

STANDARD FUNCTIONS FOR TRIGONOMETRIC CALCULATION:

Name	Description
SIN	sine
COS	cosine
TAN	tangent
ASIN	arc-sine
ACOS	arc-cosine

Name	Description
ATAN	arc-tangent
ATAN2	arc-tangent of Y / X

SEE ALSO
`UseDegrees`

ACOS / ACOSL

FUNCTION
Calculate an arc-cosine.

INPUTS

Name	Type	Description
IN	REAL/LREAL	Real value.

OUTPUTS

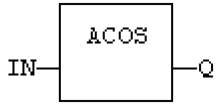
Name	Type	Description
Q	REAL/LREAL	Result: arc-cosine of IN.

REMARKS
In LD language, the operation is executed only if the input rung (EN) is **TRUE**. The output rung (ENO) keeps the same value as the input rung. In IL, the input must be loaded in the current result before calling the function.

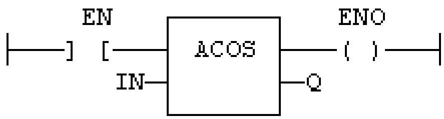
ST LANGUAGE

```
Q := ACOS (IN);
```

FBD LANGUAGE



LD LANGUAGE
The function is executed only if EN is **TRUE**.
ENO keeps the same value as EN.



IL LANGUAGE

```
Op1: LD  IN
     ACOS
     ST  Q    (* Q is: ACOS (IN) *)
```

SEE ALSO

SIN
COS
TAN
ASIN
ATAN
ATAN2

ASIN / ASINL

FUNCTION

Calculate an arc-sine.

INPUTS

Name	Type	Description
IN	REAL/LREAL	Real value.

OUTPUTS

Name	Type	Description
Q	REAL/LREAL	Result: arc-sine of IN.

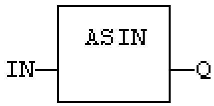
REMARKS

In LD language, the operation is executed only if the input rung (EN) is **TRUE**. The output rung (ENO) keeps the same value as the input rung. In IL, the input must be loaded in the current result before calling the function.

ST LANGUAGE

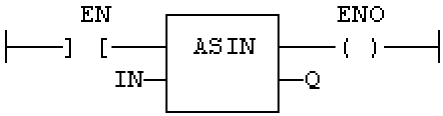
```
Q := ASIN (IN);
```

FBD LANGUAGE



LD LANGUAGE

The function is executed only if EN is **TRUE**.
ENO keeps the same value as EN.



IL LANGUAGE

```
    Op1: LD  IN
         ASIN
         ST  Q     (* Q is: ASIN (IN) *)
```

SEE ALSO

SIN
COS
TAN
ACOS
ATAN
ATAN2

ATAN / ATANL

FUNCTION

Calculate an arc-tangent.

INPUTS

Name	Type	Description
IN	REAL/LREAL	Real value.

OUTPUTS

Name	Type	Description
Q	REAL/LREAL	Result: arc-tangent of IN.

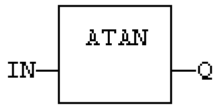
REMARKS

In LD language, the operation is executed only if the input rung (EN) is **TRUE**. The output rung (ENO) keeps the same value as the input rung. In IL, the input must be loaded in the current result before calling the function.

ST LANGUAGE

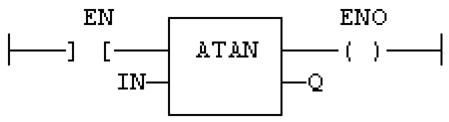
```
Q := ATAN (IN);
```

FBD LANGUAGE



LD LANGUAGE

The function is executed only if EN is **TRUE**.
ENO keeps the same value as EN.



IL LANGUAGE

```
Op1: LD   IN
     ATAN
     ST   Q    (* Q is: ATAN (IN) *)
```

SEE ALSO

SIN
COS
TAN
ASIN
ACOS
ATAN2

ATAN2 / ATANL2

FUNCTION

Calculate arc-tangent of Y/X.

INPUTS

Name	Type	Description
Y	REAL/LREAL	Real value.
X	REAL/LREAL	Real value.

OUTPUTS

Name	Type	Description
Q	REAL/LREAL	Result: arc-tangent of Y / X.

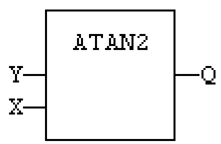
REMARKS

In LD language, the operation is executed only if the input rung (EN) is **TRUE**. The output rung (ENO) keeps the same value as the input rung. In IL, the input must be loaded in the current result before calling the function.

ST LANGUAGE

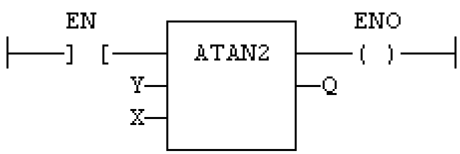
```
Q := ATAN2 (IN);
```

FBD LANGUAGE



LD LANGUAGE

The function is executed only if EN is **TRUE**.
ENO keeps the same value as EN.



IL LANGUAGE

```
Op1: LD   Y
     ATAN2 X
     ST   Q    (* Q is: ATAN2 (Y / X) *)
```

SEE ALSO

SIN
COS
TAN
ASIN

ACOS
ATAN

COS / COSL

FUNCTION
Calculate a cosine.

INPUTS

Name	Type	Description
IN	REAL/LREAL	Real value.

OUTPUTS

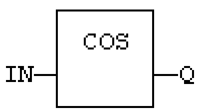
Name	Type	Description
Q	REAL/LREAL	Result: cosine of IN.

REMARKS
In LD language, the operation is executed only if the input rung (EN) is **TRUE**. The output rung (ENO) keeps the same value as the input rung. In IL, the input must be loaded in the current result before calling the function.

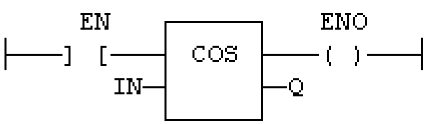
ST LANGUAGE

```
Q := COS (IN);
```

FBD LANGUAGE



LD LANGUAGE
The function is executed only if EN is **TRUE**.
ENO keeps the same value as EN.



IL LANGUAGE

```
Op1: LD   IN
     COS
     ST   Q     (* Q is: COS (IN) *)
```

SEE ALSO

SIN

TAN

ASIN

ACOS

ATAN

ATAN2

SIN / SINL

FUNCTION

Calculate a sine.

INPUTS

Name	Type	Description
IN	REAL/LREAL	Real value.

OUTPUTS

Name	Type	Description
Q	REAL/LREAL	Result: sine of IN.

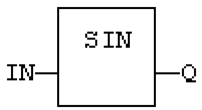
REMARKS

In LD language, the operation is executed only if the input rung (EN) is **TRUE**. The output rung (ENO) keeps the same value as the input rung. In IL, the input must be loaded in the current result before calling the function.

ST LANGUAGE

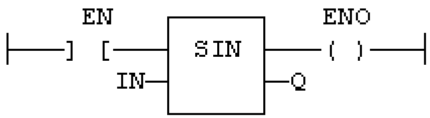
```
Q := SIN (IN);
```

FBD LANGUAGE



LD LANGUAGE

The function is executed only if EN is **TRUE**.
ENO keeps the same value as EN.



IL LANGUAGE

```
Op1: LD  IN
     SIN
     ST  Q    (* Q is: SIN (IN) *)
```

SEE ALSO

COS
TAN
ASIN
ACOS
ATAN
ATAN2

TAN / TANL

FUNCTION

Calculate a tangent.

INPUTS

Name	Type	Description
IN	REAL/LREAL	Real value.

OUTPUTS

Name	Type	Description
Q	REAL/LREAL	Result: tangent of IN.

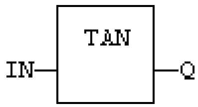
REMARKS

In LD language, the operation is executed only if the input rung (EN) is **TRUE**. The output rung (ENO) keeps the same value as the input rung. In IL, the input must be loaded in the current result before calling the function.

ST LANGUAGE

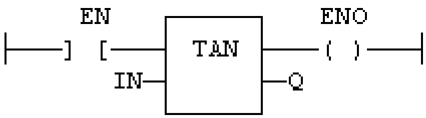
```
Q := TAN (IN);
```

FBD LANGUAGE



LD LANGUAGE

The function is executed only if EN is **TRUE**.
ENO keeps the same value as EN.



IL LANGUAGE

```
Op1: LD  IN
     TAN
     ST  Q    (* Q is: TAN (IN) *)
```

SEE ALSO

SIN
COS
ASIN
ACOS
ATAN
ATAN2

UseDegrees

FUNCTION

Sets the unit for angles in all trigonometric functions.

INPUTS

Name	Type	Description
IN	BOOL	If TRUE, turn all trigonometric functions to use degrees. If FALSE, turn all trigonometric functions to use radians (default).

OUTPUTS

Name	Type	Description
Q	BOOL	TRUE if functions use degrees before the call.

REMARKS

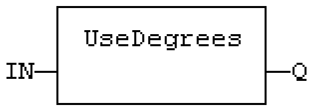
This function sets the working angular unit for the following functions:

Code	Function
SIN	sine
COS	cosine
TAN	tangent
ASIN	arc-sine
ACOS	arc-cosine
ATAN	arc-tangent
ATAN2	arc-tangent of Y / X

ST LANGUAGE

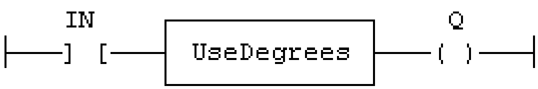
```
Q := UseDegrees (IN);
```

FBD LANGUAGE



LD LANGUAGE

Input is the rung. The rung is the output.



IL LANGUAGE

```
Op1: LD IN
UseDegrees
ST Q
```

String Operations

STANDARD OPERATORS AND FUNCTIONS THAT MANAGE CHARACTER STRINGS:

Code	Operator / Function
+	concatenation of strings
CONCAT	concatenation of strings

Code	Operator / Function
MLEN	get string length
DELETE	delete characters in a string
INSERT	insert characters in a string
FIND	find characters in a string
REPLACE	replace characters in a string
LEFT	extract a part of a string on the left
RIGHT	extract a part of a string on the right
MID	extract a part of a string
CHAR	build a single character string
ASCII	get the ASCII code of a character within a string
ATOH	converts a hexadecimal string to an integer
HTOA	converts an integer to a hexadecimal string
CRC16	CRC16 calculation
ArrayToString	copies elements of an SINT array to a STRING
StringToArray	copies characters of a STRING to an SINT array

OTHER FUNCTIONS AVAILABLE FOR MANAGING STRING TABLES AS RESOURCES:

Function	Description
StringTable	Select the active string table resource
LoadString	Load a string from the active string table

ArrayToString / ArrayToStringU

FUNCTION

Copy an array of **SINT** to a **STRING**.

INPUTS

Name	Type	Description
SRC	SINT	Source array of SINT small integers (USINT for ArrayToStringU).

Name	Type	Description
DST	STRING	Destination STRING.
COUNT	DINT	Numbers of characters to be copied.

OUTPUTS

Name	Type	Description
Q	DINT	Number of characters copied.

REMARKS

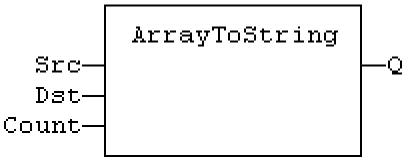
In LD language, the operation is executed only if the input rung (EN) is **TRUE**. The output rung (ENO) keeps the same value as the input rung.

This function copies the **COUNT** first elements of the SRC array to the characters of the DST string. The function checks the maximum size of the destination string and adjust the **COUNT** number if necessary.

ST LANGUAGE

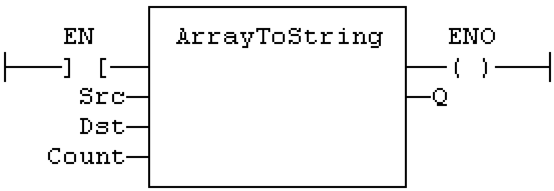
```
Q := ArrayToString (SRC, DST, COUNT);
```

FBD LANGUAGE



LD LANGUAGE

The function is executed only if EN is **TRUE**.
ENO keeps the same value as EN.



IL LANGUAGE

Not available.

SEE ALSO

StringToArray

ASCII

FUNCTION
Get the **ASCII** code of a character within a string.

INPUTS

Name	Type	Description
IN	STRING	Input string
POS	DINT	Position of the character within the string. (The first valid position is 1).

OUTPUTS

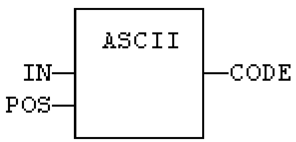
Name	Type	Description
CODE	DINT	ASCII code of the selected character, or 0 if position is invalid.

REMARKS
In LD language, the input rung (EN) enables the operation, and the output rung keeps the same value as the input rung. In IL language, the first parameter (IN) must be loaded in the current result before calling the function. The other input is the operand of the function.

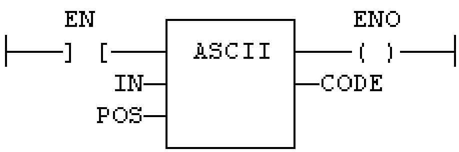
ST LANGUAGE

```
CODE := ASCII (IN, POS);
```

FBD LANGUAGE



LD LANGUAGE
The function is executed only if EN is **TRUE**.
ENO is equal to EN.



IL LANGUAGE

```
Op1: LD         IN
     AND_MASK  MSK
     ST         CODE
```

SEE ALSO
CHAR

ATOH

FUNCTION
Converts string to integer using hexadecimal basis.

INPUTS

Name	Type	Description
IN	STRING	String representing an integer in hexadecimal format.

OUTPUTS

Name	Type	Description
Q	DINT	Integer represented by the string.

TRUTH TABLE (EXAMPLES)

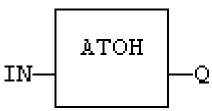
IN	Q
‘’	0
‘12’	18
‘a0’	160
A0zzz’	160

REMARKS
The function is case insensitive. The result is 0 for an empty string. The conversion stops before the first invalid character. In LD language, the operation is executed only if the input rung (EN) is **TRUE**. The output rung (ENO) keeps the same value as the input rung. In IL, the input must be loaded in the current result before calling the function.

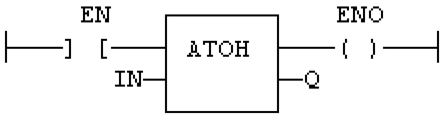
ST LANGUAGE

```
Q := ATOH (IN);
```

FBD LANGUAGE



LD LANGUAGE
The function is executed only if EN is **TRUE**.
ENO keeps the same value as EN.



IL LANGUAGE

```
Op1: LD   IN
     ATOH
     ST   Q
```

SEE ALSO

HTOA

CHAR

FUNCTION

Builds a single character string.

INPUTS

Name	Type	Description
CODE	DINT	ASCII code of the wished character.

OUTPUTS

Name	Type	Description
Q	STRING	STRING containing only the specified character.

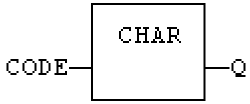
REMARKS

In LD language, the input rung (EN) enables the operation, and the output rung keeps the same value as the input rung. In IL language, the input parameter (**CODE**) must be loaded in the current result before calling the function.

ST LANGUAGE

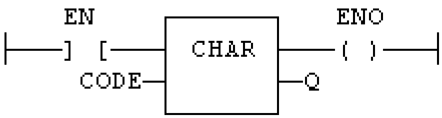
```
Q := CHAR (CODE);
```

FBD LANGUAGE



LD LANGUAGE

The function is executed only if EN is **TRUE**.
ENO is equal to EN.



IL LANGUAGE

```
Op1: LD    CODE
     CHAR
     ST    Q
```

SEE ALSO

ASCII

CONCAT

FUNCTION

Concatenate strings.

INPUTS

Name	Type	Description
IN _ 1	STRING	Any string variable or constant expression.
...		
IN_N	STRING	Any string variable or constant expression.

OUTPUTS

Name	Type	Description
Q	STRING	Concatenation of all inputs.

REMARKS

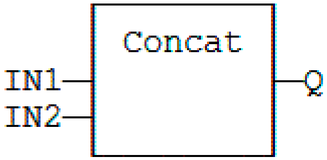
In FBD or LD language, the block may have up to 16 inputs. In IL or ST, the function accepts a variable number of inputs (at least 2).

Note that you also can use the “+” operator to concatenate strings.

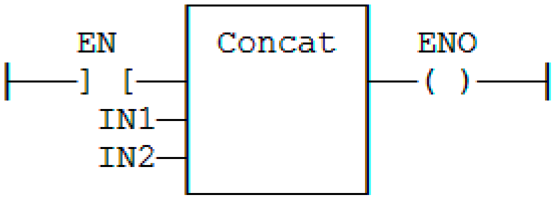
ST LANGUAGE

```
Q := CONCAT ('AB', 'CD', 'E');
(* now Q is 'ABCDE' *)
```

FBD LANGUAGE



LD LANGUAGE



IL LANGUAGE

```
Op1: LD      'AB'
     CONCAT 'CD', 'E'
     ST   Q    (* Q is now 'ABCDE' *)
```

CRC16

FUNCTION

Calculates a CRC16 on the characters of a string.

INPUTS

Name	Type	Description
IN	STRING	character string.

OUTPUTS

Name	Type	Description
Q	INT	CRC16 calculated on all the characters of the string.

REMARKS

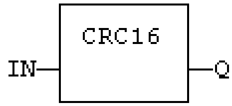
In LD language, the input rung (EN) enables the operation, and the output rung keeps the same value as the input rung. In IL language, the input parameter (IN) must be loaded in the current result before calling the function.

The function calculates a **MODBUS** CRC16, initialized at 16#**FFFF** value.

ST LANGUAGE

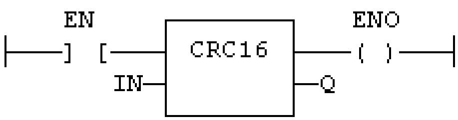
```
Q := CRC16 (IN);
```

FBD LANGUAGE



LD LANGUAGE

The function is executed only if EN is **TRUE**.
ENO is equal to EN.



IL LANGUAGE

```
Op1: LD    IN
     CRC16
     ST    Q
```

DELETE

FUNCTION

Delete characters in a string.

INPUTS

Name	Type	Description
IN	STRING	Character string.
NBC	DINT	Number of characters to be deleted.
POS	DINT	Position of the first deleted character (first character position is 1).

OUTPUTS

Name	Type	Description
Q	STRING	Modified string.

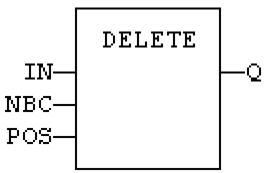
REMARKS

The first valid character position is 1. In LD language, the operation is executed only if the input rung (EN) is **TRUE**. The output rung (ENO) keeps the same value as the input rung. In IL, the first input (the string) must be loaded in the current result before calling the function. Other arguments are operands of the function, separated by comas.

ST LANGUAGE

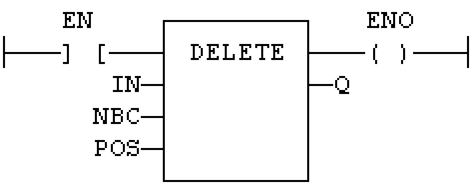
```
Q := DELETE (IN, NBC, POS);
```

FBD LANGUAGE



LD LANGUAGE

The function is executed only if EN is **TRUE**.
ENO keeps the same value as EN.



IL LANGUAGE

```
Op1: LD     IN
     DELETE NBC, POS
     ST     Q
```

SEE ALSO

+
MLEN
INSERT
FIND
REPLACE
LEFT

RIGHT
MID

FIND

FUNCTION
Find position of characters in a string.

INPUTS

Name	Type	Description
IN	STRING	Character string.
STR	STRING	String containing searched characters.

OUTPUTS

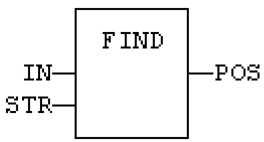
Name	Type	Description
POS	DINT	Position of the first character of STR in IN, or 0 if not found.

REMARKS
The first valid character position is 1. A return value of 0 means that the STR string has not been found. Search is case sensitive. In LD language, the operation is executed only if the input rung (EN) is **TRUE**. The output rung (ENO) keeps the same value as the input rung. In IL, the first input (the string) must be loaded in the current result before calling the function. The second argument is the operand of the function.

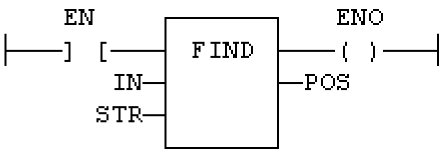
ST LANGUAGE

```
POS := FIND (IN, STR);
```

FBD LANGUAGE



LD LANGUAGE
The function is executed only if EN is **TRUE**.
ENO keeps the same value as EN.



IL LANGUAGE

```
Op1: LD     IN
     FIND   STR
     ST     POS
```

SEE ALSO

+
MLEN
DELETE
INSERT
REPLACE
LEFT
RIGHT
MID

HTOA

FUNCTION

Converts integer to string using hexadecimal basis.

INPUTS

Name	Type	Description
IN	DINT	Integer value.

OUTPUTS

Name	Type	Description
Q	STRING	String representing the integer in hexadecimal format.

TRUTH TABLE (EXAMPLES)

IN	Q
0	‘0’
18	‘12’
160	‘A0’

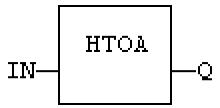
REMARKS

In LD language, the operation is executed only if the input rung (EN) is **TRUE**. The output rung (ENO) keeps the same value as the input rung. In IL, the input must be loaded in the current result before calling the function.

ST LANGUAGE

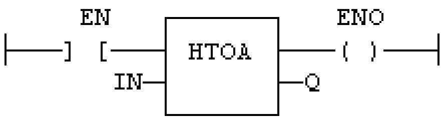
```
Q := HTOA (IN);
```

FBD LANGUAGE



LD LANGUAGE

The function is executed only if EN is **TRUE**.
ENO keeps the same value as EN.



IL LANGUAGE

```
Op1: LD   IN
     HTOA
     ST   Q
```

SEE ALSO

ATOH

INSERT

FUNCTION

Insert characters in a string.

INPUTS

Name	Type	Description
IN	STRING	Character string.
STR	STRING	String containing characters to be inserted.
POS	DINT	Position of the first inserted character (first character position is 1).

OUTPUTS

Name	Type	Description
Q	STRING	Modified string.

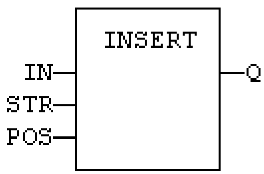
REMARKS

The first valid character position is 1. In LD language, the operation is executed only if the input rung (EN) is **TRUE**. The output rung (ENO) keeps the same value as the input rung. In IL, the first input (the string) must be loaded in the current result before calling the function. Other arguments are operands of the function, separated by comas.

ST LANGUAGE

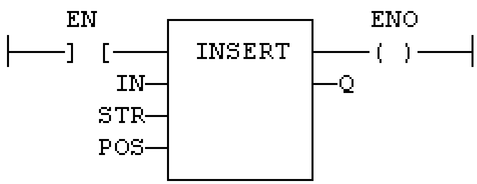
```
Q := INSERT (IN, STR, POS);
```

FBD LANGUAGE



LD LANGUAGE

The function is executed only if EN is **TRUE**.
ENO keeps the same value as EN.



IL LANGUAGE

```
Op1: LD     IN
     INSERT STR, POS
     ST     Q
```

SEE ALSO

+
MLEN
DELETE
FIND
REPLACE
LEFT

RIGHT
MID

LEFT

FUNCTION
Extract characters of a string on the left.

INPUTS

Name	Type	Description
IN	STRING	Character string.
NBC	DINT	Number of characters to extract.

OUTPUTS

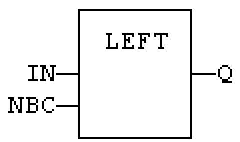
Name	Type	Description
Q	STRING	String containing the first NBC characters of IN.

REMARKS
In LD language, the operation is executed only if the input rung (EN) is **TRUE**. The output rung (ENO) keeps the same value as the input rung. In IL, the first input (the string) must be loaded in the current result before calling the function. The second argument is the operand of the function.

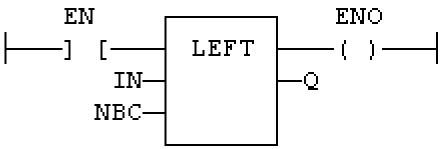
ST LANGUAGE

```
Q := LEFT (IN, NBC);
```

FBD LANGUAGE



LD LANGUAGE
The function is executed only if EN is **TRUE**.
ENO keeps the same value as EN.



IL LANGUAGE

```
    Op1: LD      IN
         LEFT    NBC
         ST      Q
```

SEE ALSO

+
MLEN
DELETE
INSERT
FIND
REPLACE
RIGHT
MID

LoadString

FUNCTION

Load a string from the active string table.

INPUTS

Name	Type	Description
ID	DINT	ID of the string as declared in the string table.

OUTPUTS

Name	Type	Description
Q	STRING	Loaded string or empty string in case of error.

REMARKS

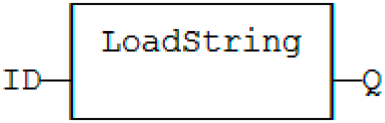
This function loads a string from the active string table and stores it into a **STRING** variable. The StringTable() (auf Seite 693) function is used for selecting the active string table.

The ID input (the string item identifier) is an identifier such as declared within the string table resource. You don’t need to “define” again this identifier. The system does it for you.

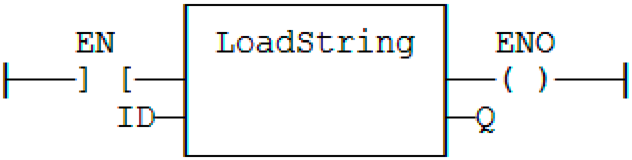
ST LANGUAGE

```
Q := LoadString (ID);
```

FBD LANGUAGE



LD LANGUAGE



IL LANGUAGE

```
Op1: LD         ID
     LoadString
     ST         Q
```

SEE ALSO

`StringTable`

`String Table`

MID

FUNCTION

Extract characters of a string at any position.

INPUTS

Name	Type	Description
IN	STRING	Character string.
NBC	DINT	Number of characters to extract.
POS	DINT	Position of the first character to extract (first character of IN is at position 1).

OUTPUTS

Name	Type	Description
Q	STRING	String containing the first NBC characters of IN.

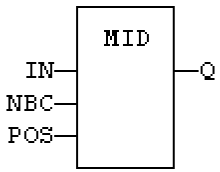
REMARKS

The first valid position is 1. In LD language, the operation is executed only if the input rung (EN) is **TRUE**. The output rung (ENO) keeps the same value as the input rung. In IL, the first input (the string) must be loaded in the current result before calling the function. Other argument are operands of the function, separated by comas.

ST LANGUAGE

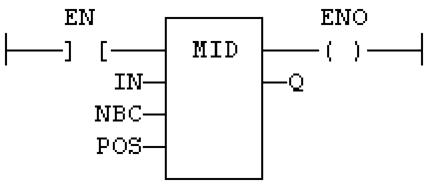
```
Q := MID (IN, NBC, POS);
```

FBD LANGUAGE



LD LANGUAGE

The function is executed only if EN is **TRUE**.
ENO keeps the same value as EN.



IL LANGUAGE

```
Op1: LD   IN
     MID  NBC, POS
     ST   Q
```

SEE ALSO

+
MLEN
DELETE
INSERT
FIND
REPLACE
LEFT
RIGHT

MLEN

FUNCTION

Get the number of characters in a string.

INPUTS

Name	Type	Description
IN	STRING	Character string.

OUTPUTS

Name	Type	Description
NBC	DINT	Number of characters currently in the string. 0 if string is empty.

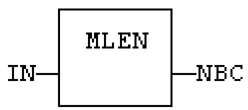
REMARKS

In LD language, the operation is executed only if the input rung (EN) is **TRUE**. The output rung (ENO) keeps the same value as the input rung. In IL, the input must be loaded in the current result before calling the function.

ST LANGUAGE

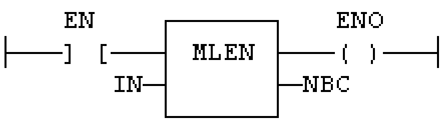
```
NBC := MLEN (IN);
```

FBD LANGUAGE



LD LANGUAGE

The function is executed only if EN is **TRUE**.
ENO keeps the same value as EN.



IL LANGUAGE

```
Op1: LD   IN
     MLEN
     ST   NBC
```

SEE ALSO

+

DELETE
INSERT
FIND
REPLACE
LEFT
RIGHT
MID

REPLACE

FUNCTION
Replace characters in a string.

INPUTS

Name	Type	Description
IN	STRING	Character string.
STR	STRING	String containing the characters to be inserted in place of NDEL removed characters.
NDEL	DINT	Number of characters to be deleted before insertion of STR.
POS	DINT	Position where characters are replaced (first character position is 1).

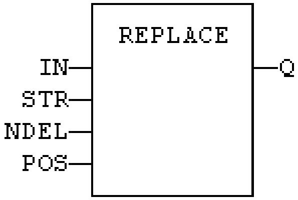
OUTPUTS

Name	Type	Description
Q	STRING	Modified string.

REMARKS
The first valid character position is 1. In LD language, the operation is executed only if the input rung (EN) is **TRUE**. The output rung (ENO) keeps the same value as the input rung. In IL, the first input (the string) must be loaded in the current result before calling the function. Other arguments are operands of the function, separated by comas.

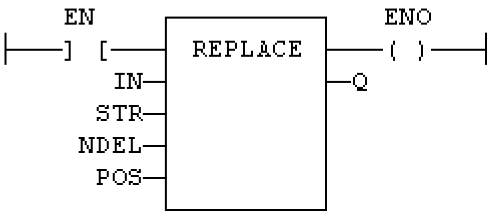
ST LANGUAGE
```
Q := REPLACE (IN, STR, NDEL, POS);
```

FBD LANGUAGE



LD LANGUAGE

The function is executed only if EN is **TRUE**.
ENO keeps the same value as EN.



IL LANGUAGE

```
Op1: LD      IN
     REPLACE STR, NDEL, POS
     ST      Q
```

SEE ALSO

+
MLEN
DELETE
INSERT
FIND
LEFT
RIGHT
MID

RIGHT

FUNCTION

Extract characters of a string on the right.

INPUTS

Name	Type	Description
IN	STRING	Character string.
NBC	DINT	Number of characters to extract.

OUTPUTS

Name	Type	Description
Q	STRING	String containing the last NBC characters of IN.

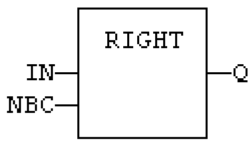
REMARKS

In LD language, the operation is executed only if the input rung (EN) is **TRUE**. The output rung (ENO) keeps the same value as the input rung. In IL, the first input (the string) must be loaded in the current result before calling the function. The second argument is the operand of the function.

ST LANGUAGE

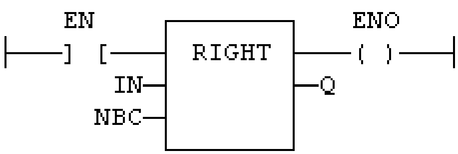
```
Q := RIGHT (IN, NBC);
```

FBD LANGUAGE



LD LANGUAGE

The function is executed only if EN is **TRUE**.
ENO keeps the same value as EN.



IL LANGUAGE

```
Op1: LD      IN
     RIGHT   NBC
     ST      Q
```

SEE ALSO

+
MLEN
DELETE
INSERT

FIND
REPLACE
LEFT
MID

StringTable

FUNCTION
Selects the active string table.

INPUTS

Name	Type	Description
TABLE	STRING	Name of the String Table resource - must be a constant.
COL	STRING	Name of the column in the table - must be a constant.

OUTPUTS

Name	Type	Description
OK	BOOL	TRUE if OK.

REMARKS
This function selects a column of a valid String Table resource to become the active string table. The LoadString() (auf Seite 686) function always refers to the active string table.

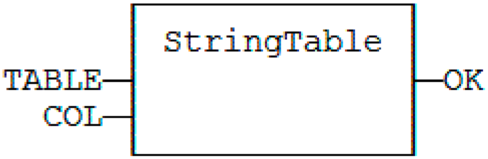
Arguments must be constant string expressions and must fit to a declared string table and a valid column name within this table.

If you have only one string table with only one column defined in your project, you do not need to call this function as it will be the default string table anyway.

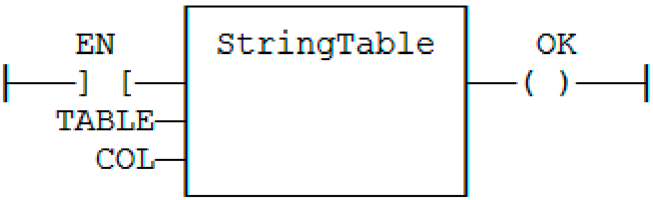
ST LANGUAGE

```
OK := StringTable (‘MyTable’, ‘FirstColumn”);
```

FBD LANGUAGE



LD LANGUAGE



IL LANGUAGE

```
    Op1: LD          'MyTable'
         StringTable 'First Column'
         ST          OK
```

SEE ALSO

`LoadString`
`String Table`

StringToArray / StringToArrayU

FUNCTION

Copies the characters of a **`STRING`** to an array of **`SINT`**.

INPUTS

Name	Type	Description
SRC	STRING	Source STRING.
DST	SINT	Destination array of SINT small integers (USINT for StringToArrayU).

OUTPUTS

Name	Type	Description
Q	DINT	Number of characters copied.

REMARKS

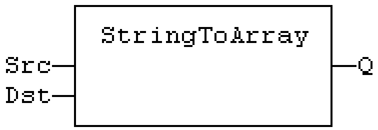
In LD language, the operation is executed only if the input rung (EN) is **TRUE**. The output rung (ENO) keeps the same value as the input rung. In IL, the input must be loaded in the current result before calling the function.

This function copies the characters of the SRC string to the first characters of the DST array. The function checks the maximum size destination arrays and reduces the number of copied characters if necessary.

ST LANGUAGE

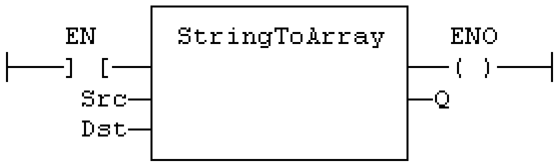
```
Q := StringToArray (SRC, DST);
```

FBD LANGUAGE



LD LANGUAGE

The function is executed only if EN is **TRUE**.
ENO keeps the same value as EN.



IL LANGUAGE

```
Op1: LD             SRC
     StringToArray  DST
     ST             Q
```

SEE ALSO

ArrayToString

Advanced Operations

Below are the standard blocks that perform advanced operations.

ANALOG SIGNAL PROCESSING

Block	Description
Average	Calculate the average of signal samples.
Integral	Calculate the integral of a signal.

Block	Description
Derivate	Derive a signal.
PID	**PID** loop.
Ramp	Ramp signal.
Lim_Alrm	Low / High level detection.
Hyster	Hysteresis calculation.
SigPlay	Play an analog signal from a resource.
SigScale	Get a point from a signal resource.
CurveLin	Linear interpolation on a curve.
SurfLin	Linear interpolation on a surface.

ALARM MANAGEMENT

Block	**Description**
Lim_Alrm	Low / High level detection.
Alarm_M	Alarm with manual reset.
Alarm_A	Alarm with automatic reset.

DATA COLLECTIONS AND SERIALIZATION

Block	**Description**
StackInt	Stack of integers.
FIFO	"First in / first out" list.
LIFO	"Last in / first out" stack.
SerializeIn	Extract data from a binary frame.
SerializeOut	Write data to a binary frame.
SerGetString	Extract a string from a binary frame.
SerPutString	Copies a string to a binary frame.

DATA LOGGING

Block	**Description**
LogFileCSV	Log values of variables to a CSV file.

SPECIAL OPERATIONS

Block	Description
GetSysInfo	Get system information.
Printf	Trace messages.

Block	Description
CycleStop	Sets the application in cycle stepping mode.
FatalStop	Breaks the cycle and stop with fatal error.
EnableEvents	Enable / disable produced events for binding.
ApplyRecipeColumn	Apply the values of a column from a recipe file.
VLID	Get the ID of an embedded list of variables.
SigID	Get the ID of a signal resource.

COMMUNICATION

SERIO: serial communication

AS-interface

TCP-IP management functions

UDP management functions

MQTT protocol handling

MBSlaveRTU

MBSlaveUDP

MBMasterRTU

MBMasterTCP

CanRcvMsg

CanSndMsg

CANopen functions

DNP3 Master function blocks

OTHERS

File management functions

Dynamic memory allocation functions

Real Time Clock

Variable size text buffer manipulation

XML writing and parsing

T5 Registry Management functions

ALARM_A

FUNCTION BLOCK

Alarm with automatic reset.

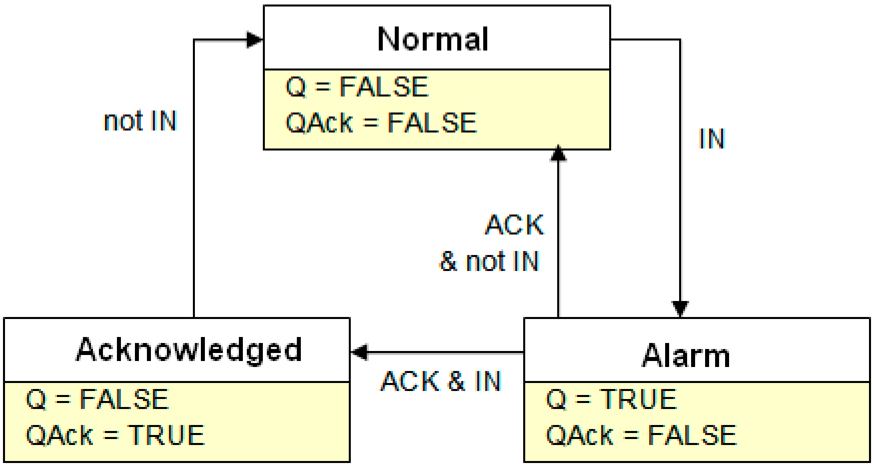
INPUTS

Name	Type	Description
IN	BOOL	Process signal.
ACK	BOOL	Acknowledge command.

OUTPUTS

Name	Type	Description
Q	BOOL	TRUE if alarm is active.
QACK	BOOL	TRUE if alarm is acknowledged.

SEQUENCE



REMARKS

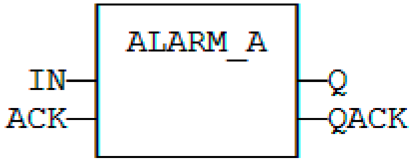
Combine this block with the **LIM_ALRM** block for managing analog alarms.

ST LANGUAGE

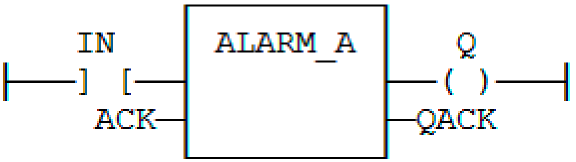
MyALARM is declared as an instance of **ALARM_A** function block.

```
MyALARM (IN, ACK, RST);
Q := MyALARM.Q;
QACK := MyALARM.QACK;
```

FBD LANGUAGE



LD LANGUAGE



IL LANGUAGE

MyALARM is declared as an instance of **ALARM_A** function block.

```
Op1: CAL MyALARM (IN, ACK, RST)
     LD  MyALARM.Q
     ST  Q
     LD  MyALARM.QACK
     ST  QACK
```

SEE ALSO

ALARM_M

LIM_ALRM

ALARM_M

FUNCTION BLOCK

Alarm with manual reset.

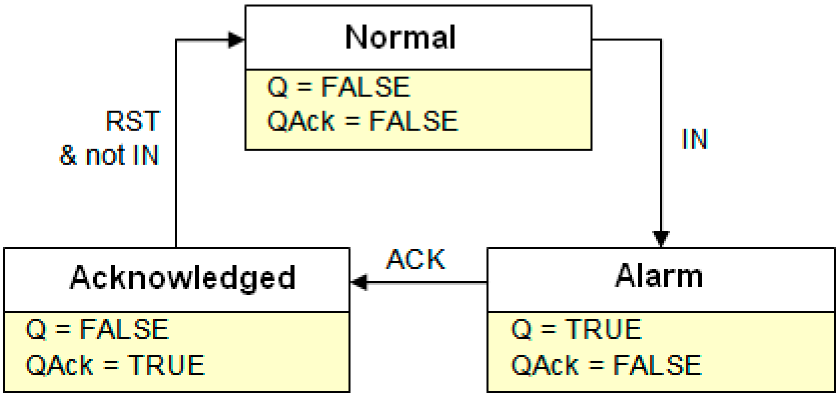
INPUTS

Name	Type	Description
IN	BOOL	Process signal.
ACK	BOOL	Acknowledge command.
RST	BOOL	Reset command.

OUTPUTS

Name	Type	Description
Q	BOOL	TRUE if alarm is active.
QACK	BOOL	TRUE if alarm is acknowledged.

SEQUENCE



REMARKS

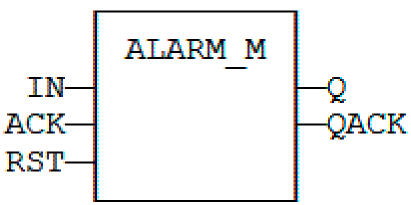
Combine this block with the **LIM_ALRM** (auf Seite 738) block for managing analog alarms.

ST LANGUAGE

MyALARM is declared as an instance of **ALARM_M** function block.

```
MyALARM (IN, ACK, RST);
Q := MyALARM.Q;
QACK := MyALARM.QACK;
```

FBD LANGUAGE



LD LANGUAGE

IL LANGUAGE

MyALARM is declared as an instance of **ALARM_M** function block.

```
Op1: CAL MyALARM (IN, ACK, RST)
     LD  MyALARM.Q
     ST  Q
     LD  MyALARM.QACK
     ST  QACK
```

SEE ALSO

ALARM_A

LIM_ALRM

ApplyRecipeColumn

FUNCTION

Apply the values of a column from a recipe file.

INPUTS

Name	Type	Description
FILE	STRING	Pathname of the recipe file (.RCP or .CSV) - must be a constant value!
COL	DINT	Index of the column in the recipe (0 based).

OUTPUTS

Name	Type	Description
OK	BOOL	TRUE if OK - FALSE if parameters are invalid.

REMARKS

The ‘**FILE**’ input is a constant string expression specifying the path name of a valid .RCP or .CSV file. If no path is specified, the file is assumed to be located in the project folder. RCP files are created using the recipe editor. CSV files can be created using **EXCEL** or **NOTEPAD**.

In CSV files, the first line must contain column headers, and is ignored during compiling. There is one variable per line. The first column contains the symbol of the variable. Other columns are values.

If a cell is empty, it is assumed to be the same value as the previous (left side) cell. If it is the first cell of a raw, it is assumed to be null (0 or **FALSE** or empty string).

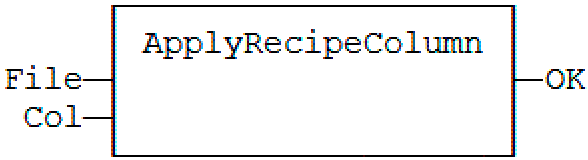
In LD language, the operation is executed only if the input rung (EN) is **TRUE**. The output rung is the result of the function.

RECIPE FILES ARE READ AT COMPILING TIME AND ARE EMBEDDED INTO THE DOWNLOADED APPLICATION CODE. THIS IMPLIES THAT A MODIFICATION PERFORMED IN THE RECIPE FILE AFTER DOWNLOADING WILL NOT BE TAKEN INTO ACCOUNT BY THE APPLICATION.

ST LANGUAGE

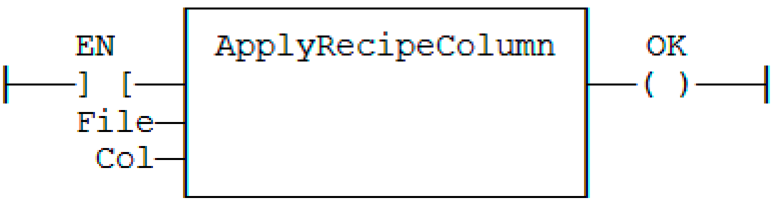
```
OK := ApplyRecipeColumn ('MyFile.rcp', COL);
```

FBD LANGUAGE



LD LANGUAGE

The function is executed only if EN is **TRUE**.



IL LANGUAGE

```
Op1: LD                'MyFile.rcp'
     ApplyRecipeColumn COL
     ST                OK
```

AVERAGE / AVERAGEL

FUNCTION BLOCK

Calculates the average of signal samples.

INPUTS

Name	Type	Description
RUN	BOOL	Enabling command.
XIN	REAL	Input signal (*).
N	DINT	Number of samples stored for average calculation - Cannot exceed 128.

OUTPUTS

Name	Type	Description
XOUT	REAL	Average of the stored samples (*).

(*) **AVERAGEL** has **LREAL** arguments.

REMARKS

The average is calculated according to the number of stored samples, that can be less that N when the block is enabled. In LD language, the input rung is the RUN command. The output rung keeps the state of the input rung.

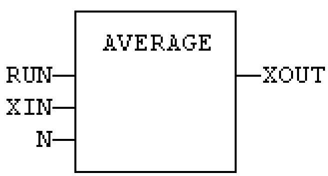
The “N” input is take into account only when the RUN input is **FALSE**. So the “RUN” needs to be reset after a change.

ST LANGUAGE

MyAve is a declared instance of **AVERAGE** function block.

```
MyAve (RUN, XIN, N);
XOUT := MyAve.XOUT;
```

FBD LANGUAGE



LD LANGUAGE

ENO has the same state as RUN.

IL LANGUAGE

MyAve is a declared instance of **AVERAGE** function block.

```
Op1: CAL MyAve (RUN, XIN, N)
     LD  MyAve.XOUT
     ST  XOUT
```

SEE ALSO

INTEGRAL

DERIVATE

LIM_ALRM

HYSTER

STACKINT

CurveLin

FUNCTION BLOCK

Linear interpolation on a curve.

INPUTS

Name	Type	Description
X	REAL	X coordinate of the point to be interpolated.
XAxis	REAL[]	X coordinates of the known points of the X axis.
YVal	REAL[]	Y coordinate of the points defined on the X axis.

OUTPUTS

Name	Type	Description
Y	REAL	Interpolated Y value corresponding to the X input
OK	BOOL	TRUE if successful.
ERR	DINT	Error code if failed - 0 if OK.

REMARKS

This function performs linear interpolation in between a list of points defined in the XAxis single dimension array. The output Y value is an interpolation of the Y values of the two rounding points defined in the X axis. Y values of defined points are passed in the YVal single dimension array.

Values in XAxis must be sorted from the smallest to the biggest. There must be at least two points defined in the X axis. YVal and XAxis input arrays must have the same dimension.

In case the X input is less than the smallest defined X point, the Y output takes the first value defined in YVal and an error is reported. In case the X input is greater than the biggest defined X point, the Y output takes the last value defined in YVal and an error is reported.

The ERR output gives the cause of the error if the function fails:

Error code	Meaning
0	OK
1	Invalid dimension of input arrays
2	Invalid points for the X axis
4	X is out of the defined X axis

CycleStop

FUNCTION

Sets the application in cycle stepping mode.

INPUTS

Name	Type	Description
IN	BOOL	Condition.

OUTPUTS

Name	Type	Description
Q	BOOL	**TRUE** if performed.

REMARKS

This function turns the Virtual Machine in Cycle Stepping mode. Restarting normal execution will be performed using the debugger.

The VM is set in **cycle stepping** mode only if the **IN** argument is **TRUE**.

The current main program and all possibly called sub-programs or UDFBs are normally performed up the end. Other programs of the cycle are not executed.

DERIVATE

FUNCTION BLOCK

Derivates a signal.

INPUTS

Name	Type	Description
RUN	BOOL	Run command: **TRUE**=derivate / **FALSE**=hold.

Name	Type	Description
XIN	REAL	Input signal.
CYCLE	TIME	Sampling period (should not be less than the target cycle timing).

OUTPUTS

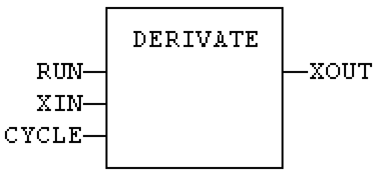
Name	Type	Description
XOUT	REAL	Output signal.

REMARKS
In LD language, the input rung is the RUN command. The output rung keeps the state of the input rung.

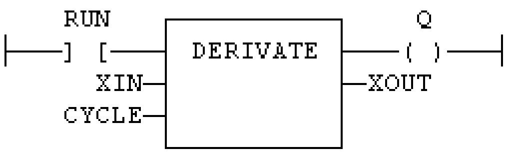
ST LANGUAGE
MyDerv is a declared instance of **DERIVATE** function block.

```
MyDerv (RUN, XIN, CYCLE);
XOUT := MyDerv.XOUT;
```

FBD LANGUAGE



LD LANGUAGE
ENO has the same state as RUN.



IL LANGUAGE
MyDerv is a declared instance of **DERIVATE** function block.

```
Op1: CAL MyDerv (RUN, XIN, CYCLE)
     LD  MyDerv.XOUT
     ST  XOUT
```

SEE ALSO
AVERAGE

INTEGRAL

LIM_ALRM

HYSTER

STACKINT

EnableEvents

FUNCTION

Enable or disable the production of events for binding (runtime to runtime variable exchange).

INPUTS

Name	Type	Description
EN	BOOL	**TRUE** to enable events / **FALSE** to disable events.

OUTPUTS

Name	Type	Description
ENO	BOOL	Echo of EN input.

REMARKS

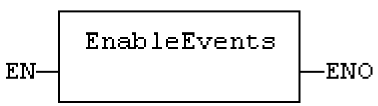
Production is enabled when the application starts. The first production will be operated after the first cycle. So to disable events since the beginning, you must call EnableEvents (**FALSE**) in the very first cycle.

In LD language, the input rung (EN) enables the event production, and the output rung keeps the state of the input rung. In IL language, the input must be loaded before the function call.

ST LANGUAGE

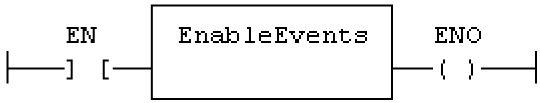
```
ENO := EnableEvents (EN);
```

FBD LANGUAGE



LD LANGUAGE

Events are enables if EN is **TRUE**.
ENO has the same value as EN.



IL LANGUAGE

```
Op1: LD  EN
     EnableEvents
     ST  ENO
```

FatalStop

FUNCTION

Breaks the application in fatal error.

INPUTS

Name	Type	Description
IN	BOOL	Condition.

OUTPUTS

Name	Type	Description
Q	BOOL	TRUE if performed.

REMARKS

This function breaks the current cycle and sets the Virtual Machine in **ERROR** mode. Restarting normal execution will be performed using the debugger.

The VM is stopped only if the **IN** argument is **TRUE**. The end of the current cycle is then not performed.

FIFO

FUNCTION BLOCK

Manages a first in / first out list.

INPUTS

Name	Type	Description
PUSH	BOOL	Push a new value (on rising edge).
POP	BOOL	Pop a new value (on rising edge).

Name	Type	Description
RST	BOOL	Reset the list.
NEXTIN	ANY	Value to be pushed.
NEXTOUT	ANY	Value of the oldest pushed value - updated after call!
BUFFER	ANY	Array for storing values.

OUTPUTS

Name	Type	Description
EMPTY	BOOL	**TRUE** if the list is empty.
OFLO	BOOL	**TRUE** if overflow on a **PUSH** command.
COUNT	DINT	Number of values in the list.
PREAD	DINT	Index in the buffer of the oldest pushed value.
PWRITE	DINT	Index in the buffer of the next push position.

REMARKS

NEXTIN, **NEXTOUT** and **BUFFER** must have the same data type and cannot be **STRING**.

The **NEXTOUT** argument specifies a variable that is filled with the oldest push value after the block is called.

Values are stored in the **BUFFER** array. Data is arranged as a roll over buffer and is never shifted or reset. Only read and write pointers and pushed values are updated. The maximum size of the list is the dimension of the array.

The first time the block is called, it remembers on which array it should work. If you call later the same instance with another **BUFFER** input, the call is considered as invalid and makes nothing. Outputs reports an empty list in this case.

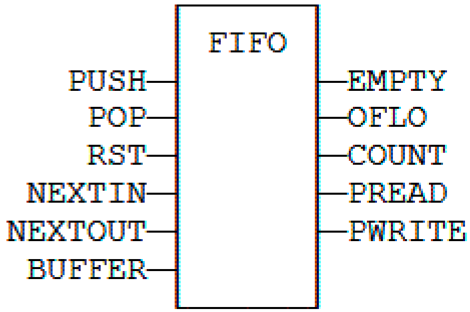
In LD language, input rung is the **PUSH** input. The output rung is the **EMPTY** output.

ST LANGUAGE

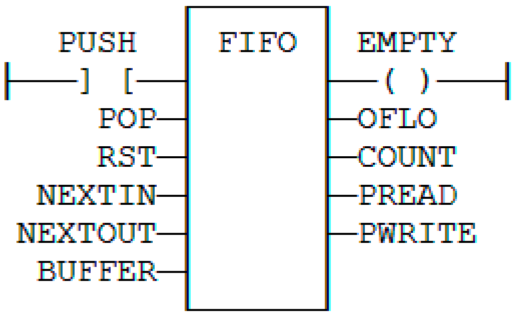
MyFIFO is a declared instance of **FIFO** function block.

```
SMyFIFO (PUSH, POP, RST, NEXTIN, NEXTOUT, BUFFER);
EMPTY := MyFIFO.EMPTY;
OFLO := MyFIFO.OFLO;
COUNT := MyFIFO.COUNT;
PREAD := MyFIFO.PREAD;
PWRITE := MyFIFO.PWRITE;
```

FBD LANGUAGE



LD LANGUAGE



IL LANGUAGE

MyFIFO is a declared instance of **FIFO** function block.

```
Op1: CAL MyFIFO (PUSH, POP, RST, NEXTIN, NEXTOUT, BUFFER)
     LD  MyFIFO.EMPTY
     ST  EMPTY
     LD  MyFIFO.OFLO
     ST  OFLO
     LD  MyFIFO.COUNT
     ST  COUNT
     LD  MyFIFO.PREAD
     ST  PREAD
     LD  MyFIFO.PWRITE
     ST  PWRITE
```

SEE ALSO

LIFO

GETSYSINFO

FUNCTION

Returns system information.

INPUTS

Name	Type	Description
INFO	DINT	Identifier of the requested information.

OUTPUTS

Name	Type	Description
Q	DINT	Value of the requested information or 0 if error.

REMARKS

The **INFO** parameter can be one of the following predefined values:

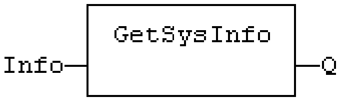
Value	Description
_SYSINFO_TRIGGER_MICROS	Programmed cycle time in micro-seconds.
_SYSINFO_TRIGGER_MS	Programmed cycle time in milliseconds.
_SYSINFO_CYCLETIME_MICROS	Duration of the previous cycle in micro-seconds.
_SYSINFO_CYCLETIME_MS	Duration of the previous cycle in milliseconds.
_SYSINFO_CYCLEMAX_MICROS	Maximum detected cycle time in micro-seconds.
_SYSINFO_CYCLEMAX_MS	Maximum detected cycle time in milliseconds.
_SYSINFO_CYCLESTAMP_MS	Time stamp of the current cycle in milliseconds (OEM dependent).
_SYSINFO_CYCLEOVERFLOWS	Number of detected cycle time overflows.
_SYSINFO_CYCLECOUNT	Counter of cycles.
_SYSINFO_APPVERSION	Version number of the application.
_SYSINFO_APPSTAMP	Compiling date stamp of the application.
_SYSINFO_CODECRC	CRC of the application code.
_SYSINFO_DATACRC	CRC of the application symbols.
_SYSINFO_FREEHEAP	Available space in memory heap (bytes)
_SYSINFO_DBSIZE	Space used in RAM (bytes)
_SYSINFO_ELAPSED	Seconds elapsed since startup

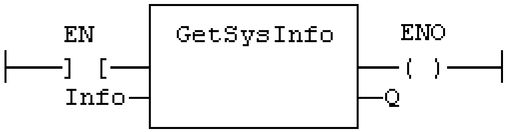
In LD language, the operation is executed only if the input rung (EN) is **TRUE**. The output rung (ENO) keeps the same value as the input rung. In IL, the input must be loaded in the current result before calling the function.

ST LANGUAGE

```
Q := GETSYSINFO (INFO);
```

FBD LANGUAGE





IL LANGUAGE

```
Op1: LD  INFO
     GETSYSINFO
     ST  Q
```

HYSTER

FUNCTION BLOCK

Hysteresis detection.

INPUTS

Name	Type	Description
XIN1	REAL	First input signal.
XIN2	REAL	Second input signal.
EPS	REAL	Hysteresis.

OUTPUTS

Name	Type	Description
Q	BOOL	Detected hysteresis: TRUE if XIN1 becomes greater than XIN2+EPS and is not yet below XIN2-EPS.

REMARKS

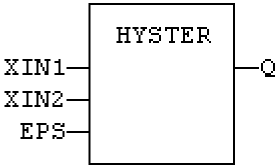
The hysteresis is detected on the difference of XIN1 and XIN2 signals. In LD language, the input rung (EN) is used for enabling the block. The output rung is the Q output.

ST LANGUAGE

MyHyst is a declared instance of **HYSTER** function block.

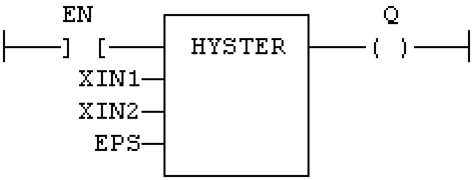
```
MyHyst (XIN1, XIN2, EPS);
Q := MyHyst.Q;
```

FBD LANGUAGE



LD LANGUAGE

The block is not called if EN is **FALSE**.



IL LANGUAGE

MyHyst is a declared instance of **HYSTER** function block.

```
Op1: CAL MyHyst (XIN1, XIN2, EPS)
     LD  MyHyst.Q
     ST  Q
```

SEE ALSO

AVERAGE

INTEGRAL

DERIVATE

LIM_ALRM

STACKINT

INTEGRAL

FUNCTION BLOCK

Calculates the integral of a signal.

INPUTS

Name	Type	Description
RUN	BOOL	Run command: TRUE=integrate / FALSE=hold.
R1	BOOL	Overriding reset.
XIN	REAL	Input signal.
X0	REAL	Initial value.
CYCLE	TIME	Sampling period (should not be less than the target cycle timing).

OUTPUTS

Name	Type	Description
Q	DINT	Running mode report: NOT (R1).
XOUT	REAL	Output signal.

REMARKS

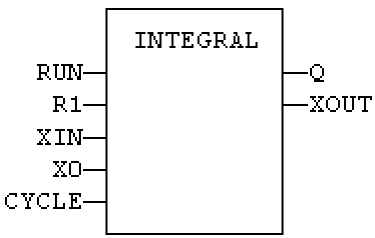
In LD language, the input rung is the RUN command. The output rung is the Q report status.

ST LANGUAGE

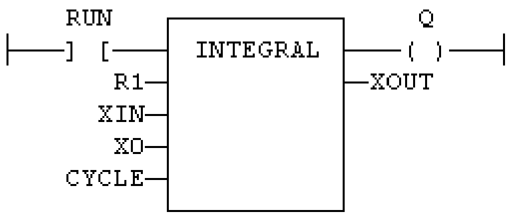
MyIntg is a declared instance of **INTEGRAL** function block.

```
MyIntg (RUN, R1, XIN, X0, CYCLE);
Q := MyIntg.Q;
XOUT := MyIntg.XOUT;
```

FBD LANGUAGE



LD LANGUAGE



IL LANGUAGE

MyIntg is a declared instance of **INTEGRAL** function block.

```
Op1: CAL MyIntg (RUN, R1, XIN, X0, CYCLE)
     LD  MyIntg.Q
     ST  Q
     LD  MyIntg.XOUT
     ST  XOUT
```

SEE ALSO

AVERAGE

DERIVATE

LIM_ALRM

HYSTER

STACKINT

LIFO

FUNCTION BLOCK

Manages a last in / first out stack.

INPUTS

Name	Type	Description
PUSH	BOOL	Push a new value (on rising edge).
POP	BOOL	Pop a new value (on rising edge).
RST	BOOL	Reset the list.
NEXTIN	ANY	Value to be pushed.
NEXTOUT	ANY	Value at the top of the stack - updated after call!
BUFFER	ANY	Array for storing values.

OUTPUTS

Name	Type	Description
EMPTY	BOOL	TRUE if the stack is empty.
OFLO	BOOL	TRUE if overflow on a PUSH command.
COUNT	DINT	Number of values in the stack.
PREAD	DINT	Index in the buffer of the top of the stack.
PWRITE	DINT	Index in the buffer of the next push position.

REMARKS

NEXTIN, **NEXTOUT** and **BUFFER** must have the same data type and cannot be **STRING**.

The **NEXTOUT** argument specifies a variable that is filled with the value at the top of the stack after the block is called.

Values are stored in the **BUFFER** array. Data is never shifted or reset. Only read and write pointers and pushed values are updated. The maximum size of the stack is the dimension of the array.

The first time the block is called, it remembers on which array it should work. If you call later the same instance with another **BUFFER** input, the call is considered as invalid and makes nothing. Outputs reports an empty stack in this case.

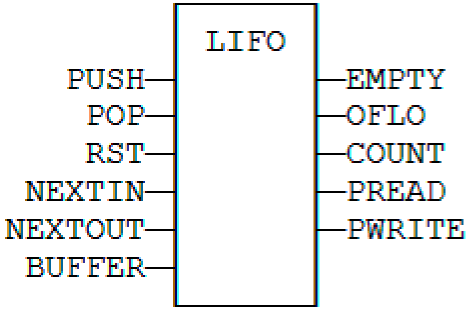
In LD language, input rung is the **PUSH** input. The output rung is the **EMPTY** output.

ST LANGUAGE

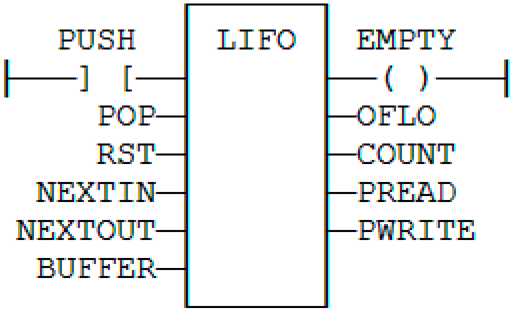
MyLIFO is a declared instance of **LIFO** function block.

```
MyLIFO (PUSH, POP, RST, NEXTIN, NEXTOUT, BUFFER);
EMPTY := MyLIFO.EMPTY;
OFLO := MyLIFO.OFLO;
COUNT := MyLIFO.COUNT;
PREAD := MyLIFO.PREAD;
PWRITE := MyLIFO.PWRITE;
```

FBD LANGUAGE



LD LANGUAGE



IL LANGUAGE

MyLIFO is a declared instance of **LIFO** function block.

```
Op1: CAL MyLIFO (PUSH, POP, RST, NEXTIN, NEXTOUT, BUFFER)
     LD  MyLIFO.EMPTY
     ST  EMPTY
     LD  MyLIFO.OFLO
     ST  OFLO
     LD  MyLIFO.COUNT
     ST  COUNT
     LD  MyLIFO.PREAD
     ST  PREAD
     LD  MyLIFO.PWRITE
     ST  PWRITE
```

SEE ALSO

FIFO

LIM_ALRM

FUNCTION BLOCK

Detects High and Low limits of a signal with hysteresis.

INPUTS

Name	Type	Description
H	REAL	Value of the High limit.
X	REAL	Input signal.
L	REAL	Value of the Low limit.
EPS	REAL	Value of the hysteresis.

OUTPUTS

Name	Type	Description
QH	BOOL	TRUE if the signal exceeds the High limit.
Q	BOOL	TRUE if the signal exceeds one of the limits (equals to QH OR QL).
QL	BOOL	TRUE if the signal exceeds the Low limit.

REMARKS

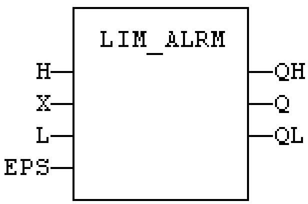
In LD language, the input rung (EN) is used for enabling the block. The output rung is the QH output.

ST LANGUAGE

MyAlarm is a declared instance of **LIM_ALRM** function block.

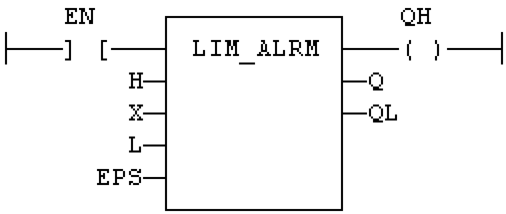
```
MyAlarm (H, X, L, EPS);
QH := MyAlarm.QH;
Q := MyAlarm.Q;
QL := MyAlarm.QL;
```

FBD LANGUAGE



LD LANGUAGE

The block is not called if EN is **FALSE**.



IL LANGUAGE

MyAlarm is a declared instance of **LIM_ALRM** function block.

```
Op1: CAL MyAlarm (H, X, L, EPS)
     LD  MyAlarm.QH
     ST  QH
     LD  MyAlarm.Q
     ST  Q
```

```
    LD  MyAlarm.QL
    ST  QL
```

SEE ALSO

ALARM_A

ALARM_M

PID

FUNCTION BLOCK

PID loop.

INPUTS

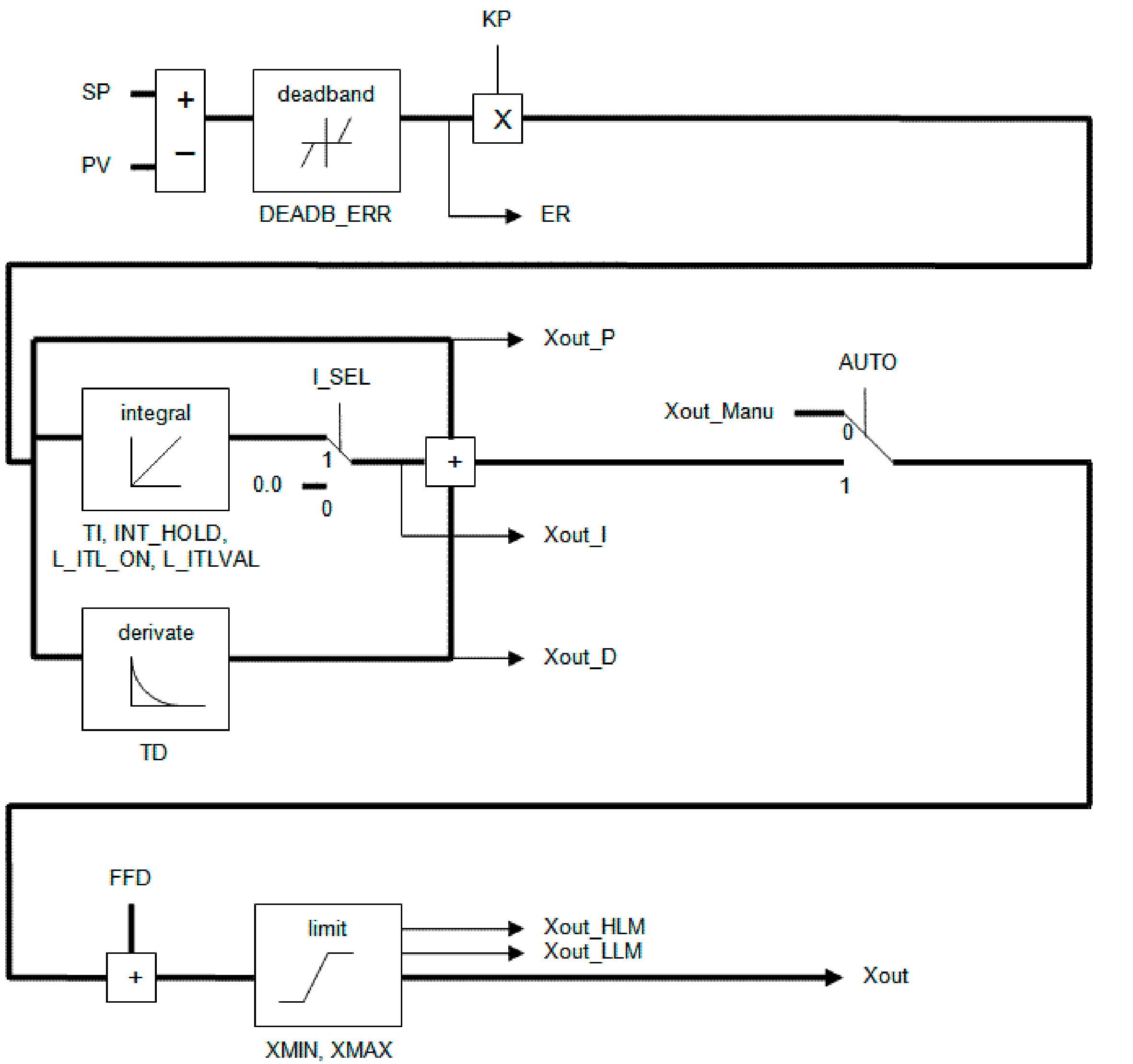
Name	Type	Description
AUTO	BOOL	TRUE = normal mode - FALSE = manual mode.
PV	REAL	Process value.
SP	REAL	Set point.
Xout _ Manu	REAL	Output value in manual mode.
KP	REAL	Gain.
TI	REAL	Integration time.
TD	REAL	Derivation time.
TS	TIME	Sampling period.
XMIN	REAL	Minimum allowed output value.
XMAX	REAL	Maximum output value.
I_SEL	BOOL	If FALSE, the integrated value is ignored.
INT_HOLD	BOOL	If TRUE, the integrated value is frozen.
I_ITL_ON	BOOL	If TRUE, the integrated value is reset to I_ITLVAL.
I_ITLVAL	REAL	Reset value for integration when I_ITL_ON is TRUE.
DEADB_ERR	REAL	Hysteresis on PV. PV will be considered as unchanged if greater than (PVprev - DEADBAND_W) and less that (PRprev + DEADBAND_W).
FFD	REAL	Disturbance value on output.

OUTPUTS

Name	Type	Description
Xout	REAL	Output command value.
ER	REAL	Last calculated error.

Name	Type	Description
Xout _ P	REAL	Last calculated proportional value.
Xout _ I	REAL	Last calculated integrated value.
Xout _ D	REAL	Last calculated derivated value.
Xout _ HLM	BOOL	TRUE if the output valie is saturated to XMIN.

Name	Type	Description
Xout _ LLM	BOOL	TRUE if the output value is saturated to XMAX.



REMARKS

It is important for the stability of the control that the TS sampling period is much bigger than the cycle time.

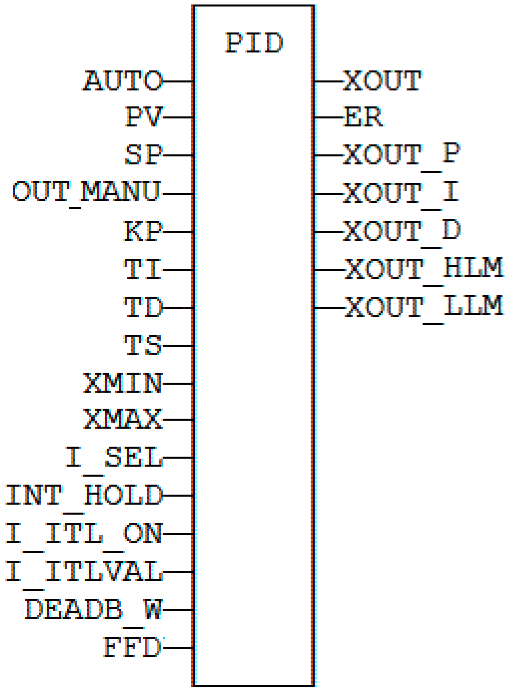
In LD language, the output rung has the same value as the **AUTO** input, corresponding to the input rung.

ST LANGUAGE

MyPID is a declared instance of PID function block.

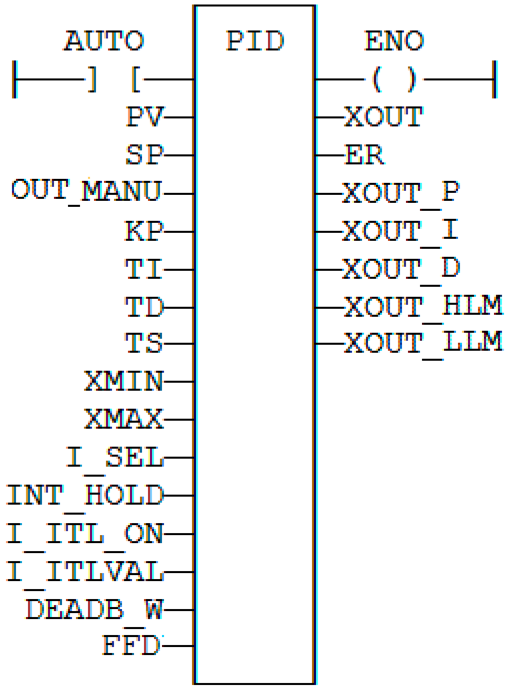
```
MyPID (AUTO, PV, SP, XOUT_MANU, KP, TI, TD, TS, XMIN, XMAX,
        I_SEL, I_ITL_ON, I_ITLVAL, DEADB_ERR, FFD);
XOUT := MyPID.XOUT;
ER := MyPID.ER;
XOUT_P := MyPID.XOUT_P;
XOUT_I := MyPID.XOUT_I;
XOUT_D := MyPID.XOUT_D;
XOUT_HLM := MyPID.XOUT_HLM;
XOUT_LLM := MyPID.XOUT_LLM;
```

FBD LANGUAGE



LD LANGUAGE

ENO has the same state as the input rung.



IL LANGUAGE

MyPID is a declared instance of PID function block.

```
Op1: CAL MyPID (AUTO, PV, SP, XOUT_MANU, KP, TI, TD, TS,
                XMIN, XMAX, I_SEL, I_ITL_ON, I_ITLVAL,
                DEADB_ERR, FFD)
     LD   MyPID.XOUT
     ST   XOUT
     LD   MyPID.ER
     ST   ER
     LD   MyPID.XOUT_P
     ST   XOUT_P
     LD   MyPID.XOUT_I
     ST   XOUT_I
     LD   MyPID.XOUT_D
     ST   XOUT_D
     LD   MyPID.XOUT_HLM
     ST   XOUT_HLM
     LD   MyPID.XOUT_LLM
     ST   XOUT_LLM
```

printf

FUNCTION
Display a trace output.

INPUTS

Name	Type	Description
FMT	STRING	Trace message.
ARG1..ARG4	DINT	Numerical arguments to be included in the trace.

OUTPUTS

Name	Type	Description
Q	BOOL	Return check.

REMARKS
This function works as the famous “printf” function of the “C” language, with up to 4 integer arguments. You can use the following pragmas in the FMT trace message to represent the arguments according to their left to the right order:

%ld signed value in decimal
%lu unsigned value in decimal
%lx value in hexadecimal

The trace message is displayed in the LOG window with runtime messages. Trace is supported by the simulator.

YOUR TARGET PLATFORM MAY NOT SUPPORT TRACE FUNCTIONS. PLEASE REFER TO OEM INSTRUCTIONS FOR FURTHER DETAILS ON AVAILABLE FEATURES.

EXAMPLE

```
(* i1, i2, i3, i4 are declared as DINT *)
i1 := 1;
i2 := 2;
i3 := 3;
i4 := 4;
printf ('i1=%ld; i2=%ld; i3=%ld; i4=%ld', i1, i2, i3, i4);
```

Output message:

i1=1; i2=2; i3=3; i4=4;

RAMP

FUNCTION BLOCK

Limit the ascendance or descendance of a signal.

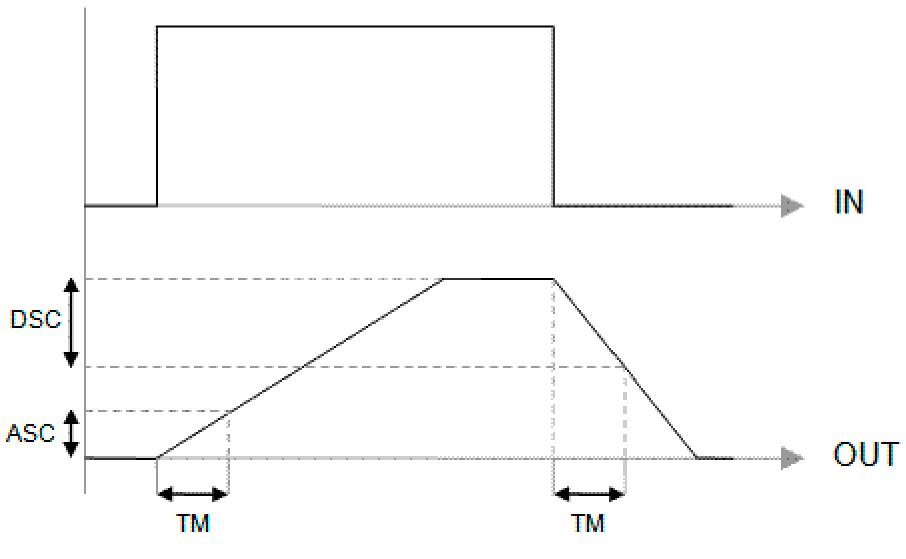
INPUTS

Name	Type	Description
IN	REAL	Input signal.
ASC	REAL	Maximum ascendance during time base.
DSC	REAL	Maximum descendance during time base.
TM	TIME	Time base.
RST	BOOL	Reset.

OUTPUTS

Name	Type	Description
OUT	REAL	Ramp signal.

TIME DIAGRAM



REMARKS

Parameters are not updated constantly. They are taken into account when only:

- The first time the block is called.
- When the reset input (**RST**) is **TRUE**.

In these two situations, the output is set to the value of IN input.

ASC and DSC give the maximum ascendant and descendant growth during the TB time base.
Both must be expressed as **positive** numbers.

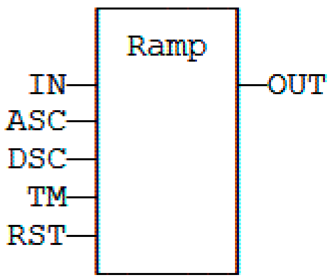
In LD language, the operation is executed only if the input rung (EN) is **TRUE**. The output rung (ENO) keeps the same value as the input rung.

ST LANGUAGE

MyRamp is a declared instance of **RAMP** function block.

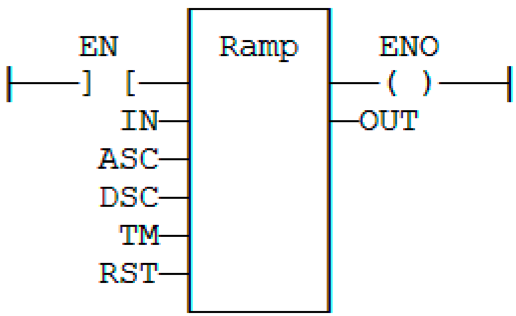
```
MyRamp (IN, ASC, DSC, TM, RST);
OUT := MyRamp.OUT;
```

FBD LANGUAGE



LD LANGUAGE

The function is executed only if EN is **TRUE**.
ENO keeps the same value as EN.



IL LANGUAGE

MyRamp is a declared instance of **RAMP** function block.

```
Op1: CAL MyRamp (IN, ASC, DSC, TM, RST)
     LD  MyBlinker.OUT
     ST  OUT
```

SerializeIn

FUNCTION

Extract the value of a variable from a binary frame.

INPUTS

Name	Type	Description
FRAME	USINT	Source buffer - must be an array.
DATA	ANY(*)	Destination variable to be copied.
POS	DINT	Position in the source buffer.
BIGENDIAN	BOOL	TRUE if the frame is encoded with Big Endian format.

(*) **DATA** cannot be a **STRING**.

OUTPUTS

Name	Type	Description
NEXTPOS	DINT	Position in the source buffer after the extracted data. 0 in case or error (invalid position / buffer size).

REMARKS

This function is commonly used for extracting data from a communication frame in binary format.

In LD language, the operation is executed only if the input rung (EN) is **TRUE**. The output rung (ENO) keeps the same value as the input rung. This function is not available in IL language.

The **FRAME** input must fit the input position and data size. If the value cannot be safely extracted, the function returns 0.

The **DATA** input must be directly connected to a variable, and cannot be a constant or complex expression. This variable will be forced with the extracted value.

The function extracts the following number of bytes from the source frame:

1 byte for **BOOL**, **SINT**, **USINT** and **BYTE** variables
2 bytes for INT, **UINT** and **WORD** variables
4 bytes for **DINT**, **UDINT**, **DWORD** and **REAL** variables
8 bytes for **LINT** and **LREAL** variables

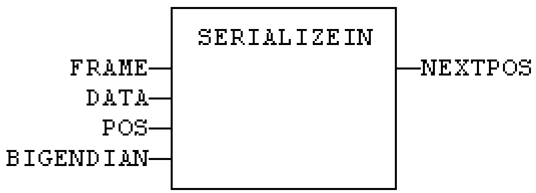
The function cannot be used to serialize **STRING** variables.

The function returns the position in the source frame, after the extracted data. Thus the return value can be used as a position for the next serialization.

ST LANGUAGE

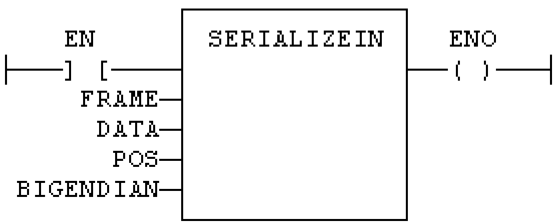
```
Q := SERIALIZEIN (FRAME, DATA, POS, BIGENDIAN);
```

FBD LANGUAGE



LD LANGUAGE

The function is executed only if EN is **TRUE**.
ENO keeps the same value as EN.



IL LANGUAGE

Not available.

SEE ALSO

SERIALIZEOUT

SerializeOut

FUNCTION

Copy the value of a variable to a binary frame.

INPUTS

Name	Type	Description
FRAME	USINT	Destination buffer - must be an array.
DATA	ANY(*)	Source variable to be copied.
POS	DINT	Position in the destination buffer.
BIGENDIAN	BOOL	TRUE if the frame is encoded with Big Endian format.

(*) **DATA** cannot be a **STRING**.

OUTPUTS

Name	Type	Description
NEXTPOS	DINT	Position in the destination buffer after the copied data. 0 in case or error (invalid position / buffer size).

REMARKS

This function is commonly used for building a communication frame in binary format.

In LD language, the operation is executed only if the input rung (EN) is **TRUE**. The output rung (ENO) keeps the same value as the input rung. This function is not available in IL language.

The **FRAME** input must be an array large enough to receive the data. If the data cannot be safely copied to the destination buffer, the function returns 0.

The function copies the following number of bytes to the destination frame:

1 byte for **BOOL**, **SINT**, **USINT** and **BYTE** variables
2 bytes for INT, **UINT** and **WORD** variables
4 bytes for **DINT**, **UDINT**, **DWORD** and **REAL** variables
8 bytes for **LINT** and **LREAL** variables

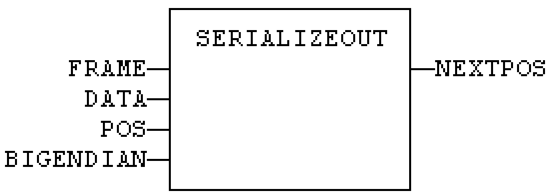
The function cannot be used to serialize **STRING** variables.

The function returns the position in the destination frame, after the copied data. Thus the return value can be used as a position for the next serialization.

ST LANGUAGE

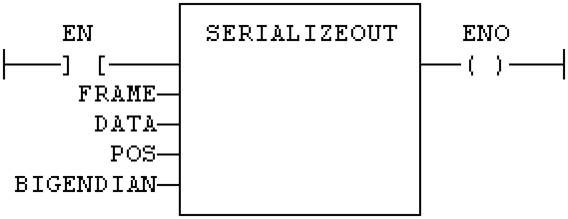
```
Q := SERIALIZEOUT (FRAME, DATA, POS, BIGENDIAN);
```

FBD LANGUAGE



LD LANGUAGE

The function is executed only if EN is **TRUE**.
ENO keeps the same value as EN.



IL LANGUAGE

Not available.

SEE ALSO

SERIALIZEIN

SerGetString

FUNCTION

Extract a string from a binary frame.

INPUTS

Name	Type	Description
FRAME	USINT	Source buffer - must be an array.
DST	STRING	Destination variable to be copied.
POS	DINT	Position in the source buffer.
MAXLEN	DINT	Specifies a fixed length string.
EOS	BOOL	Specifies a null terminated string.
HEAD	BOOL	Specifies a string headed with its length.

OUTPUTS

Name	Type	Description
NEXTPOS	DINT	Position in the source buffer after the extracted data. 0 in case or error (invalid position / buffer size)

REMARKS

This function is commonly used for extracting data from a communication frame in binary format.

In LD language, the operation is executed only if the input rung (EN) is **TRUE**. The output rung (ENO) keeps the same value as the input rung. This function is not available in IL language.

The **FRAME** input must fit the input position and data size. If the value cannot be safely extracted, the function returns 0.

The DST input must be directly connected to a variable, and cannot be a constant or complex expression. This variable will be forced with the extracted value.

The function extracts the following bytes from the source frame:

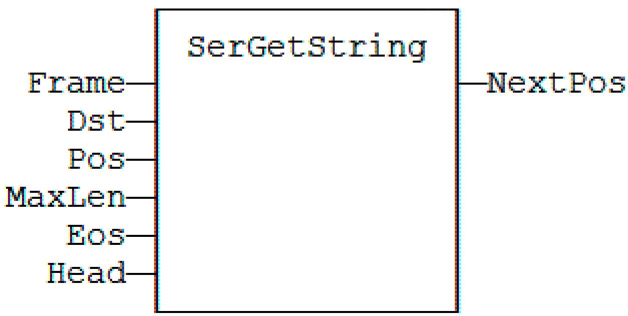
MAXLEN	EOS	HEAD	Description
<> 0	any	any	The string is stored on a fixed length specified by MAXLEN. If the string is actually smaller, the space is completed with null bytes.
= 0	TRUE	any	The string is stored with its actual length and terminated by anull byte.
= 0	FALSE	TRUE	The string is stored with its actual length and preceded by its length stored on one byte
=0	FALSE	FALSE	invalid call

The function returns the position in the source frame, after the extracted data. Thus the return value can be used as a position for the next serialization.

ST LANGUAGE

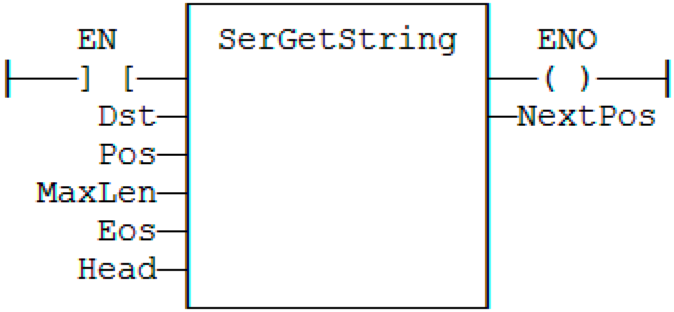
```
Q := SerGetString (FRAME, DSR, POS, MAXLEN, EOS, HEAD);
```

FBD LANGUAGE



LD LANGUAGE

The function is executed only if EN is TRUE.
ENO keeps the same value as EN.



IL LANGUAGE

Not available.

SERIO

FUNCTION BLOCK

Serial communication.

INPUTS

Name	Type	Description
RUN	BOOL	Enable comunication (opens the comm port).
SND	BOOL	TRUE if data has to be sent.
CONF	STRING	Configuration of the communication port.
DATASND	STRING	Data to send.

OUTPUTS

Name	Type	Description
OPEN	BOOL	TRUE if the communication port is open.
RCV	BOOL	TRUE if data has been received.
ERR	BOOL	TRUE if error detected during sending data.
DATARCV	STRING	Received data.

REMARKS

The RUN input does not include an edge detection. The block tries to open the port on each call if RUN is **TRUE** and if the port is still not successfully open. The **CONF** input is used for settings when opening the port. Please refer to your OEM instructions for further details about possible parameters.

The SND input does not include an edge detection. Characters are sent on each call if SND is **TRUE** and **DATASND** is not empty.

The **DATARCV** string is erased on each cycle with received data (if any). Your application is responsible for checking or storing received character immediately after the call to **SERIO** block.

SERIO is available during simulation. In that case, the **CONF** input defines the communication port according to the syntax of the **MODE** command. For example:

```
'COM1:9600,N,8,1'
```

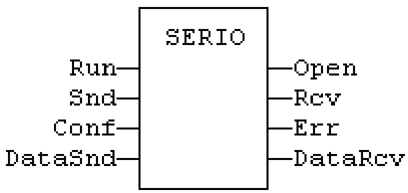
The **SERIO** block may not be supported on some targets. Refer to your OEM instructions for further details.

ST LANGUAGE

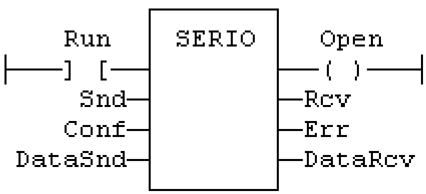
MySer is a declared instance of **SERIO** function block.

```
MySer (RUN, SND, CONF, DATASND);
OPEN := MySer.OPEN;
RCV := MySer.RCV;
ERR := MySer.ERR;
DATARCV := MySer.DATARCV;
```

FBD LANGUAGE



LD LANGUAGE



IL LANGUAGE

MySer is a declared instance of **SERIO** function block.

```
Op1: CAL    MySer (RUN, SND, CONF, DATASND)
     LD     MySer.OPEN
     ST     OPEN
     LD     MySer.RCV
     ST     RCV
     LD     MySer.ERR
     ST     ERR
     LD     MySer.DATARCV
     ST     DATARCV
```

SerPutString

FUNCTION
Copies a string to a binary frame

INPUTS

Name	Type	Description
FRAME	USINT	Destination buffer - must be an array.
DST	STRING	Source variable to be copied.
POS	DINT	Position in the source buffer.
MAXLEN	DINT	Specifies a fixed length string.
EOS	BOOL	Specifies a null terminated string.
HEAD	BOOL	Specifies a string headed with its length.

OUTPUTS

Name	Type	Description
NEXTPOS	DINT	Position in the destination buffer after the copied data. 0 in case or error (invalid position / buffer size)

REMARKS
This function is commonly used for storing data to a communication frame.

In LD language, the operation is executed only if the input rung (EN) is **TRUE**. The output rung (ENO) keeps the same value as the input rung. This function is not available in IL language.

The **FRAME** input must fit the input position and data size. If the value cannot be safely copied, the function returns 0.

The function copies the following bytes to the frame:

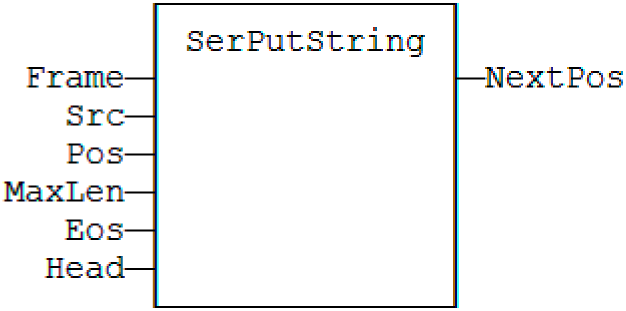
MAXLEN	EOS	HEAD	Description
<> 0	any	any	The string is stored on a fixed length specified by MAXLEN. If the string is actually smaller, the space is completed with null bytes. If the string is longer, it is truncated.
= 0	TRUE	any	The string is stored with its actual length and terminated by anull byte.
= 0	FALSE	TRUE	The string is stored with its actual length and preceded by its length stored on one byte.
=0	FALSE	FALSE	invalid call

The function returns the position in the source frame, after the stored data. Thus the return value can be used as a position for the next serialization.

ST LANGUAGE

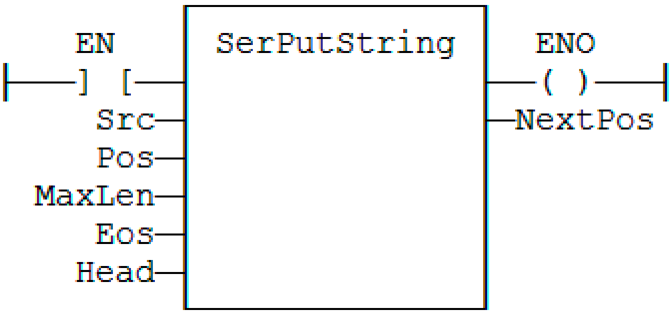
```
Q := SerPutString (FRAME, DSR, POS, MAXLEN, EOS, HEAD);
```

FBD LANGUAGE



LD LANGUAGE

The function is executed only if EN is **TRUE**.
ENO keeps the same value as EN.



IL LANGUAGE

Not available.

SigID

FUNCTION

Get the identifier of a Signal resource.

INPUTS

Name	Type	Description
SIGNAL	STRING	Name of the signal resource - must be a constant value!

Name	Type	Description
COL	STRING	Name of the column within the signal resource - must be a constant value!

OUTPUTS

Name	Type	Description
ID	DINT	ID of the signal - to be passed to other blocks.

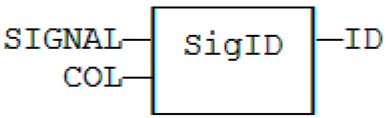
REMARKS

Some blocks have arguments that refer to a signal resource. For all these blocks, the signal argument is materialized by a numerical identifier. This function enables you to get the identifier of a signal defined as a resource.

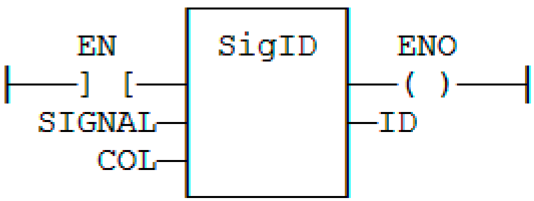
ST LANGUAGE

```
ID := SigID ('MySignal', 'FirstColumn');
```

FBD LANGUAGE



LD LANGUAGE



IL LANGUAGE

```
Op1: LD    'MySignal'
     SigID 'FirstColumn'
     ST    ID
```

SEE ALSO

SigPlay

SigScale

Signal resources

SigPlay

FUNCTION BLOCK

Generate a signal defined in a resource.

INPUTS

Name	Type	Description
IN	BOOL	Triggering command.
ID	DINT	ID of the signal resource, provided by the SigID function.
RST	BOOL	Reset command.
TM	TIME	Minimum duration in between two changes of the output.

OUTPUTS

Name	Type	Description
Q	BOOL	TRUE when the signal is finished.
OUT	REAL	Generated signal.
ET	TIME	Elapsed time.

REMARKS

The ID argument is the identifier of the signal resource. Use the SigID function to get this value.

The IN argument is used as a Play / Pause command to play the signal. The signal is not reset to the beginning when IN becomes **FALSE**. Instead, use the RST input that resets the signal and forces the OUT output to 0.

The TM input specifies the minimum amount of time in between two changes of the output signal. This parameter is ignored if less than the cycle scan time.

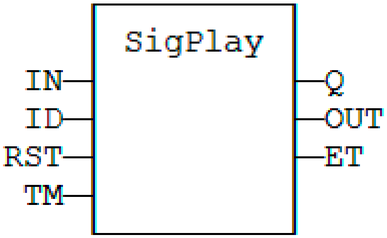
This function block includes its own timer. Alternatively, you can use the SigScale function if you want to trigger the signal using a specific timer.

ST LANGUAGE

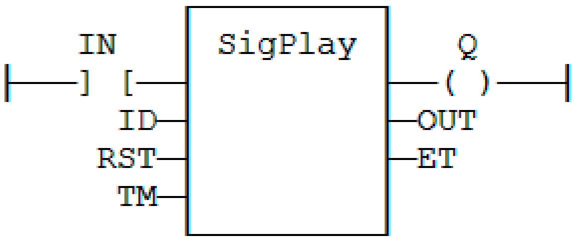
MySig is a declared instance of **SIGPLAY** function block.

```
MySig (II, ID, RST, TM);
Q := MySig.Q;
OUT := MySig.OUT;
ET := MySig.ET;
```

FBD LANGUAGE



LD LANGUAGE



IL LANGUAGE

MySig is a declared instance of **SIGPLAY** function block.

```
Op1: CAL MySig (II, ID, RST, TM)
     LD  MySig.Q
     ST  Q
     LD  MySig.OUT
     ST  OUT
     LD  MySig.ET
     ST  ET
```

SEE ALSO

SigScale

SigID

Signal resources

SigScale

FUNCTION

Get a point from a Signal resource.

INPUTS

Name	Type	Description
ID	DINT	ID of the signal resource, provided by SigID function.
IN	TIME	Time (X) coordinate of the wished point within the signal resource.

OUTPUTS

Name	Type	Description
Q	REAL	Value (Y) coordinate of the point in the signal.

REMARKS

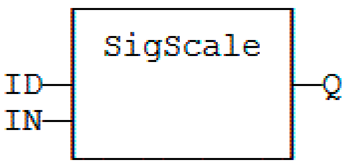
The ID argument is the identifier of the signal resource. Use the SigID function to get this value.

This function converts a time value to a analog value such as defined in the signal resource. This function can be used instead of SigPlay function block if you want to trigger the signal using a specific timer.

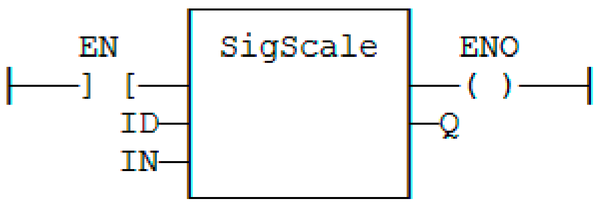
ST LANGUAGE

```
Q := SigScale (ID, IN);
```

FBD LANGUAGE



LD LANGUAGE



IL LANGUAGE

```
Op1: LD       IN
     SigScale ID
     ST       Q
```

SEE ALSO

SigPlay

SigID
Signal resources

STACKINT

FUNCTION BLOCK
Manages a stack of **DINT** integers.

INPUTS

Name	Type	Description
PUSH	BOOL	Command: when changing from FALSE to TRUE, the value of IN is pushed on the stack.
POP	BOOL	Pop command: when changing from FALSE to TRUE, deletes the top of the stack.
R1	BOOL	Reset command: if TRUE, the stack is emptied and its size is set to N.
IN	DINT	Value to be pushed on a rising pulse of PUSH.
N	DINT	Maximum stack size - cannot exceed 128.

OUTPUTS

Name	Type	Description
EMPTY	BOOL	TRUE if the stack is empty.
OFLO	BOOL	TRUE if the stack is full.
OUT	DINT	Value at the top of the stack.

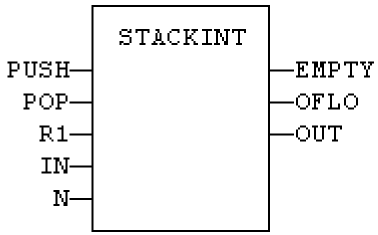
REMARKS
Push and pop operations are performed on rising pulse of **PUSH** and POP inputs. In LD language, the input rung is the **PUSH** command. The output rung is the **EMPTY** output.

The specified size (N) is taken into account only when the R1 (reset) input is **TRUE**.

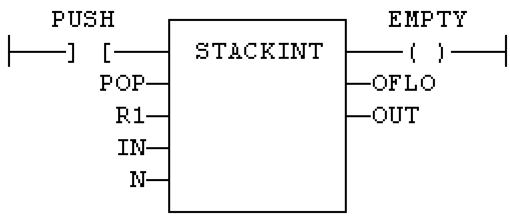
ST LANGUAGE
MyStack is a declared instance of **STACKINT** function block.

```
MyStack (PUSH, POP, R1, IN, N);
EMPTY := MyStack.EMPTY;
OFLO := MyStack.OFLO;
OUT := MyStack.OUT;
```

FBD LANGUAGE



LD LANGUAGE



IL LANGUAGE

MyStack is a declared instance of **STACKINT** function block.

```
Op1: CAL MyStack (PUSH, POP, R1, IN, N)
     LD  MyStack.EMPTY
     ST  EMPTY
     LD  MyStack.OFLO
     ST  OFLO
     LD  MyStack.OUT
     ST  OUT
```

SEE ALSO

AVERAGE
INTEGRAL
DERIVATE
LIM_ALRM
HYSTER

SurfLin

FUNCTION BLOCK

Linear interpolation on a surface.

INPUTS

Name	Type	Description
X	REAL	X coordinate of the point to be interpolated.
Y	REAL	Y coordinate of the point to be interpolated.
XAxis	REAL[]	X coordinates of the known points of the X axis.
YAxis	REAL[]	Y coordinates of the known points of the Y axis.
ZVal	REAL[,]	Z coordinate of the points defined by the axis.

OUTPUTS

Name	Type	Description
Z	REAL	Interpolated Z value corresponding to the X,Y input point
OK	BOOL	TRUE if successful.
ERR	DINT	Error code if failed - 0 if OK.

REMARKS

This function performs linear surface interpolation in between a list of points defined in XAxis and YAxis single dimension arrays. The output Z value is an interpolation of the Z values of the four rounding points defined in the axis. Z values of defined points are passed in the ZVal matrix (two dimension array).

ZVal dimensions must be understood as: ZVal [iX , iY]

Values in X and Y axis must be sorted from the smallest to the biggest. There must be at least two points defined in each axis. ZVal must fit the dimension of XAxis and YAxis arrays. For instance:

```
XAxis : ARRAY [0..2] of REAL;
YAxis : ARRAY [0.3] of REAL;
ZVal : ARRAY [0..2,0..3] of REAL;
```

In case the input point is outside the rectangle defined by XAxis and YAxis limits, the Z output is bound to the corresponding value and an error is reported.

The ERR output gives the cause of the error if the function fails:

Error code	Meaning
0	OK
1	Invalid dimension of input arrays
2	Invalid points for the X axis
3	Invalid points for the Y axis
4	X,Y point is out of the defined axis

RTC Management Functions

The following functions read the real time clock of the target system:

Function	Description
DTCurDate	Get current date stamp.
DTCurTime	Get current time stamp.
DTCurDateTime	Get current date and time stamp.
DTDay	Get day from date stamp.
DTMonth	Get month from date stamp.
DTYear	Get year from date stamp.
DTSec	Get seconds from time stamp.
DTMin	Get minutes from time stamp.
DTHour	Get hours from time stamp.
DTMs	Get milliseconds from time stamp.

The following functions format the current date/time to a string:

Function	Description
DAY_TIME	With predefined format.
DTFORMAT	With custom format.

The following function Blocks are used for triggering operations:

Function Block	Description
DTAt	Pulse signal at the given date/time.
DTEvery	Pulse signal with long period.

REAL TIME CLOCK MAY BE NOT AVAILABLE ON SOME TARGETS. PLEASE REFER TO OEM INSTRUCTIONS FOR FURTHER DETAILS ABOUT AVAILABLE FEATURES.

DAY_TIME

FUNCTION
Format the current date/time to a string.

INPUTS

Name	Type	Description
SEL	DINT	Format selector.

OUTPUTS

Name	Type	Description
Q	STRING	String containing formatted date or time.

REAL TIME CLOCK MAY BE NOT AVAILABLE ON SOME TARGETS. PLEASE REFER TO OEM INSTRUCTIONS FOR FURTHER DETAILS ABOUT AVAILABLE FEATURES.

REMARKS

Possible values of the SEL input are:

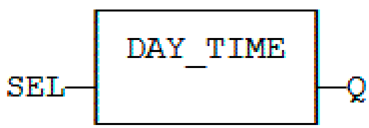
Value	Meaning
1	current time - format: ‘HH:MM:SS’.
2	day of the week.
0 (default)	current date - format: ‘YYYY/MM/DD’.

In LD language, the operation is executed only if the input rung (EN) is **TRUE**. The output rung (ENO) keeps the same value as the input rung.

ST LANGUAGE

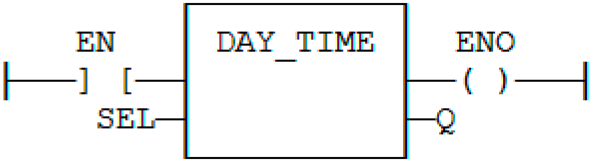
```
Q := DAY_TIME (SEL);
```

FBD LANGUAGE



LD LANGUAGE

The function is executed only if EN is **TRUE**.
ENO keeps the same value as EN.



IL LANGUAGE

```
Op1: LD SEL
     DAY_TIME
     ST Q
```

SEE ALSO

HYPERLINK “AO-RTC-DTFORMAT.docx” DTFORMAT

DTAT

FUNCTION BLOCK

Generate a pulse at given date and time

INPUTS

Name	Type	Description
YEAR	DINT	Desired year (e.g. 2006).
MONTH	DINT	Desired month (1 = January).
DAY	DINT	Desired day (1 to 31).
TMOFDAY	TIME	Desired time.
RST	BOOL	Reset command.

OUTPUTS

Name	Type	Description
QAT	BOOL	Pulse signal.
QPAST	BOOL	**TRUE** if elapsed.

REAL TIME CLOCK MAY BE NOT AVAILABLE ON SOME TARGETS. PLEASE REFER TO OEM INSTRUCTIONS FOR FURTHER DETAILS ABOUT AVAILABLE FEATURES.

REMARKS

Parameters are not updated constantly. They are taken into account when only:

- the first time the block is called.
- when the reset input (**RST**) is **TRUE**.

In these two situations, the outputs are reset to **FALSE**.

The first time the block is called with RST=**FALSE** and the specified date/stamp is passed, the output **QPAST** is set to **TRUE**, and the output QAT is set to **TRUE** for one cycle only (pulse signal).

Highest units are ignored if set to 0. For instance, if arguments are year=0, month=0, day = 3, tmofday=t#10h then the block will trigger on the next 3rd day of the month at 10h.

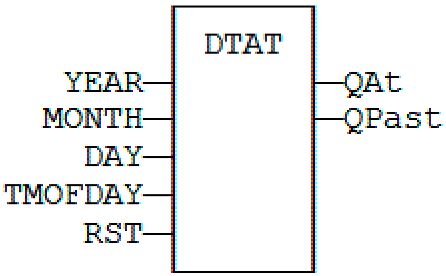
In LD language, the block is activated only if the input rung is **TRUE**.

ST LANGUAGE

MyDTAT is a declared instance of **DTAT** function block.

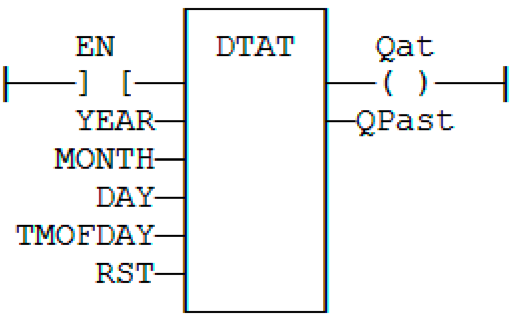
```
MyDTAT (YEAR, MONTH, DAY, TMOFDAY, RST);
QAT := MyDTAT.QAT;
QPAST := MyDTATA.QPAST;
```

FBD LANGUAGE



LD LANGUAGE

Called only if EN is **TRUE**.



IL LANGUAGE

MyDTAT is a declared instance of **DTAT** function block.

```
    Op1: CAL MyDTAT (YEAR, MONTH, DAY, TMOFDAY, RST)
         LD  MyDTAT.QAT
         ST  QAT
         LD  MyDTATA.QPAST
         ST  QPAST
```

SEE ALSO
`DTEvery`

DTCURDATE

FUNCTION
Get current date stamp

SYNTAX
`Q := DTCurDate ();`

OUTPUTS

Name	Type	Description
Q	DINT	numerical stamp representing the current date.

DTCURDATETIME

FUNCTION BLOCK
Get current time stamp

SYNTAX
`Inst _ DTCurDateTime (bLocal);`

OUTPUTS

Name	Type	Description
bLocal	BOOL	TRUE if local time is requested (GMT if FALSE).
.Year	DINT	Output: current year
.Month	DINT	Output: current month
.Day	DINT	Output: current day
.Hour	DINT	Output: current time: hours
.Min	DINT	Output: current time: minutes
.Sec	DINT	Output: current time: seconds

Name	Type	Description
.MSec	DINT	Output: current time: milliseconds
.TmOfDay	TIME	Output: current time of day (since midnight)

DTCURTIME

FUNCTION
Get current time stamp

SYNTAX
Q := DTCurTime ();

OUTPUTS

Name	Type	Description
Q	DINT	numerical stamp representing the current time of the day.

DTDAY

FUNCTION
Extract the day of the month from a date stamp

SYNTAX
Q := DTDay (iDate);

INPUTS

Name	Type	Description
IDATE	DINT	numerical stamp representing a date.

OUTPUTS

Name	Type	Description
Q	DINT	day of the month of the date (1..31).

DTEVERY

FUNCTION BLOCK

Generate a pulse signal with long period.

INPUTS

Name	Type	Description
RUN	DINT	Enabling command.
DAYS	DINT	Period : number of days.
TM	TIME	Rest of the period (if not a multiple of 24h).

OUTPUTS

Name	Type	Description
Q	BOOL	Pulse signal.

REMARKS

This block provides a pulse signal with a period of more than 24h. The period is expressed as:

DAYS * 24h + TM

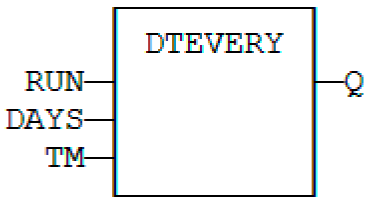
For instance, specifying **DAYS**=1 and TM=6h means a period of 30 hours.

ST LANGUAGE

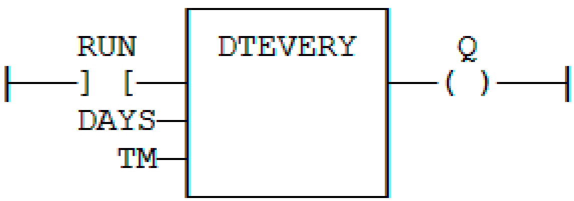
MyDTEVERY is a declared instance of **DTEVERY** function block.

```
MyDTEVERY (RUN DAYS, TM);
Q := MyDTEVERY.Q;
```

FBD LANGUAGE



LD LANGUAGE



IL LANGUAGE

MyDTEVERY is a declared instance of **DTEVERY** function block.

```
Op1: CAL MyDTEVERY (RUN DAYS, TM)
     LD  MyDTEVERY.Q
     ST  Q
```

SEE ALSO

HYPERLINK “AO-RTC-DTAt.docx” DTAT

DTFORMAT

FUNCTION

Format the current date/time to a string with a custom format.

INPUTS

Name	Type	Description
FMT	STRING	Format string.

OUTPUTS

Name	Type	Description
Q	STRING	String containing formatted date or time.

REAL TIME CLOCK MAY BE NOT AVAILABLE ON SOME TARGETS. PLEASE REFER TO OEM INSTRUCTIONS FOR FURTHER

REMARKS

The format string may contain any character. Some special markers beginning with the '%' character indicates a date/time information:

%Y	Year including century (e.g. 2006)
%y	Year without century (e.g. 06)
%m	Month (1..12)
%d	Day of the month (1..31)
%H	Hours (0..23)
%M	Minutes (0..59)
%S	Seconds (0..59)

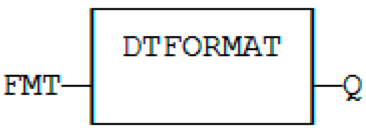
EXAMPLE

```
(* let's say we are at July 04th 2006, 18:45:20 *)
   Q := DTFORMAT ('Today is %Y/%m/%d - %H:%M:%S');
(* Q is 'Today is 2006/07/04 - 18:45:20 *)
```

ST LANGUAGE

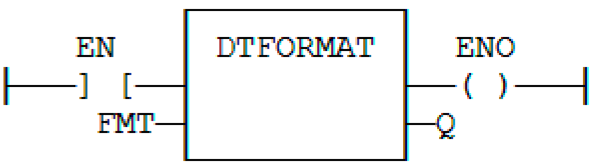
```
Q := DTFORMAT (FMT);
```

FBD LANGUAGE



LD LANGUAGE

The function is executed only if EN is **TRUE**.
ENO keeps the same value as EN.



IL LANGUAGE

```
Op1: LD FMT
     DTFORMAT
     ST Q
```

SEE ALSO
HYPERLINK “AO-RTC-DAY_TIME.docx” DAY_TIME

DTHOUR

FUNCTION
Extract the hours from a time stamp

SYNTAX
Q := DTHour (iTime);

INPUTS

Name	Type	Description
ITIME	DINT	numerical stamp representing a time.

OUTPUTS

Name	Type	Description
Q	DINT	Hours of the time (0..23).

DTMIN

FUNCTION
Extract the minutes from a time stamp

SYNTAX
Q := DTMin (iTime);

INPUTS

Name	Type	Description
ITIME	DINT	numerical stamp representing a time.

OUTPUTS

Name	Type	Description
Q	DINT	Minutes of the time (0..59).

DTMONTH

FUNCTION
Extract the month from a date stamp

SYNTAX
```
Q := DTMonth (iDate);
```

INPUTS

Name	Type	Description
IDATE	DINT	numerical stamp representing a date.

OUTPUTS

Name	Type	Description
Q	DINT	month of the date (1..12).

DTMS

FUNCTION
Extract the milliseconds from a date stamp

SYNTAX
```
Q := DTMs (iDate);
```

INPUTS

Name	Type	Description
IDATE	DINT	numerical stamp representing a date.

OUTPUTS

Name	Type	Description
Q	DINT	milliseconds of the time (0..999).

DTSEC

FUNCTION
Extract the seconds from a time stamp

SYNTAX
```
Q := DTSec (iTime);
```

INPUTS

Name	Type	Description
ITIME	DINT	numerical stamp representing a time.

OUTPUTS

Name	Type	Description
Q	DINT	Seconds of the time (0..59).

DTYEAR

FUNCTION
Extract the year from a date stamp

SYNTAX
```
Q := DTYear (iDate);
```

INPUTS

Name	Type	Description
IDATE	DINT	numerical stamp representing a date.

OUTPUTS

Name	Type	Description
Q	DINT	year of the date (ex: 2004).

Text Buffer Manipulation

Strings are limited to 255 characters. Here is a set of functions and function blocks for working with not limited text buffers. Text buffers are dynamically allocated or re-allocated.

TEXT BUFFERS MANAGEMENT FUNCTIONS USE SAFE DYNAMIC MEMORY ALLOCATION THAT NEEDS TO BE CONFIGURED IN THE PROJECT SETTINGS. FROM THE PROJECT SETTINGS, PRESS THE “ADVANCED” PUSH BUTTON AND GO TO “MEMORY” TAB. HERE YOU CAN SETUP THE MEMORY FOR SAFE DYNAMIC ALLOCATION.

THERE MUST BE ONE INSTANCE OF THE TXBMANAGER DECLARED IN YOUR APPLICATION FOR USING THESE FUNCTIONS.

THE APPLICATION SHOULD TAKE CARE OF RELEASING MEMORY ALLOCATED FOR EACH BUFFER. ALLOCATING BUFFERS WITHOUT FREING THEM WILL LEAD TO MEMORY LEAKS.

The application is responsible for freeing all allocated text buffers. However, all allocated buffers are automatically released when the application stops.

MEMORY MANAGEMENT / MISCELLANEOUS

`TxbManager`	Main gatherer of text buffer data in memory.
`TxbLastError`	Get detailed error report about last call.

ALLOCATION / EXCHANGE WITH FILES

`TxbNew`	Allocate a new empty buffer.
`TxbNewString`	Allocate a new buffer initialized with string.
`TxbFree`	Release a text buffer.
`TxbReadFile`	Allocate a new buffer from file.
`TxbWriteFile`	Store a text buffer to file.

DATA EXCHANGE

`TxbGetLength`	Get length of a text buffer.
`TxbGetData`	Store text contents to an array of characters.
`TxbGetString`	Store text contents to a string.
`TxbSetData`	Store an array of characters to a text buffer.
`TxbSetString`	Store string to text buffer.
`TxbClear`	Empty a text buffer.
`TxbCopy`	Copy a text buffer.

SEQUENTIAL READING

`TxbRewind`	Rewind sequential reading.
`TxbGetLine`	Sequential read line by line.

SEQUENTIAL WRITING

`TxbAppend`	Append variable value.
`TxbAppendLine`	Append a text line.

TxbAppendEol	Append end of line characters.
TxbAppendTxb	Append contents of another buffer.

UNICODE CONVERSIONS

TxbAnsiToUtf8	Convert a text buffer to UNICODE.
TxbUtf8ToAnsi	Convert a text buffer to ANSI.

TxbAnsiToUtf8

FUNCTION

DESCRIPTION

This function converts the whole contents of a text buffer from **ANSI** to **UNICODE** UTF8 encoding. Warning:

THIS FUNCTION MAY BE TIME AND MEMORY CONSUMING FOR LARGE BUFFERS.

UNICODE CONVERSION MAY BE NOT AVAILABLE ON SOME OPERATING SYSTEMS

INPUTS

Name	Type	Description
hTxb	DINT	Handle of the text buffer.

OUTPUTS

Name	Type	Description
bOK	BOOL	TRUE if successful.

TxbAppend

FUNCTION

DESCRIPTION

This function adds the contents of a variable, formatted as text, to a text buffer. The specified variable can have any data type.

INPUTS

Name	Type	Description
hTxb	DINT	Handle of the text buffer.
data	ANY	Any variable.

OUTPUTS

Name	Type	Description
bOK	BOOL	TRUE if successful.

TxbAppendEol

FUNCTION

DESCRIPTION

This function adds end of line characters to a text buffer.

INPUTS

Name	Type	Description
hTxb	DINT	Handle of the text buffer.

OUTPUTS

Name	Type	Description
bOK	BOOL	TRUE if successful.

TxbAppendLine

FUNCTION

DESCRIPTION

This function adds the contents of the specified string variable to a text buffer, plus end of line characters.

INPUTS

Name	Type	Description
hTxb	DINT	Handle of the text buffer.
szText	STRING	String to be added to the text.

OUTPUTS

Name	Type	Description
bOK	BOOL	TRUE if successful.

TxbAppendTxb

FUNCTION

DESCRIPTION

This function adds the contents of the hTxb text buffer to the hTxbDst text buffer.

INPUTS

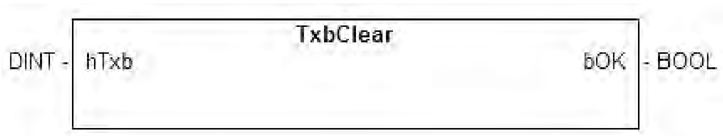
Name	Type	Description
hTxbDst	DINT	Handle of the text buffer to be completed.
hTxb	DINT	Handle of the text buffer to be added.

OUTPUTS

Name	Type	Description
bOK	BOOL	TRUE if successful.

TxbClear

FUNCTION



DESCRIPTION

This function empties a text buffer.

INPUTS

Name	Type	Description
hTxb	DINT	Handle of the text buffer.

OUTPUTS

Name	Type	Description
bOK	BOOL	TRUE if successful.

TxbCopy

FUNCTION

DESCRIPTION

This function copies the contents of the hTxb buffer the to hTxbDst buffer.

INPUTS

Name	Type	Description
hTxbDst	DINT	Handle of the destination text buffer.
hTxb	DINT	Handle of the source text buffer.

OUTPUTS

Name	Type	Description
bOK	BOOL	TRUE if successful.

TxbFree

FUNCTION

DESCRIPTION

This function releases a text buffer from memory.

INPUTS

Name	Type	Description
hTxb	DINT	Handle of a valid text buffer.

OUTPUTS

Name	Type	Description
bOK	BOOL	TRUE if successful.

TxbGetData

FUNCTION

DIAGRAM

DESCRIPTION

This function copies the contents of a text buffer to an array of characters.

INPUTS

Name	Type	Description
hTxb	DINT	Handle of the text buffer.
arData	SINT[]	Array of characters to be filled with text.

OUTPUTS

Name	Type	Description
bOK :	BOOL	TRUE if successful.

TxbGetLength

FUNCTION

DESCRIPTION

This function returns the current length of a text buffer.

INPUTS

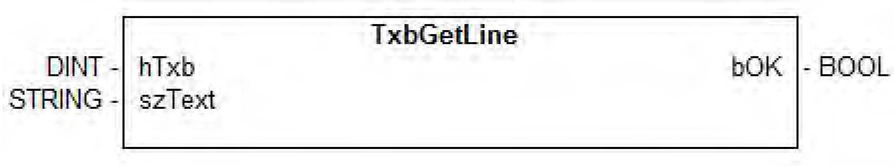
Name	Type	Description
hTxb	DINT	Handle of the text buffer.

OUTPUTS

Name	Type	Description
len	DINT	Number of characters in the text buffer.

TxbGetLine

FUNCTION



DESCRIPTION

This function sequentially reads a line of text from a text buffer. End of line characters are not copied to the output string.

INPUTS

Name	Type	Description
hTxb	DINT	Handle of the text buffer.
szText	STRING	String to be filled with read line.

OUTPUTS

Name	Type	Decription
bOK	BOOL	TRUE if successful.

TxbGetString

FUNCTION

DESCRIPTION

This function copies the contents of a text buffer to a string. The text is truncated if the string is not large enough.

INPUTS

Name	Type	Description
hTxb	DINT	Handle of the text buffer

OUTPUTS

Name	Type	Description
szText	STRING	String to be filled with text

TxbLastError

FUNCTION

DESCRIPTION

All TXB functions and blocks simply return a boolean information as a return value. This function can be called after any other function giving a **FALSE** return. It gives a detailed error code about the last detected error.

OUTPUTS

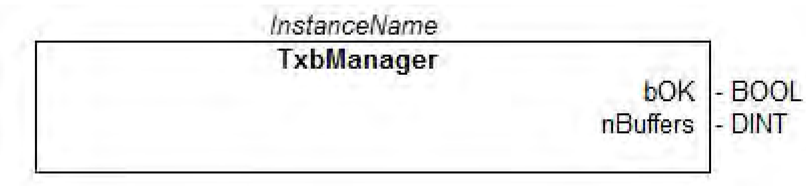
Name	Type	Description
iErr	DINT	Error code reported by the last call: 0 = OK other = error (see below)

ERROR CODES:

Code	Meaning
1	Invalid instance of TXBManager - should be only one.
2	Manager already open - should be only one instance of TxbManager.
3	Manager not open - no instance of TxbManager declared.
4	Invalid handle.
5	String has been truncated during copy.
6	Cannot read file.
7	Cannot write file.
8	Unsupported data type.
9	Too many text buffers allocated.

TxbManager

FUNCTION



DESCRIPTION

This function block is used for managing the memory allocated for text buffers. It takes care of releasing the corresponding memory when the application stops, and can be used for tracking memory leaks.

THERE MUST BE ONE AND ONLY ONE INSTANCE OF THIS BLOCK DECLARED IN THE IEC APPLICATION IN ORDER TO USE ANY OTHER TXB... FUNCTION.

OUTPUTS

Name	Type	Description
bOK	BOOL	TRUE if the text buffers memory system is correctly initialized.
nBuffers	DINT	Number of text buffers currently allocated in memory.

TxbNew

FUNCTION

DESCRIPTION

This function allocates a new text buffer initially empty. The application will be responsible for releasing the buffer by calling the TxbFree() function.

OUTPUTS

Name	Type	Description
hTxb	DINT	Handle of the new buffer

TxbNewString

FUNCTION

DESCRIPTION

This function allocates a new text buffer initially filled with the specified string. The application will be responsible for releasing the buffer by calling the TxbFree() function.

INPUTS

Input	Type	Description
szText	STRING	Initial value of the text buffer

OUTPUTS

Output	Type	Description
hTxb	DINT	Handle of the new buffer

TxbReadFile

FUNCTION

DESCRIPTION

This function allocates a new text buffer and fills it with the contents of the specified file. The application will be responsible for releasing the buffer by calling the TxbFree() function.

INPUTS

Name	Type	Description
szPath	STRING	Full qualified pathname of the file to be read

OUTPUTS

Name	Type	Description
hTxb	DINT	Handle of the new buffer

TxbRewind

FUNCTION

DESCRIPTION

This function resets the sequential reading of a text buffer (rewind to the beginning of the text).

INPUTS

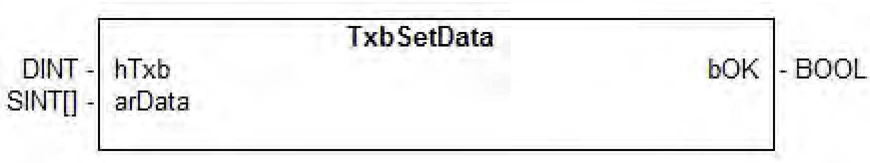
Name	Type	Description
hTxb	DINT	Handle of the text buffer.

OUTPUTS

Name	Type	Description
bOK	BOOL	TRUE if successful.

TxbSetData

FUNCTION



DESCRIPTION

This function copies an array of characters to a text buffer. All characters of the input array are copied.

INPUTS

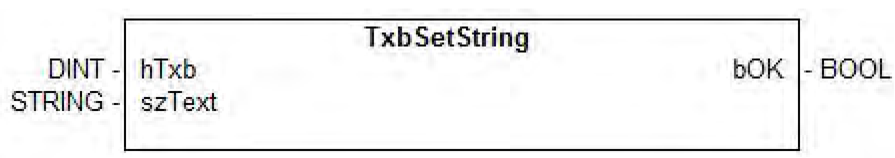
Name	Type	Description
hTxb	DINT	Handle of the text buffer
arData	SINT[]	Array of characters to copy

OUTPUTS

Name	Type	Description
bOK	BOOL	TRUE if successful

TxbSetString

FUNCTION



DESCRIPTION

This function copies the contents of a string to a text buffer.

INPUTS

Name	Type	Description
hTxb	DINT	Handle of the text buffer.
szText	STRING	String to be copied.

OUTPUTS

Name	Type	Description
bOK	BOOL	TRUE if successful.

TxbUtf8ToAnsi

FUNCTION

DESCRIPTION

This function converts the whole contents of a text buffer from **UNICODE** UTF8 to **ANSI** encoding.

THIS FUNCTION MAY BE TIME AND MEMORY CONSUMING FOR LARGE BUFFERS.

UNICODE CONVERSION MAY BE NOT AVAILABLE ON SOME OPERATING SYSTEMS

INPUTS

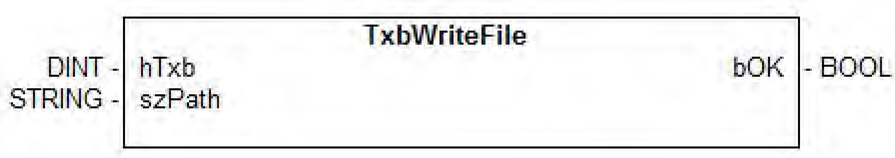
Name	Type	Description
hTxb	DINT	Handle of the text buffer.

OUTPUTS

Name	Type	Description
bOK	BOOL	TRUE if successful.

TxbWriteFile

FUNCTION



DESCRIPTION

This function stores the contents of a text buffer to a file. The text buffer remains allocated in memory.

INPUTS

Name	Type	Description
hTxb	DINT	Handle of the text buffer.
szPath	STRING	Full qualified pathname of the file to be created.

OUTPUTS

Name	Type	Description
bOK	BOOL	TRUE if successful.

UDP Management Functions

The following functions enable management of UDP sockets for building client or server applications over **ETHERNET** network:

Name	Description
udpCreate	create a UDP socket.
udpAddrMake	build an address buffer for UDP functions.
updSendTo	send a telegram.
udpRcvFrom	receive a telegram.
udpClose	close a socket.
udpIsValid	test if a socket is valid.

Each socket is identified in the application by a unique handle manipulated as a **DINT** value.

ALTHOUGH THE SYSTEM PROVIDES A SIMPLIFIED INTERFACE, YOU MUST BE FAMILIAR WITH THE SOCKET INTERFACE SUCH AS EXISTING IN OTHER PROGRAMMING LANGUAGES SUCH AS "C".

SOCKET MANAGEMENT MAY BE NOT AVAILABLE ON SOME TARGETS. PLEASE REFER TO OEM INSTRUCTIONS FOR FURTHER DETAILS ABOUT AVAILABLE FEATURES.

udpAddrMake

FUNCTION
Build an address buffer for UDP functions

SYNTAX
```
OK := udpAddrMake (IPADDR, PORT, ADD);
```

INPUTS

Name	Type	Description
IPADDR	STRING	IP address in form xxx.xxx.xxx.xxx
PORT	DINT	IP port number.
ADD	USINT[32]	Buffer where to store the UDP address (filled on output).

OUTPUTS

Name	Type	Description
OK	BOOL	TRUE if successful.

REMARKS
This functions is required for building a internal UDP address to be passed to the udpSendTo function in case of UDP client processing.

udpClose

FUNCTION
Release a socket

SYNTAX
```
OK := udpClose (SOCK);
```

INPUTS

Name	Type	Description
SOCK	DINT	ID of any socket.

OUTPUTS

Name	Type	Description
OK	BOOL	TRUE if successful.

REMARKS

You are responsible for closing any socket created by tcpListen, tcpAccept or tcpConnect functions, even if they have become invalid.

udpCreate

FUNCTION

Create a UDP socket

SYNTAX

```
SOCK := udpCreate (PORT);
```

INPUTS

Name	Type	Description
PORT	DINT	TCP port number to be attached to the server socket or 0 for a client socket.

OUTPUTS

Name	Type	Description
SOCK	DINT	ID of the new server socket.

REMARKS

This functions creates a new UDP socket. If the **PORT** argument is not 0, the socket is bound to the port and thus can be used as a server socket.

udpIsValid

FUNCTION

Test if a socket is valid

SYNTAX

```
OK := udpIsValid (SOCK);
```

INPUTS

Name	Type	Description
SOCK	DINT	ID of the socket.

OUTPUTS

Name	Type	Description
OK	BOOL	TRUE if specified socket is still valid.

udpRcvFrom

FUNCTION
Receive a UDP telegram

SYNTAX
```
OK := udpRcvFrom (SOCK, NB, ADD, DATA);
```

INPUTS

Name	Type	Description
SOCK	DINT	ID of the client socket.
NB	DINT	Maximum number of characters received.
ADD	USINT[32]	Buffer containing the UDP address of the transmitter (filled on output).
DATA	STRING	buffer where to store received characters.

OUTPUTS

Name	Type	Description
Q	DINT	number of actually received characters.

REMARKS
If characters are received, the function fills the ADD argument with the internal UDP of the sender. This buffer can then be passed to the udpSendTo function to send the answer.

udpSendTo

FUNCTION
Send a UDP telegram

SYNTAX

```
OK := udpSendTo (SOCK, NB, ADD, DATA);
```

INPUTS

Name	Type	Description
SOCK	DINT	ID of the client socket.
NB	DINT	number of characters to send.
ADD	USINT[32]	buffer containing the UDP address (on input).
DATA	STRING	characters to send.

OUTPUTS

Name	Type	Description
OK	BOOL	TRUE if successful.

REMARKS

The ADD buffer must contain a valid UDP address either constructed by the udpAddrMake function or returned by the udpRcvFrom function.

VLID

FUNCTION

Get the identifier of an embedded list of variables.

INPUTS

Name	Type	Description
FILE	STRING	Pathname of the list file (.SPL or .TXT) - must be a constant value!

OUTPUTS

Name	Type	Description
ID	DINT	ID of the list - to be passed to other blocks.

REMARKS

Some blocks have arguments that refer to a list of variables. For all these blocks, the list argument is materialized by a numerical identifier. This function enables you to get the identifier of a list of variables.

Embedded lists of variables can be:

- Watch lists created with the Workbench. Such files are suffixed with `.SPL`.
- Simple `.TXT` text files with one variable name per line.

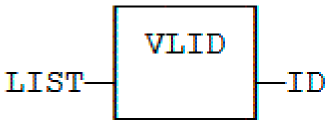
Lists must contain single variables only. Items of arrays and structures must be specified one by one. The length of the list is not limited by the system.

LIST FILES ARE READ AT COMPILING TIME AND ARE EMBEDDED INTO THE DOWNLOADED APPLICATION CODE. THIS IMPLIES THAT A MODIFICATION PERFORMED IN THE LIST FILE AFTER DOWNLOADING WILL NOT BE TAKEN INTO ACCOUNT BY THE APPLICATION.

ST LANGUAGE

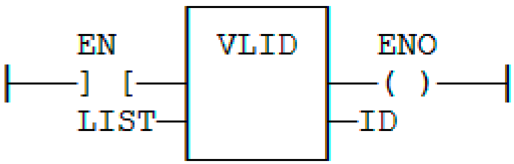
```
ID := VLID ('MyFile.spl');
```

FBD LANGUAGE



LD LANGUAGE

THE FUNCTION IS EXECUTED ONLY IF EN IS TRUE.



IL LANGUAGE

```
Op1: LD   'MyFile.spl'
     VLID COL
     ST   ID
```

T5 Registry for runtime parameters

The T5 Registry enables you to design and monitor remotely a hierarchical registry of parameters. Parameters can be set apart from the Workbench, and can also be read or written from the IEC program.

DESIGNING THE REGISTRY OF PARAMETERS

The Registry Design tool enables you to design the set of runtime parameters and how they will be edited during monitoring. The definition of parameters is stored in an XML file. The design tool works mainly on this file. Additionally, the design tool is used to send the new registry to the runtime in binary form. Optionally the XML file can also be stored in the runtime.

To run the design tool from the Workbench, use the menu command Tools / Runtime Parameters / Design.

Parameters are freely organized with folders. The left part of the editor shows you the complete hierarchy of folders and parameters. The right hand area is used for entering the detailed description of the folder or parameter currently selected in the tree. Use the commands of the Edit menu to add new folders and parameters.

For each folder you must specify the following pieces of information:

Information	Description
Name	Name of the folder - cannot contain “/” characters.
Access	Access mode (read-write or read only).
Protection	Protection mode for write (using password): No = no protection. Yes = password protected. Inherit = use the same protection as the parent folder.
Password	Password for write.
Description	Free description text.

For each parameter you must specify the following pieces of information:

Information	Description
Name	Name of the folder - cannot contain / characters.
Type	Data type.
Max. length	Maximum length for STRING. Maximum lengthy cannot exceed 200 characters.
Access	Access mode (read-write or read only).
Protection	Protection mode for write (using password): No = no protection. Yes = password protected. Inherit = use the same protection as the parent folder.
Password	Password for write.
Default	Default value when registry is loaded for the first time.
Description	Free description text.
Editing mode	Editing method when monitoring.
Extra data	Description of choices for a list or combo box editing mode.
Minimum	Minimum allowed value for numbers.
Maximum	Maximum allowed value for numbers.

The commands of the Project menu are used for updating the runtime:

CHECK REGISTRY

This command checks the whole contents of the designed registry and reports possible consistency errors.

SEND REGISTRY

This command sends the registry in binary format to the runtime system. You normally need to select the communication parameters of a remote runtime, but you can also save the binary registry in a local file if you want to use another transfer method. For remote sending, you can optionally send the XML definition file together with the binary registry.

Depending on the runtime system, it may happen that the new registry is not taken into account immediately, if it is currently in use by the system. Some runtimes will need a Stop/Restart of the IEC application. Some other runtimes may require a full reboot. Refer to the OEM instructions.

MONITORING PARAMETERS

The Register Host tool is used for monitoring runtime parameters On Line. Parameters can be displayed, and possibly modified according to their protection as defined in the Design tool. To run the host tool from the Workbench, use the menu command Tools / Runtime Parameters / Monitor. Then use the File / Open command to connect to the runtime and monitor its parameters.

The left side tree shows the folders of the Registry. The right-side area shows the parameters of the selected folder. Double click on a parameter to change its value.

Use the View / Refresh command to refresh the value of the parameters.

The File / Save menu command asks the runtime system to save the contents of the registry to backup support (flash or disk).

T5 Registry Management Functions

The T5 Registry enables you to design and monitor remotely a hierarchical registry of parameters. Parameters can be set apart from the Workbench, and can also be read or written from the IEC program.

The following functions are available:

Function	Description
RegParGet	Get the current value of a parameter.
RegParPut	Change the value of a parameter.

PARAMETER PATHNAMES

Any parameter is specified by a full qualified pathname that gives its exact location in the registry. The / separator is used to separate folders in the pathname. For a registry defined as:

```
(Root)
   |
   └─Folder1
        |
        └─Folder2
             |
             └─MyParam
```

The pathname of the parameter will be:

```
/Folder1/Folder2/MyParam
```

T5 REGISTRY MAY BE NOT AVAILABLE ON SOME TARGETS. PLEASE REFER TO OEM INSTRUCTIONS FOR FURTHER DETAILS ABOUT AVAILABLE FEATURES.

ALL PARAMETER PATHNAMES ARE CASE SENSITIVE.

RegParGet

FUNCTION

Get the current value of a parameter

SYNTAX

```
Q := RegParGet (PATH, DEF);
```

INPUTS

Name	Type	Description
PATH	STRING	specifies the full pathname of the parameter in the registry.
DEF	ANY	default value to be returned if the parameter does not exist.

OUTPUTS

Name	Type	Description
Q	ANY	current value of the parameter.

THE DEF INPUT DEFINES THE ACTUAL TYPE FOR ANY PINS. IF YOU SPECIFY A CONSTANT EXPRESSION, IT MUST BE FULLY TYPE-QUALIFIED ACCORDING TO THE WISHED RETURNED VALUE. EXAMPLE FOR GETTING A PARAMETER AS AN INT:

INTVARIABLE := REGPARGET ('\MYPARAM', INT#0);

ALL PATHNAMES ARE CASE SENSITIVE.

RegParPut

FUNCTION
Change the value of a parameter

SYNTAX

```
OK := RegParPut (PATH, IN);
```

INPUTS

Name	Type	Description
PATH	STRING	specifies the full pathname of the parameter in the registry.
IN	ANY	new value for the parameter.

OUTPUTS

Name	Type	Description
OK	BOOL	TRUE if successful.

The function will returned **FALSE** in the following cases:

- The specified pathname is not found in the registry.
- The registry is currently being saved and cannot be changed.

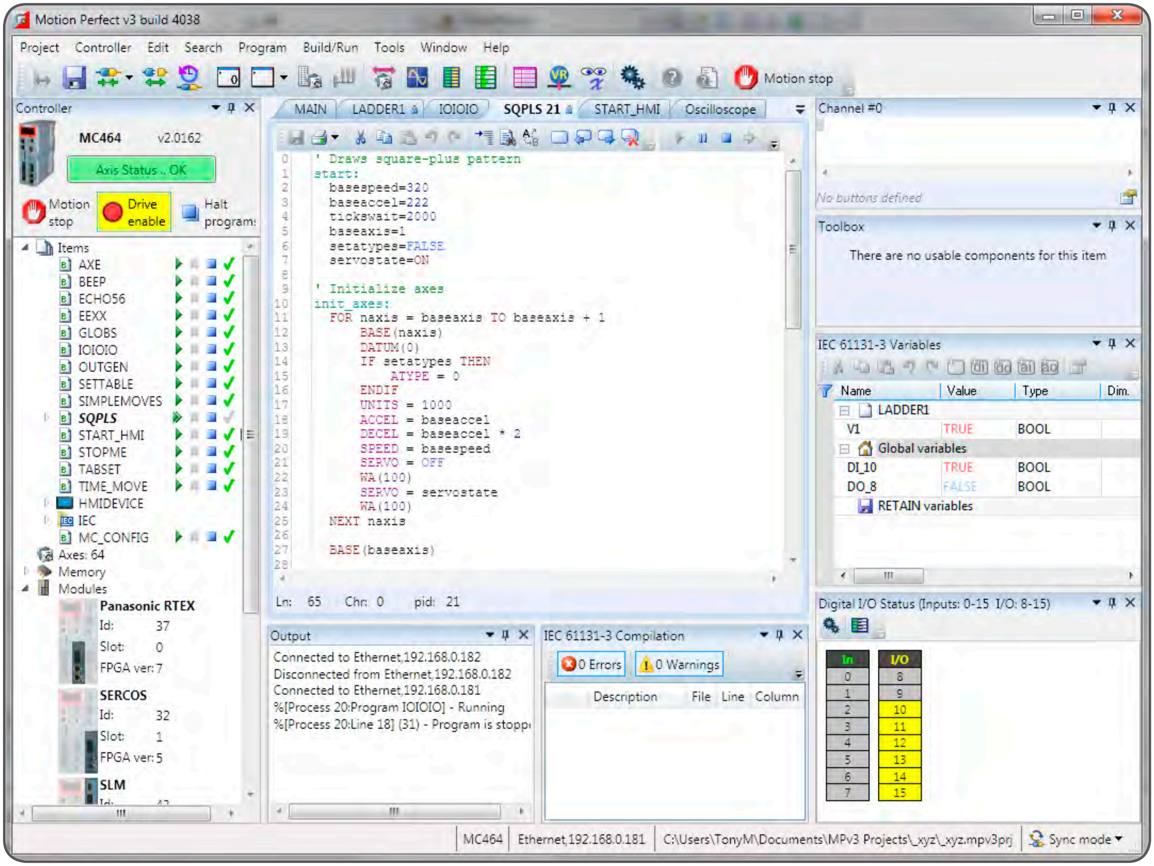
ALL PATHNAMES ARE CASE SENSITIVE.

MOTION PERFECT V3

5

Introduction to *Motion* Perfect 3

Motion Perfect 3 is an Microsoft Windows™ based application for the PC, designed to be used in conjunction with Trio Motion Technology's Series 4 *Motion Coordinator* range of multi-tasking motion controllers.



Motion Perfect 3 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

PC

A PC with the following specifications is required to run *Motion* Perfect 3:

	Minimum	**Recommended**
Operating System	Windows XP, SP 3	Windows 7
.NET Library	3.5	3.5
Processor		
RAM	2MBytes	4MBytes
Hard Disk Space	50MBytes + space for projects	200MBytes

Due to limitations in some of the third party libraries used, *Motion* Perfect 3 is only available as a 32 bit application. This will however run on 64 bit Microsoft Windows™.

It is recommended that your copy of Microsoft Windows™ has all current service packs and updates applied.

CONTROLLER

The requirements for a controller are different depending on the mode of connection.

DIRECT MODE

To connect in Direct Mode the controller can be almost any Trio series 2, 3 or 4 *Motion Coordinator*.

TOOL MODE / SYNC MODE

To connect in Tool Mode or Sync Mode the controller must be a Trio series 4 *Motion Coordinator* running system firmware version 2.0177 or later.

Operating Modes

Motion Perfect 3 has four operating modes:

- Disconnected
- Direct
- Tool Mode
- Sync Mode

The current connection mode is displayed on the right of the status bar at the bottom of *Motion* Perfect's main window.

DISCONNECTED

Not connected to a controller. All tools are closed and no communications ports are open.

DIRECT MODE

A direct connection is made to a controller allowing a Terminal tool to be used for direct interaction with the command line on the controller.

TOOL MODE

A multichannel connection is made to a controller allowing the monitoring tools within *Motion* Perfect to be used. This mode allows the user to see a list of the programs on the controller (so that they can be started and stopped) but does not allow editing of any of the programs.

SYNC MODE

A multichannel connection is made to a controller and a local project on the PC is opened. The contents of the controller and the project are synchronized so that the local copy of all programs matches those on the controller. All of *Motion* Perfect's tools are available and programs can be edited. The synchronization process can involve deleting programs or copying them from the controller to the PC of vice versa.

A connection (direct or multichannel) to a controller consists of a single TCP/IP socket connection over Ethernet.

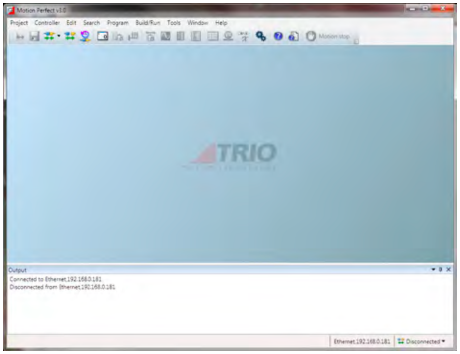
Main Window

The "Main Window" is the main user interface of *Motion* Perfect 3. It acts as a desktop for displaying all controls needed to interact with a single controller.

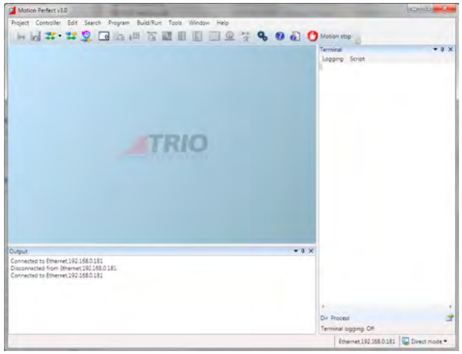
Because the tools available to the user are different for each operating mode the Main Window tends to take on a different appearance for each mode.

In all operation modes the user has access to the Main Menu and Main Toolbar for commands, although the commands available will depend on the operation mode.

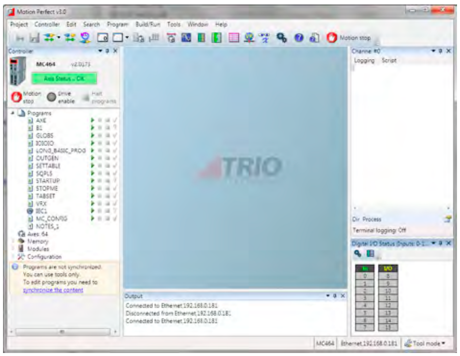
Disconnected Mode



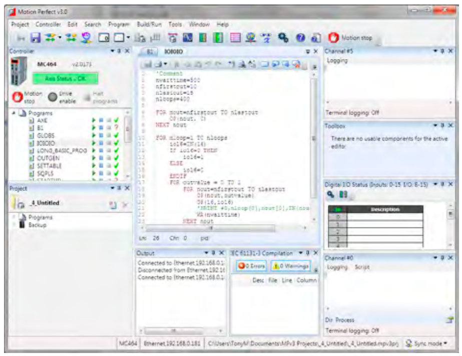
Direct Mode



Tool Mode



Sync Mode



Main Menu

The Main Menu has a set of sub-menus which splits the menu commands into functional groups as follows

PROJECT

New	Create a new project and erase any controller content
Load	Load an existing project onto the controller
Change	Change to a different project and reconcile with the existing controller contents
Create from Controller	Create a new project from the existing controller contents
Save	Save the current project (flushes all changes to disk)
Save As	Save the current project under a different name
Export	Export the project in a different format
Project Check	Check the current project against the controller contents
Create Backup	Create a backup copy of the current project
Backup	Open the “Backup Manager” tool to create or manage project backups
Close	Close the current project (this results in the connection changing to Tool Mode)
Modify STARTUP program	Modifies the STARTUP program
Recent Projects	Allows easy working with recently used projects
Solution Manager	Opens the solution manager to allow working with more than one controller
Print	Prints the current active editing session
Exit	Exits from the application

CONTROLLER

Connect in Sync Mode	Connect to the controller in Sync Mode
Connect in Tool Mode	Connect to the controller in Tool Mode
Connect In Direct Mode	Connect to the controller in Direct Mode
Disconnect	Disconnect from the controller
Connection Settings	Change the connections settings used for the communicating with the controller
Reset Controller	Reset the controller by performing a warm restart
CANIO status	View the CANIO status (not implemented)
Interfaces	Open the sub-menu which allows the configuration of all communications interfaces on the controller.
Enable Features	Enable and disable soft features

Memory Card	Open the “Memory Card Manager” to manipulate the contents of the memory card in the controller.
Load Firmware	Load new system firmware
Directory	Shows an extended directory listing of the programs on the controller
Processes	Shows a list of all user processes currently running on the controller
Lock Controller	Lock the controller using a locking code
Unlock Controller	Unlock a locked controller
Date and Time	Sets the real-time clock on the controller using the “Date and Time” tool

EDIT

Undo	Undo the last editing operation
Redo	Redo the last undone editing operation
Cut	Cut the currently selected text into the clipboard
Copy	Copy the currently selected text into the clipboard
Paste	Paste test from the clipboard
Select All	Select all text in the document
Select None	Deselect the current selection
Delete	Delete the currently selected text
TrioBASIC	Open the TrioBASIC sub-menu which gives access to reformatting and auto-commenting operations.

SEARCH

All search commands apply to the current active editing session

Find	Search for a text string
Find Next	Find the next occurrence of the last search string
Find Prev	Find the previous occurrence of the last search string
Find Next Occurrence Current Selection	Find the next occurrence of the currently selected text string
Find Prev Occurrence Current Selection	Find the previous occurrence of the currently selected text string
Replace	Replace one text string with another
Toggle Bookmark	Toggle a bookmark on the current line
Goto Next Bookmark	Go to the next bookmark
Goto Prev Bookmark	Go to the previous bookmark
Goto Line/Label	Go to a line or label

Match Scope	Go to the end / beginning of the scope started / ended on the current line

PROGRAM

New	Create a new empty program (see "Creating a New Program")
Load	Load an existing program and add to the current project
Edit	Edit a program in the current project
Debug	Debug a program in the current project
Save	Save current program to disk (only available if there are unsaved changes).
Copy	Copy a program in the current project
Rename	Rename a program in the current project
Delete	Delete a program in the current project
Delete All	Delete all programs in the current project
Compile All	Compile all programs in the current project
Set Autorun	Set the Autorun process of a program in the current project
Run Autorun programs	Run all programs set to autorun
Stop All (Halt)	Stop all running programs
IEC 61131-3	

BUILD/RUN

The commands I n this sub-menu operate on the program open in the current active editing session.

Compile	Compile the program (any changes are saved first)
Run	Run the program
Step	Step the program
Step In	Step program into a function or subroutine
Step Out	Step program out of a function or subroutine
Pause	Pause program execution
Stop	Stop program execution
Toggle Breakpoint	Toggle breakpoint on the current line
Enable/Disable Breakpoint	Toggles the enabled state of the breakpoint on the current line
Breakpoints	Opens a dialog to display all current breakpoints
Watch Variable	Add a watch for the currently selected variable
Set Autorun	Set the Autorun process number

The availability of the commands in the Build/Run sub-menu depends on the type or program being edited and the run state of the program.

TOOLS

Axis Parameters	View and modify axis parameters using the "Axis Parameters" tool
Intelligent Drives	Configure intelligent drives attached to the controller. This is to be implemented using add-ons (at present none are available).
Oscilloscope	A software Oscilloscope tool which can be used to show traces of how parameters vary with time
Digital I/O	View the states of digital inputs and outputs and change the state of digital outputs using the "Digital I/O Viewer" tool
Jog Axes	Manually jog axis positions using the "Jog Axes" tool
Table Viewer	View and change table data values using the "Table Viewer" tool
VR viewer	View and change VR variable data values using the "VR Viewer" tool
Watch Variables	View and change the values of local and global variables whilst debugging using the "Variable Watch" tool
Analogue Inputs	View the status of analogue inputs using the "Analogue I/O Viewer" tool
Terminal	Open a Terminal Tool to interact with the controller
Diagnostics	Configure Diagnostics for fault finding
Options	Change the Options for *Motion* Perfect and its tools

WINDOW

Toolbar	Show / hide the main toolbar
Status Bar	Show hide the application status bar
Output Window	Show / hide the Output Window
Controller Tree Window	Show / hide the Controller Tree Window
Project Tree Window	Show / hide the Project Tree Window
Toolbox	Show / hide the Toolbox
Show Recent Work	Show the Recent Work dialog
Clear Output Window	Clear the Output Window
Close Window	Close the current window
Reset Window Layout	Reset the window layout to the default layout

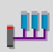
HELP

Motion Perfect v3 Help	Displays *Motion* Perfect help

TrioBASIC Help	Displays TrioBASIC language help
About *Motion* Perfect v3	Displays the *Motion* Perfect About Box which shows software versions.

Main Toolbar

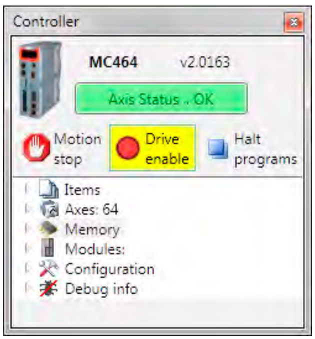
The Main Toolbar gives the user quick access to Motion Perfect's main tools and functions.

Icon	Command	Operation
	Open Project	Opens a project and synchronizes with the controller contents
	Save Project	Saves the current project to disk (Sync Mode only)
	Connect	Opens up a sub-menu with options to connect in Sync Mode, Tool Mode or Direct Mode
	Disconnect	Disconnects
	Recent Work	Opens the "Recent Work dialog" Which allows reconnection to recently used connections or opening of recently used projects.
	Terminal (channel 0)	Opens a Terminal tool on Channel 0 if in Tool or Sync Mode or directly connected to the command line if connected in Direct Mode
	Terminal	Opens a Terminal on a user selectable channel when connected in Tool or Sync Mode
	Axis Parameters	Opens the Axis Parameters Tool (Tool and Sync Modes only)
	Intelligent Drives	Allows the user to configure Intelligent Drives (Sync Mode only, depends on installed add-ons)
	Jog Axes	Opens the Jog Axis Tool (Tool and Sync Modes only)
	Oscilloscope	Opens the Oscilloscope Tool (Tool and Sync Modes only)
	Digital I/O	Opens the Digital I/O Viewer Tool (Tool and Sync Modes only)

Icon	Command	Operation
	Analogue I/O	Opens the Analogue Input Viewer Tool (Tool and Sync Modes only)
	TABLE Viewer	Opens the TABLE Viewer Tool (Tool and Sync Modes only)
	VR Viewer	Opens the VR Viewer Tool (Tool and Sync Modes only)
	Variable Watch	Opens the Variable Watch Tool (Tool and Sync Modes only)
	Options	Opens the main Options dialog
	Motion Perfect Help	Displays help for *Motion* Perfect
	TrioBASIC Help	Displays help for the TrioBASIC language
	IEC 61131-3 Help	Displays help foe IEC 61131-3 programming

Controller Tree

The controller tree can be displayed when *Motion* Perfect is operating in “Tool Mode” or in “Sync Mode”. It contains information about the controller connected to *Motion* Perfect and its contents.



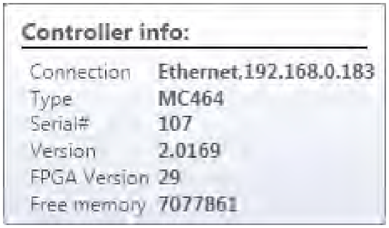
The tree consists of a header section and the tree body.

TREE HEADER

The tree header contains basic information about the controller plus some important controls. The top of the header contains a pictorial representation of the controller, the controller model (MC464 in the case above), the system software version number and an “Axis Status” control. The bottom of the header contains three button controls: “Motion Stop”, “Drive Enable” and “Halt Programs”

CONTROLLER INFORMATION

The controller is shown as an icon to the left of the header. The controller model and system software version are displayed towards the top of the header. If the mouse cursor is moved over the icon a tooltip is displayed giving some basic information about the controller.



“AXIS STATUS” CONTROL

This control shows the error status of the controller. It is a passive control when there is no error and is coloured green. When an error occurs the control becomes coloured red and then acts as a button which, when clicked, will clear the error on the controller.

Some errors, notably hardware errors, cannot be cleared by clicking the “Axis Status” button.

“MOTION STOP” BUTTON

Clicking on the “Motion Stop” button stops all currently running programs and empties all the move buffers on the controller causing all motion to stop. Its action is similar to an “Emergency Stop” button but, as it is implemented in software, it is less reliable that a properly implemented hardware emergency stop.

It is important that a proper hardware emergency stop is implemented on any system. This button must not be used as a substitute.

“DRIVE ENABLE” BUTTON

Clicking on this button toggles the state of the drive enable (watchdog output) on the controller. When drives are enabled the background of the button is coloured yellow.

“HALT PROGRAMS” BUTTON

Clicking on this button halts all currently running programs but does not stop and current or buffered moves.

Use the “Motion Stop” button if you want to stop the motion as well as the programs.

TREE BODY

The body of the tree contains information in several expandable sections:

Section Name	Contents
Programs	Programs and files stored on the controller.
Axes: (Max Axes)	A list if the Axes defined as visible.
Memory	Memory related information.
Modules	Interface modules connected to the controller.
Configuration	Controller configuration information.

PROGRAMS

These are the programs and files stored on the controller. The following types of item can be stored on the controller:

- TrioBASIC program
- Text file
- MC _ **CONFIG** program (one only)
- **HMI** project (not available on all controllers) containing one or more **HMI** page definitions.
- IEC 61131-3 project (not available on all controllers) containing one or more programs in one or more of the IEC 61131-3 defined program types.

The “Programs” item in the tree has a context menu to allow creation of programs and some operations on all programs as follows:

Menu Entry	Operation
New	Create a new empty program (see “Creating a New Program”)
Import...	Import a program
Compile all	Compile all compilable programs
Stop all (Halt)	Stop all running programs
Delete all programs	Delete all programs

The program entries in the tree allow the user to run, pause, stop and compile the program by means of a set if icons after each program entry.

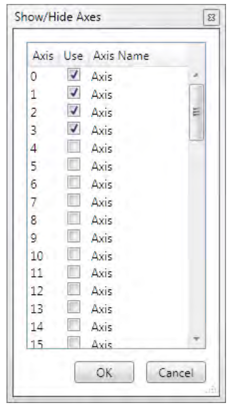
When a program is running it has an extra entry in the tree representing the running instance, showing the process number.

Icon	Operation	Notes
	Run	Run the program. Also run a paused instance from its current (paused) position.
	Run another instance	Run another instance of a program on a different process from currently running instance(s)
	Pause	Pause running program or step non-running program to first line
	Step	Step program onto next line
	Stop	Only available when program is running
	Compile	Icon shows that the program is not compiled.
	Compile	Icon shows that the program is already compiled. Not available when program is running

AXES: (MAX AXES)

The value of Max Axes is the total number of axes available on the controller, both real and virtual.

When expanded the list of axes shown is that specified by the user. To specify which axes are to be shown, right click on axes and select "Shaw/Hide Axes..." to display the "Show/Hide Axes" dialog and select which axes to display.



MEMORY

This shows various memory related items as follows:

VR

The maximum number of VR variables allowed. Double clicking on this launches the VR Viewer tool.

TABLE

The size (in values) of the **TABLE** memory area. Double clicking on this launches the Table Viewer Tool.

LOCAL VARIABLES

Double clicking on this launcher the variable viewer tool.

GLOBALS

Currently not used.

FREE PROGRAM SPACE

The number of bytes of unused memory available for storing programs in.

MODULES

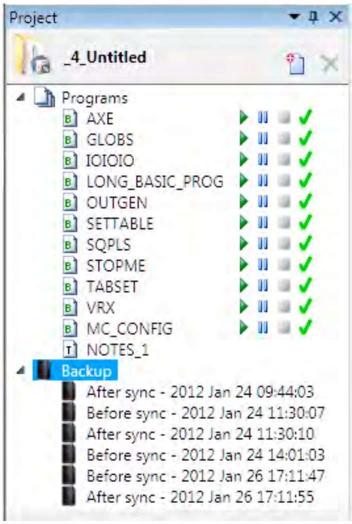
This give a list of the modules connected to a controller. Currently this only supports the local modules of a modular controller such as the MC464.

CONFIGURATION

Shows the current controller configuration and allows the user to change some user configurable features.

Project Tree

The project tree can be displayed when *Motion* Perfect is operating in “Sync Mode”. It contains information about the current project *Motion* Perfect.



The tree consists of a header section and the tree body.

TREE HEADER

The tree header contains basic information about the project plus some important controls. The header contains a project icon, the project name, a “New Program” button and a “Delete Item” button.

“MOTION STOP” BUTTON

Clicking on the “Motion Stop” button stops all currently running programs and empties all the move buffers on the controller causing all motion to stop. Its action is similar to an “Emergency Stop” button but, as it is implemented in software, it is less reliable that a properly implemented hardware emergency stop.

It is important that a proper hardware emergency stop is implemented on any system. This button must not be used as a substitute.

"NEW PROGRAM" BUTTON
Clicking on this button creates a new program in the project. (See "Creating a New Program")

"DELETE ITEM" BUTTON
Clicking on this button deletes the currently selected program.

TREE BODY
The body of the tree contains information in several expandable sections:

Section Name	Contents
Programs	Programs and files stored in the project.
Backup	Automatically and manually created backups of the project.
Settings	User changeable settings of the project.

PROGRAMS
This section duplicates the functionality of the "Programs" section in the "Controller Tree"

BACKUPS
Every time *Motion* Perfect synchronizes with a project a backup of the project is made before and after the synchronization operation (the backup after is only made if synchronization has been successful). The tree contains a list of the backups currently stored on the PC.

The "Backups" item in the tree has a context menu as follows:

Entry	Description
Create Backup	Create a backup of the current state of the project
Delete All Backups	Delete all the stored backups
Manage	Start the "Backup Manager" tool

Each backup entry also has a context menu as follows:

Entry	Description
Revert to Selected Backup	Reverts the project to the state saved in the selected backup
Set Name	Allows the user to give the backup a meaningful name
Delete Backup	Deletes the backup entry

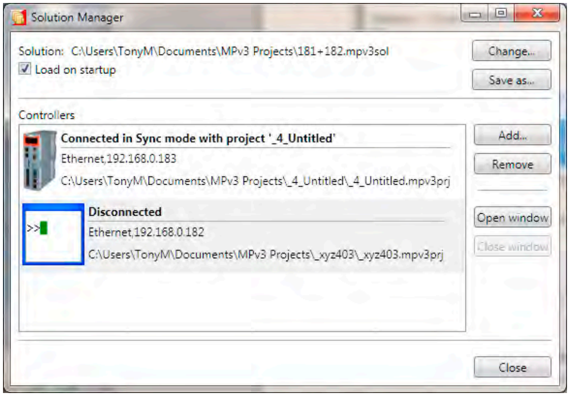
Output Window

The "Output Window" displays the status messages received from the controller.

Solutions

In order to handle systems which contain more than one controller *Motion* Perfect uses a “Solution” to manage the connections to more than one controller and their associated projects. The solution defines a list of controllers included in the solution. For each controller it also defines a connection used to communicate with the controller and a project associated with it. No two controllers can be associated with the same project. The user can create and edit a solution using the Solution Manager.

SOLUTION MANAGER



The Solution Manager is used to manage a collection of projects (solution) which are used for applications containing multiple controllers. In single applications which contain only one project, *Motion* Perfect uses a default solution so that the user does not need to use the solution manager.

The default solution cannot contain more than one project.

CONTROLS

LOAD ON STARTUP CHECKBOX

If checked, the solution manager and the current solution will be loaded when *Motion* Perfect is started.

CHANGE SOLUTION BUTTON

Change to a different solution.

SAVE SOLUTION AS BUTTON

Save the current solution under a new name

ADD CONTROLLER BUTTON

Add a controller (connection) to the solution.

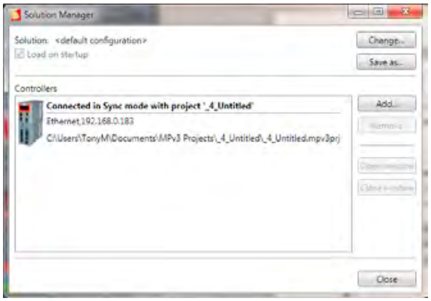
REMOVE CONTROLLER BUTTON
Remove the currently selected controller (connection) from the solution

OPEN WINDOW BUTTON
Open a window for the currently selected controller

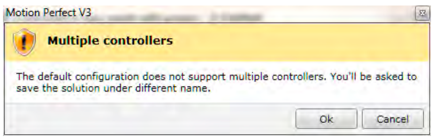
CLOSE WINDOW BUTTON
Close the open window for the currently selected controller

CLOSE BUTTON
Close the “Solution Manager” window

CREATING A SOLUTION
- Create a project for one controller as normal.
- Open the “Solution Manager” from the Project section of the main menu. This will display the existing project as part of the “Default Solution”.



- Click on the “Add” button. A warning about multiple controllers will be displayed.



- Clicking on the “**OK**” button will cause the “Connection Dialog” to be displayed. Configure an appropriate connection for another controller. On closing the “Connection Dialog” you will be prompted to save the solution. A desktop window will appear for the connection to the new controller.
- To associate a project with the new controller, attempt to connect to it in Sync Mode (this may happen automatically depending on the stored state of the connection). The “Controller Project Dialog” will be displayed to allow this.

Project

A *Motion* Perfect project contains a set of programs and settings which represents the contents of the controller for a given application. Al files relating to a project are stored in a single directory on the PC this is known as the **project directory**.

PROJECT DIRECTORY

The files contained in the **project directory** will depend on the programs used in the project. There are three main files in the **project directory** which all have the same name as the project directory but have different file extensions.

PROJECT FILE (EXTENSION "MPV3PRJ")

This contains a definition of the contents of the project (programs) and any customization such as axis names.

DESKTOP FILE (EXTENSION "MPV3DSK")

This contains the desktop layout used when *Motion* Perfect is connected in sync mode to the controller.

TOOL INTERNAL CONDITIONS (EXTENSION "MPV3IC")

This contains the internal state of each open tool window when *Motion* Perfect is connected in sync mode to the controller.

PROGRAM FILES

Program files are also stored in the project directory. The type of each file can be determined by its file extension the most important being .BAS which is used for TioBASIC programs. Each TrioBASIC program may also have a .PRG file of the same name which specifies editor/debugger settings for the program. Some complex types of program (usually handled by an add-in) can have sub-directories which contain their data as well as one or more files in the project directory.

There is also a "Backup" sub-directory in which backups of the project are stored.

WARNING

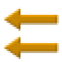
Although many of the files which form part of the project are text files the user should not edit them directly using a text editor as this may cause compatibility problems between the project and the controller. All changes should be made using *Motion* Perfect.

Project Check

A project check is performed every time *Motion* Perfect connects in "Sync Mode" and if the user initiates a project check from the main menu. The programs in the project are checked against those on the controller and if there are any differences the "Resolve Program Differences" dialog is displayed so the user can resolve the differences.

RESOLVING DIFFERENCES

The “Resolve Program Differences” dialog can perform several different operations to resolve differences.

Icon	Operation
	Change the project
	Create a new empty project
	Make the contents of the project the same as that in the controller
	Make the contents of the controller the same as that in the project
	Copy a program from the controller to the project
	Copy a program from the project to the controller
	Delete a program (from the project or controller or both)
	Use a “Resolve Differences” tool to examine the differences between the copy of a program on the controller and the one in the project and optionally to make changes to the file in the project (which will then be loaded onto the controller).

The synchronization operation is carried out when the user clicks on the “Synchronize” button which is only enabled

Once a set of operations has been selected which will resolve all differences.

The synchronization operations available depend on the types of program in the project and on the controller.

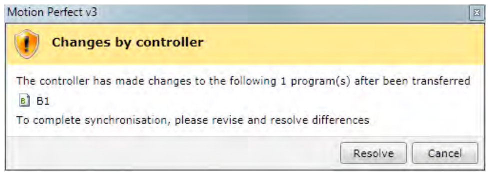
It is possible that a program copied from the project onto the controller will still cause a project check failure if the controller supports different keywords to those supported by the controller on which the program was written. This problem can be resolved by saving the copy on the controller into the project or manually resolving the differences.

PROBLEMS LOADING PROGRAMS

Even though it appears that differences can be resolved by loading the project or some of its programs onto the controller it is still possible to get a mismatch between the controller and the project. This is usually due to different TrioBASIC keywords being supported on the controller to those supported on the controller on which the program was written. This can cause variables to become keywords, keywords to become variables or keywords to change.

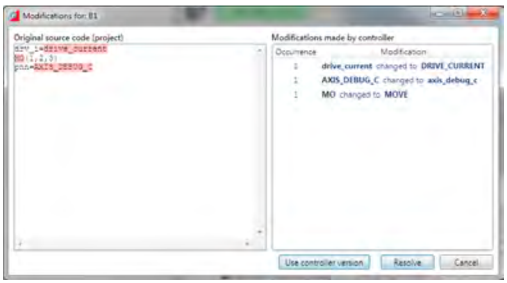
All the letters in a keyword are always upper case whereas all the letters in a variable name are always lower case.

When this occurs a warning dialog will be displayed to show that the controller has made changes to the program.



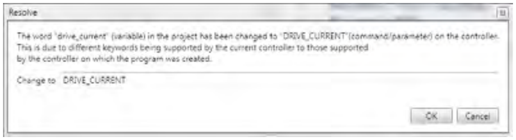
The user now has the choice of resolving the differences using the program modifications dialog or cancelling. If you cancel it is then possible to resolve differences by doing another project check and manually resolving the differences using the "Resolve Differences" tool.

MODIFICATIONS DIALOG



This shows the original program source (on the PC) on the left and the changes made to it on the right. The user can resolve the differences by either using the controller version of the program or by clicking on the "Resolve" button which steps through the differences to allow the used to make a decision for each one using the "Resolve" dialog.

RESOLVE DIALOG



The new value for the word to resolve is automatically filled in using the value obtained from the controller. The user can type any valid keyword, variable name, or number to replace the word in the source file. Clicking on “OK” makes the change and clicking on “Cancel” cancels the whole resolution process.

Program Types

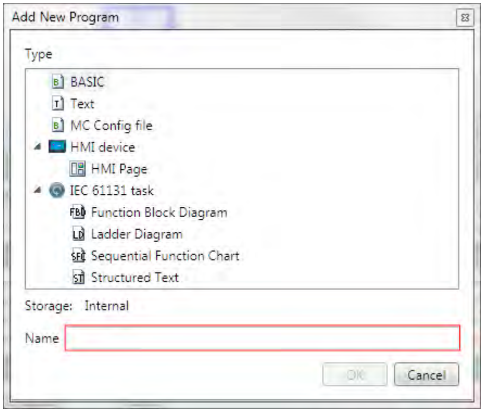
Motion Perfect supports several different program types as follows:

Icon	Type	Note
B	TrioBASIC	
B	Encrypted TrioBASIC	This type of file can only be written to a controller, it cannot be read. It is produced by encrypting an normal TrioBASIC program.
T	Text	This is textural information stored on the controller and does not represent a runnable program.
	IEC Task	Consists of one or more of the EIC program types below.
LD	IEC Ladder Diagram	
ST	IEC Structured Text	
FBD	IEC Function Block Diagram	
SFC	IEC Sequential Function Chart	

Creating a New Program

A new program can be created by Selecting “Program / New” from the main menu or by selecting “New” from the “Programs” item in the controller menu.

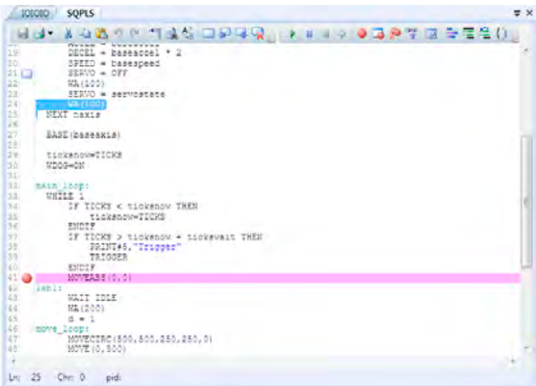
The “New Program” dialog is launched. This allows the user to select the type of program required and enter a name. Clicking on “OK” will create the new program.



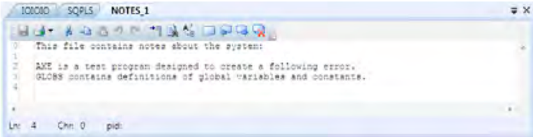
This is only available while connected in Sync Mode.

Program Editor

The Program Editor is used to edit TrioBASIC program files and text files which form part of a *Motion* Perfect project and to provide debugging facilities for TrioBASIC programs.



Editing a Trio**BASIC *program***



Editing a text file

The editor performs in a similar way to most modern text editors. Editing functions are available for all supported program/file types, debugging functions and special formatting functions are only available when editing a TrioBASIC program.

EDITING FUNCTIONS

Editing functions are available from the Edit Toolbar:

The available editing functions are as follows and apply to the current program/file being edited:

Save to disk
Print
Cut selected text to clipboard
Copy selected text to clipboard
Paste text from clipboard
Undo last operation
Redo last undone operation
Go to line or label
Find text
Replace text
Toggle bookmark on current line
Go to previous bookmark
Go to next bookmark
Clear all bookmarks

Some editing functions are available on the Editor Context Menu.

DEBUGGING FUNCTIONS

Debugging functions are available from the Debug Toolbar.

The available debugging functions are as follows and apply to the current program being edited:

Run

Pause/Step

Stop

Go to current execution line (when stepping program)

Toggle breakpoint on current line

Show all breakpoints

Remove all breakpoints

Watch variable

Compile program

Auto-format text

Comment out selected lines

Un-comment selected lines

Go to end/start of scope (program structure) which starts/ends on the current line

Some debugging functions are available on the Editor Context Menu.

OPERATION

Although the editor appears to work like any other text editor it has one main difference. Each line of text is sent to the connected controller as it is entered or edited. This means that the controller is always kept up to date with changes. The controller is used to perform syntax checking when editing a TrioBASIC program, removing any possibility that the syntax is checked against out of date rules. All compiling and debugging operations are also carried out on the actual controller.

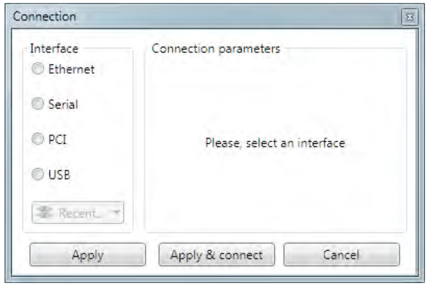
The general appearance of the editor can be customized using the Program Editor pages in the main Options Dialog.

WATCHING VARIABLES

The values of variables can be watched while a program is running or being stepped. This is done using the “Watch Variables” tool, which can be used to monitor both local and VR variables.

To add a variable to the watch list, select the variable name (including index if a VR) in the editor, then select “Watch Variable” from the context menu or click on the icon in the editor toolbar. Alternatively, if the “Watch Variables” tool is open, select the variable name then drag and drop it into the “Watch Variables” tool.

Connection Dialogue

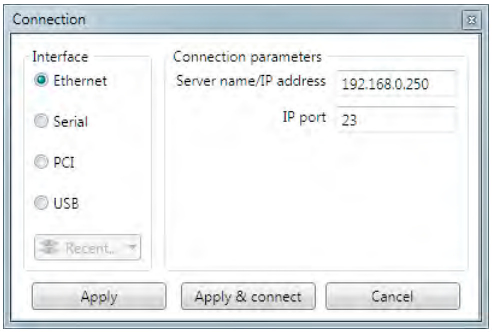


The connection dialog allows the user to configure a communications interface in order to connect to a controller. Ethernet, Serial, PCI and USB interfaces are supported by *Motion* Perfect. It is possible to select a communications interface and configure it manually or choose from recently used connections.

RECENT CONNECTIONS

To choose a recent connection, click on the “Recent” button and choose a connection from the drop-down list.

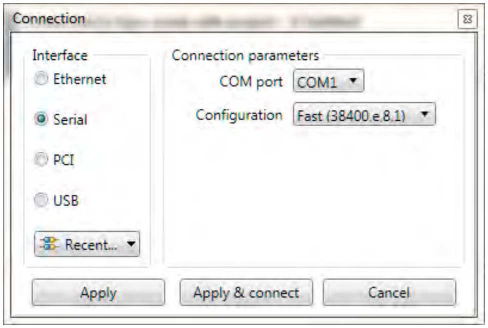
ETHERNET



It is possible to change the server IP address (IP address of the controller) and the IP port on which it communicates.

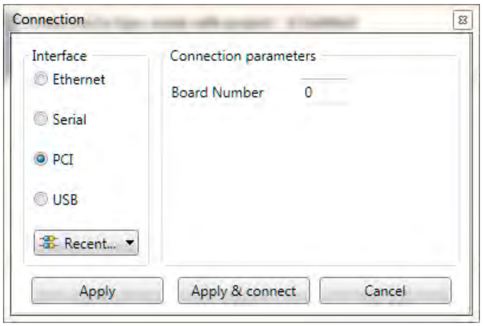
By default a controller will expect a connection from *Motion* Perfect to be made on port 23.

SERIAL



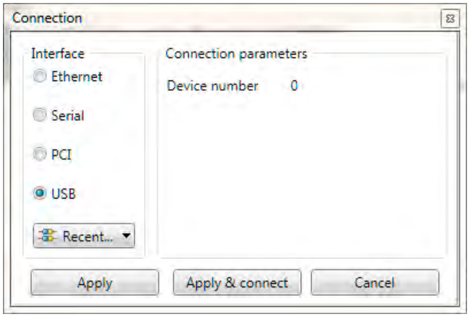
It is possible to select the COM interface and the configuration (serial link parameters) from a choice of Slow (9600,e,7,2) and Fast (38400,e,8,1), these being the default settings for series 2 & 3 Trio *Motion Coordinator*s.

PCI



It is possible to select the board number. Board numbers are allocated when the PC is started up and is enumerated between 0 and the one less than the number of Trio PCI cards connected.

USB



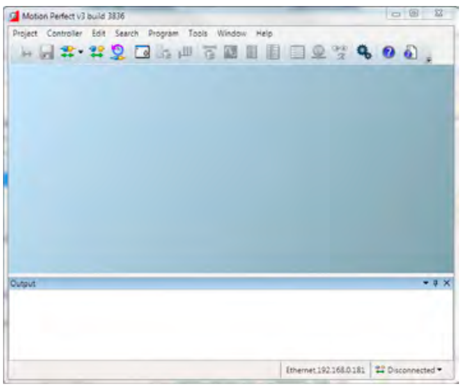
It is possible to select the device number. Device numbers are allocated when the PC is started up and when devices are added or removed. It is normally enumerated between 0 and the one less than the number of Trio USB devices connected. Because of the nature of the internal scanning process which enumerates USB devices and the possibility that devices are added or removed after the initial scan has completed, a given device may not always have the same device number.

It is recommended that only one Trio USB device be connected to a PC at any one time.

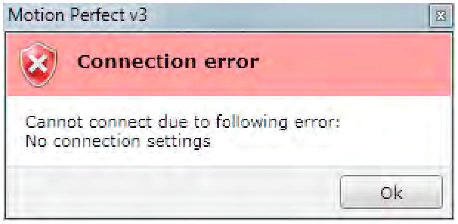
Initial Connection

To make the initial connection to a controller:

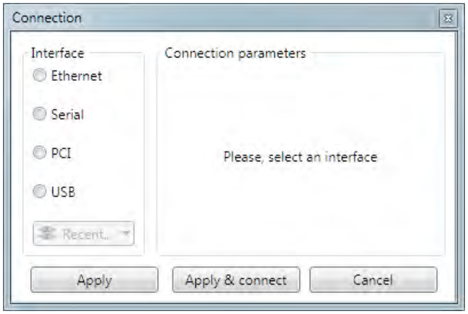
1. Make sure that your controller is powered up and connected to the computer
2. Start *Motion* Perfect 3. Once it has started up the initial screen should be displayed.



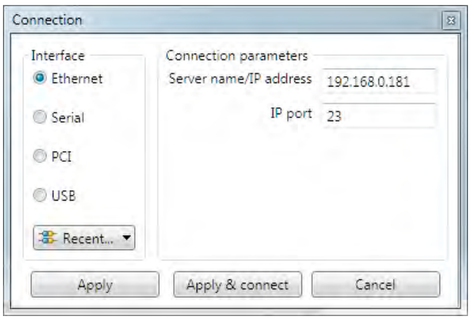
3. Select “ Connect in Direct mode” from the “Controller” menu. As *Motion* Perfect has not been connected before the “Connection Error” dialog will be displayed.



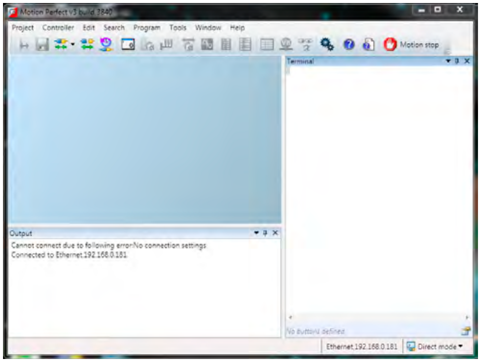
4. Click on the “**OK**” button. The “Connection” dialog will then be displayed.



5. Select the communications interface used by your controller (this will usualyl be Ethernet), then enter it’s parameters. For an Ethernet connection this will be the **IP** address (defailt 192.168.0.250) and the **TCP** port (default 23).



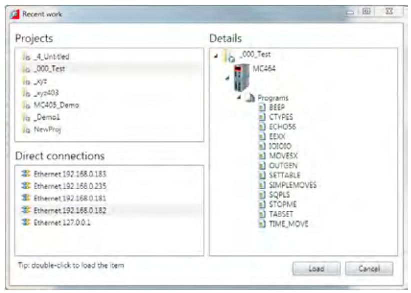
6. Click on the “Apply & Connect” button. The “Connect” will close and *Motion* Perfect will go into Direct Mode with an active Terminal tool.



Motion Perfect will remember the last used connection parameters and will automatically try and use them when reconnecting in Direct Mode in the future.

Recent Work Dialogue

The “Recent Work Dialog” lists recently used projects and connections to allow the user to quickly switch to a different, recently used, project or connection. When a project is selected the “Details” pane on the right of the dialog shows the contents of the project, otherwise, if a connection is selected it shows connection details. Clicking on the load button will load the selected project or connect using the selected connection.



Tools

Motion Perfect 3 has several tools which are used to monitor the controller and interact with it. Some tools are built into *Motion* Perfect, others are implemented as add-ons. The add-on mechanism allows the easy addition of extra tools in the future. Most tools are available in both “Tool Mode” and “Sync Mode”.

BUILT-IN TOOLS

Terminal – direct interaction with the controller’s command line and character I/O

Axis Parameters – view and change the control parameters for each axis

Digital I/O Viewer – view and change digital I/O values

Analogue I/O Viewer – view and change analogue I/O values

Table Viewer – view and change values in **TABLE** memory

VR Viewer – view and change global VR variables

Variable Watch – view and change program internal variables

Options - change the configuration options for *Motion* Perfect

Diagnostics – enable and disable diagnostic functions

Jog Axes - manually jog the control axes

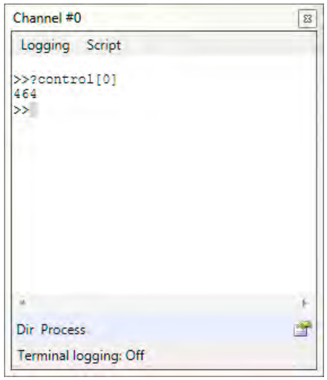
ADD-ON TOOLS

Oscilloscope – capture and view parameters graphically

Intelligent Drives – configure intelligent drives

Terminal

The “Terminal” tool allows the user to interact directly with the controller, either with the command line (channel 0) or with user programs (channel 5, 6 or 7). Characters typed on the keyboard are sent to the controller and characters output by the controller are displayed in the terminal window.



TERMINAL MENU

The menu controls terminal logging and scripting.

TERMINAL LOGGING

When logging is active all the data displayed on the terminal is also written to a file. The name of the log file is displayed in the status bar at the bottom of the terminal window.

TERMINAL SCRIPTING (ONLY AVAILABLE ON CHANNEL 0)

INTRODUCTION

Motion Perfect has built in support for simple terminal scripting. This allows the user to write files of commands and then send the file contents to the controller in a single operation. In addition to the commands to be sent to the controller there are some extra commands which are used by Motion Perfect to control the running of the script.

INTERACTION WITH THE CONTROLLER

Command lines are sent to the controller one at a time in sequence. Motion Perfect sends a command then waits to receive a prompt (>>) before sending the next one.

To not wait for a prompt put the two character sequence \& on the end of the line. These extra characters are not sent to the controller.

SCRIPT COMMANDS

Script commands control the running of the script. All script commands start with two colons. The following commands are valid:

Command	Parameter	Description
::Timeout	timeout in seconds	Changes the time *Motion* Perfect waits for a prompt to be returned. The default value is 10 seconds.
::Wait	wait time in seconds	Wait and do nothing for the given time

e.g.:

```
::Timeout 55
```

sets the timeout to 55 seconds

TESTS

Special support has been added in order to enable the use of scripts for testing purposes. The response from a command can be tested by Motion Perfect and the results written to a log file. A test is written on the line after the one whose response is to be tested and consists of a single ^ character followed by a list of alternative responses separated by single | characters. The comparison is done as a string comparison after all leading and training spaces have been removed.

e.g.:

```
^12.0000|13.0000
```

gives a **PASS** if the returned string is “12.0000” or “13.0000”, otherwise a **FAIL**.

The **PASS** or **FAIL** state of each test is logged in the log file and a summary of passes and failures is given at the end.

EDITING SCRIPTS

To edit or write a new script, select “Script / Edit” from the terminal window menu.

RUNNING SCRIPTS

To run a script normally, select “Script / Run” from the terminal window menu. This does not produce a log of what has happened.

To run a script with full logging, select “Script/Run logged” from the terminal window menu. The log will contain a full log of what has happened including test results.

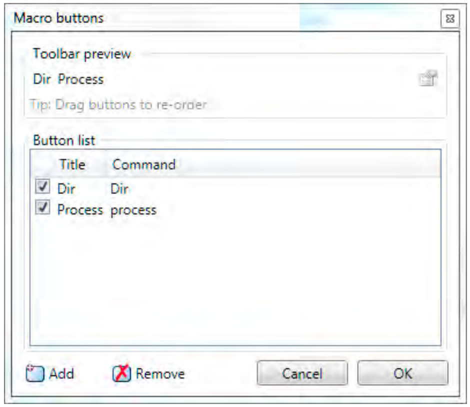
To run a script in test mode, select “Script/Run Test” from the terminal window menu. This will produce a log containing only test failures and a **PASS/FAIL** summary.

CONTEXT MENU

Entries allow the user to clear the terminal display, and copy and paste text in the terminal window.

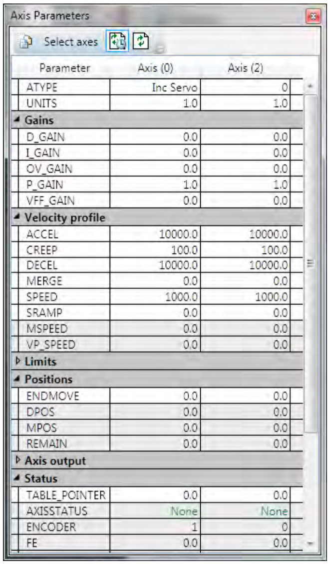
MACRO BUTTONS

There are a row of user configurable macro buttons above the status bar at the bottom of the terminal window. The user can configure these to send often used strings (commands) to the controller. To configure these buttons click on the icon at the right of the macro button bar. This will cause the "Terminal Macro Buttons" dialog to be displayed.



The "Add" button will add an entry in the button list and the "Remove" button will remove the selected entry. The title of is the text which is displayed in the button in the terminal window. The command is the string of characters sent to the controller. A carriage return character will be appended to the string when it is sent.

Axis Parameters

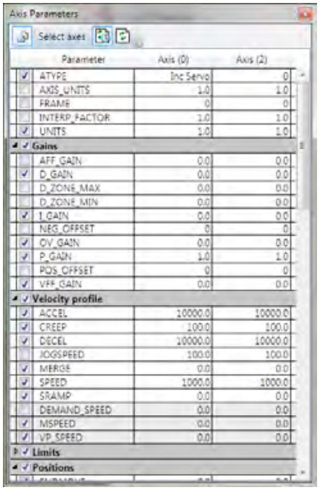


The Axis Parameters window enables the user to monitor and change the motion parameters for any axis on the controller. The display is made up of collapsible groups of parameters. This is done to make locating a parameter in the display easer and also allows the hiding of whole groups of parameters so that only parameters of interest are shown. It is also possible to individually show or hide individual parameters.

Parameters which can be edited have the normal edit box background and those which are read-only have a greyed-out background.

VIEWS

There are two main views; filtered view which shows selected parameters (see above) and all parameter view which allows the selection of individual parameters for the filtered view. Normally the filtered view is used. The view is selected by using the “all parameters” toggle button on the left of the window’s toolbar.



The “all parameters” view has a check box next to each parameter and group. If the box is checked then the corresponding parameter or group is displayed in the filtered view, otherwise it is hidden.

EDITING A PARAMETER

To enter a new value foe a parameter:

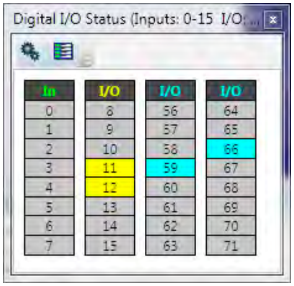
1. select its cell in the grid
2. type a new value

To edit a parameter:

1. double click on its cell in the grid

Digital I/O Viewer

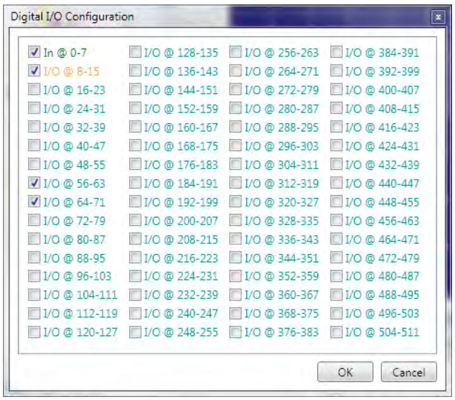
The digital I/O viewer is used to show the states of the digital inputs and outputs of the controller (both local and remote).



The display divides the I/O address space up into blocks of 8 lines. Usually all the lines in a block are the same type. The types available and their associated colours are shown in the table below:

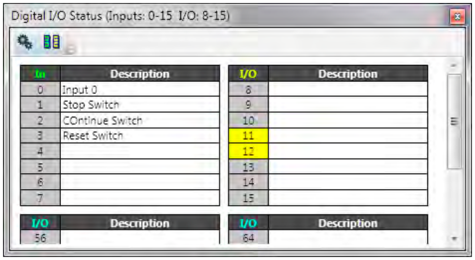
Type	**Colour**
Input	Green
Output	Orange
Input/Output	Yellow
Virtual Input/Output	Cyan

It is possible to change which banks are displayed by clicking on the “Configuration” button which then displays the configuration dialog.



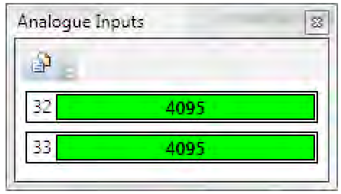
Using this dialog the user can select which banks of I/O lines to display.

Each i/0 line can be given a description. The description can be shown or hidden by clicking on the “Show/ Hide Descriptions” button or .



Analogue I/O Viewer

The analogue input viewer is used to show the values measured on the analogue inputs of the controller (both local and remote).



The tool normally displays inputs selected by the user. This defaults to showing all inputs until the user has selected which inputs to show. The value shown for each input is the raw value decoded by the hardware.

Clicking on the “Show All Inputs” button  in the toolbar toggles the display between the normal (filtered) display and the “All Inputs” display.

In “All Inputs” display mode there is a check box for each input to determine which inputs are displayed in normal mode. When in normal mode only the inputs which are checked will be displayed.

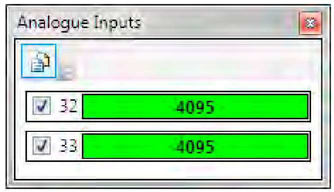
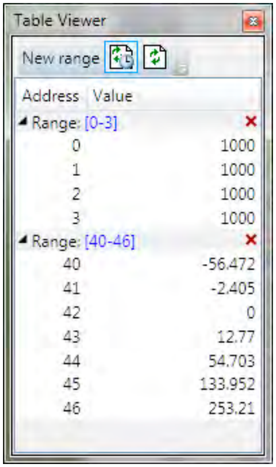


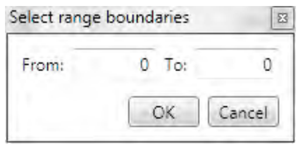
Table Viewer



The Table Viewer tool allows the user to view and edit ranges of **TABLE** memory.

VIEWING A RANGE

To add a range of **TABLE** values to the display click on the "New Range" button in the toolbar. This will bring up the "Select Range" dialog to allow the user to specify the range required.



After a range has been added to the viewer it can be edited by clicking on the corresponding range display in the tree (blue numbers), collapsed or expanded by clicking on the corresponding arrow in the tree, or deleted by on the corresponding red cross in the tree.

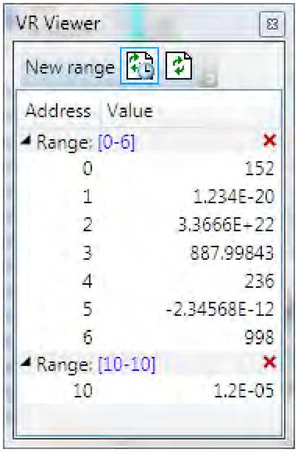
EDITING A VALUE

A value can be overwritten by clicking on it and entering a new value. A value can be edited by double clicking on it. In both of these cases the value is written to the controller when the "Enter" key is pressed. Pressing the "Esc" key will abort the edit. Changes can be made whilst programs are running.

REFRESHING THE VALUES DISPLAYED

The displayed valued can be updated automatically using periodic polling of the controller or manually when the user clicks on the refresh button . Automatic refresh is controlled by the "Periodic update" button. Clicking on the periodic update button changes its state from "Polling" to "Not Polling" . The update rate can be changes on the "General" tab of the main application options dialog.

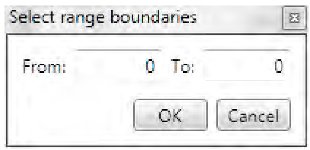
VR Viewer



The VR Viewer tool allows the user to view and edit ranges of VR values.

VIEWING A RANGE

To add a range of VRs to the display click on the “New Range” button in the toolbar. This will bring up the “Select Range” dialog to allow the user to specify the range required.



After a range has been added to the viewer it can be edited by clicking on the corresponding range display in the tree (blue numbers), collapsed or expanded by clicking on the corresponding arrow in the tree, or deleted by on the corresponding red cross in the tree.

EDITING A VALUE

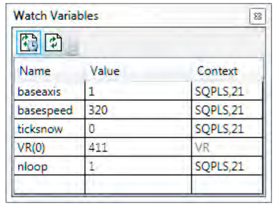
A value can be overwritten by clicking on it and entering a new value. A value can be edited by double clicking on it. In both of these cases the value is written to the controller when the “Enter” key is pressed. Pressing the “Esc” key will abort the edit. Changes can be made whilst programs are running.

REFRESHING THE VALUES DISPLAYED

The displayed valued can be updated automatically using periodic polling of the controller or manually when the user clicks on the refresh button . Automatic refresh is controlled by the “Periodic update” button. Clicking on the periodic update button changes its state from “Polling” to “Not Polling” . The update rate can be changes on the “General” tab of the main application options dialogue.

Watch Variables

The “Watch Variables” tool allows the user to look at the values of program internal variables and global variables while a program is running or stepping.



ADDING VARIABLES

The methods of adding variables to be watched is covered in the “Program Editor” section under “Watching Variables”.

VARIABLE INFORMATION

The entry for each variable contains the name of the variable, its present value (blank if not yet read) and its context. The context is either “VR” denoting a global VR variable or the program name and the process on which it is running.

UPDATING

The displayed values can be automatically updated periodically. Periodic updating enabled or disabled by clicking on the “Toggle Periodic Updating” button (when enabled, when disabled).

Clicking on the refresh button will cause the values to be updated regardless of the state of periodic updating.

CHANGING VALUES

Values can be edited by double clicking on the value in the grid and pressing the “Return” key. The act of pressing the “Return” key sends the value to the controller.

Options Dialogue

The options dialog has several pages of options for various tools in Motion Perfect. The page displayed is controlled by a tree control on the left of the dialog.

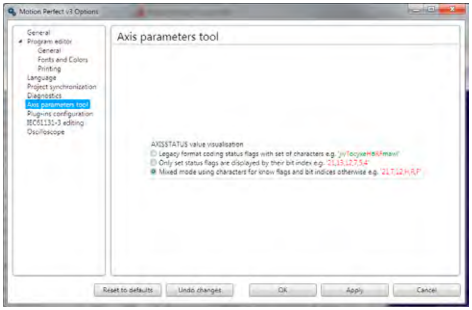
The following can be selected from the tree:

- General
- Program editor
- Language
- Project synchronization
- Diagnostics
- Axis Parameters Tool
- Plug-ins

Plugin options pages. These depend on which plugins are installed but may include:

- Oscilloscope
- IEC61131-3 Editing
- **HMI** Editing

Options - Axis Parameters Tool



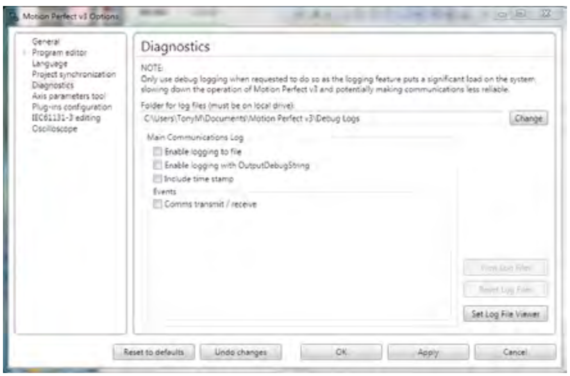
AXISSTATUS VISUALIZATION

This controls how the **AXISSTATUS** parameter is displayed in the parameter grid. The parameter can be displayed in one of three ways:

- **Legacy Format** – This is the same as *Motion* Perfect 2 and shows each known status bit as an alphabetic character, lower case green for clear, upper case red for set.
- **Numeric Set Flag Format** – This shows all known set status bits as their bit number. No clear bits are shown.
- **Mixed Set Flag Format** – This shows all known set bits as an alphabetic character and all unknown set bits as their bit number. No clear bits are shown.

Unknown flag bits can occur when new features are added to a controller.

Options - Diagnostics

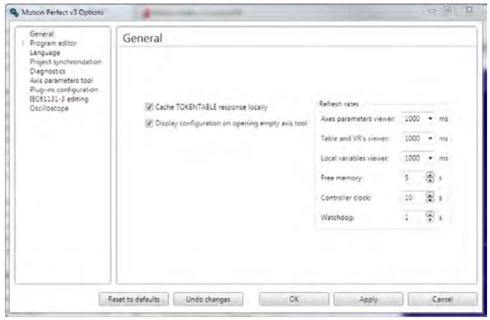


This page give options for diagnostics functions used to aid Trio Motion Technology in finding and rectifying faults in *Motion* Perfect.

Diagnostic functions should only be enabled on instruction from Trio Motion Technology as they reduce the application's performance and can lead to the application being less reliable.

Options – General

Options are available for the following:



TOKEN TABLE CACHING

When "Cache **TOKENTABLE** response locally" is checked, token table data for each controller type and system version used is stored on the PC. The token data is used by *Motion* Perfect to check that certain TrioBASIC commands are supported on the controller. If the token table data is not cached locally then it has to be read from the controller every time *Motion* Perfect connects in Tool Mode or Sync Mode.

Token table caching should be left enabled in order to speed up the connection process. The only time when it may need to be disabled is if special versions of controller system software (provided by Trio Motion Technology) are used on a controller.

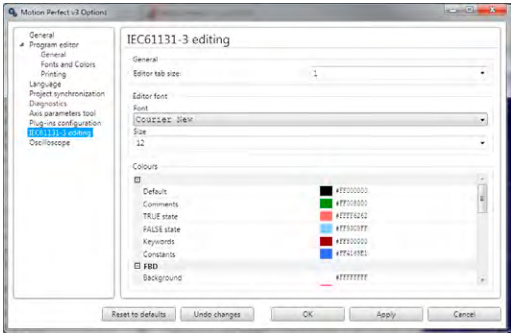
DISPLAY CONFIGURATION ON OPENING EMPTY AXIS TOOL

When checked, opening a tool which displays axis date will open an axis selection dialog if no axes have been previously selected.

REFRESH RATES

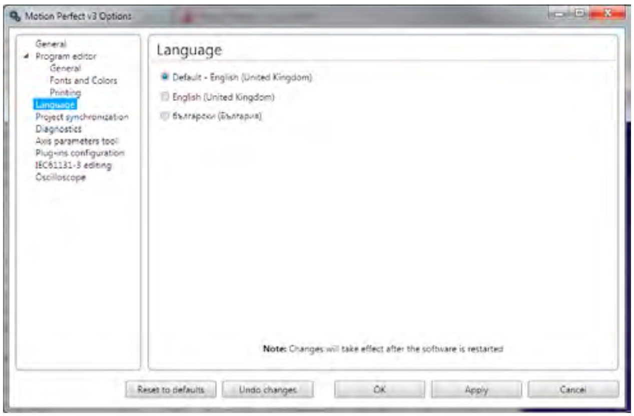
This allows the user to select the update rates used by various tools and monitoring processes. If a tool is set to update too frequently it may interfere with the operation of other tools due to the limited bandwidth of the communications link,

Options – IEC 61131 Editing



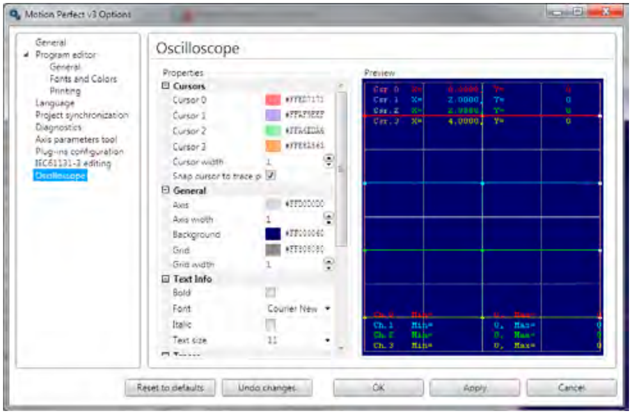
This allows the user to select options for the IEC61131-3 program editors. Some sections are common to all IEC61131-3 editors, others specific to the IEC61131-3 program type.

Options - Language



This allows the user to choose which of the available languages will be used by *Motion* Perfect to display text in the user interface. English (UK) will always be available, the availability of other languages may vary with application version.

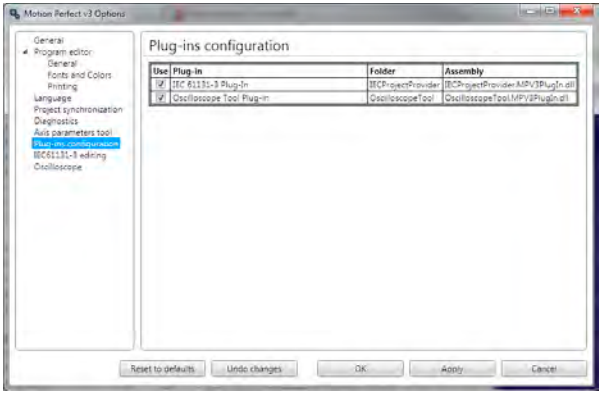
Options – Oscilloscope



This allows the user to change the display parameters used by the oscilloscope including:

- Background colour
- Grid colour and line thickness
- Trace colour, line thickness and data point size
- Cursor colour and line thickness
- Font used to display text
- Scale matching for X/Y plots
- Data set buffering for X/Y plots

Options – Plug-ins

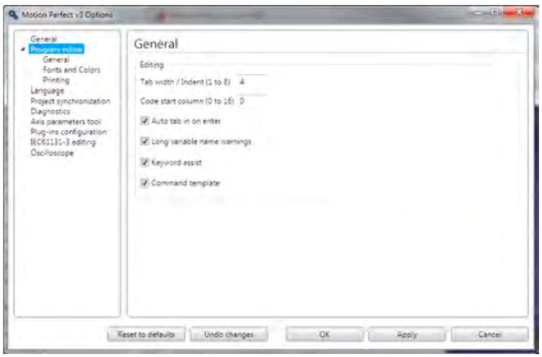


This page lists all the installed plug-ins and allows the user to enable or disable each one by means of a check box.

Options – Program Editor

The program editor options are controlled using three different pages:

PROGRAM EDITOR – GENERAL PAGE



This page specifies the options for automatic assistance whilst editing:

Tab width – the number of spaces to use for tabs

Code start column - the start column for line of TrioBASIC code when auto-formatting (label definition lines always start in column 0).

Auto-tab on enter – When checked enters spaces at the start of the new line to match the start column of the current line.

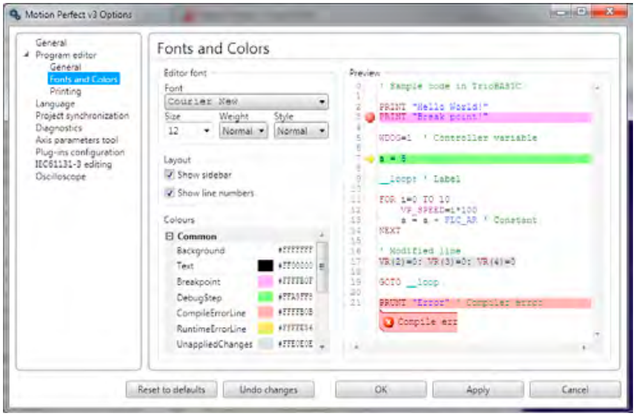
Long variable names warning – if checked the user is warned if a variable name is longer than the unique name size supported by the controller.

Variable names can be longer than the unique name size but the controller only checks the first "unique name size" characters for uniqueness.

Keyword assist – If checked the user is presented with a list of possible keywords as a keyword (or variable name) is being typed in.

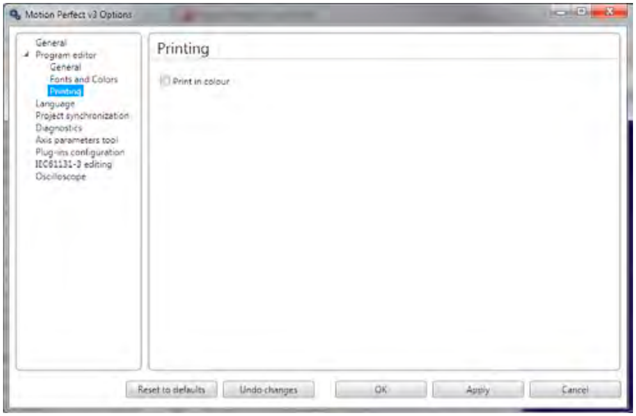
Command template – If checked, when the user types a command which has parameters in brackets, a template is displayed to remind the user of the parameters.

PROGRAM EDITOR – FONTS PAGE



This page allows the user to specify which font is to be used in the editor (including its weight and size). It also specifies the colours used for editing and debugging including syntax highlighting of TrioBASIC programs.

PROGRAM EDITOR – PRINTING PAGE

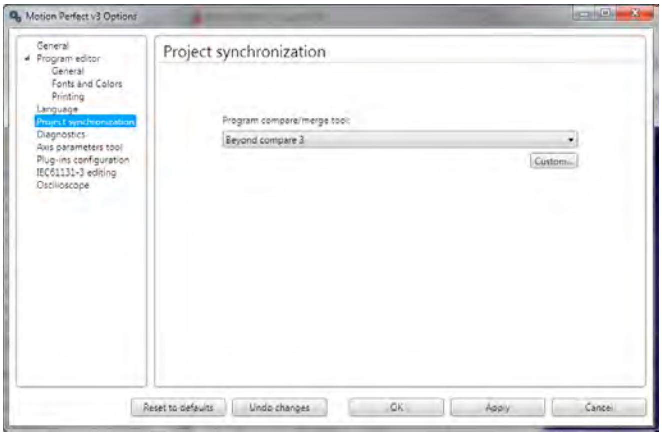


This page controls how program listings are printed.

PRINT IN COLOUR

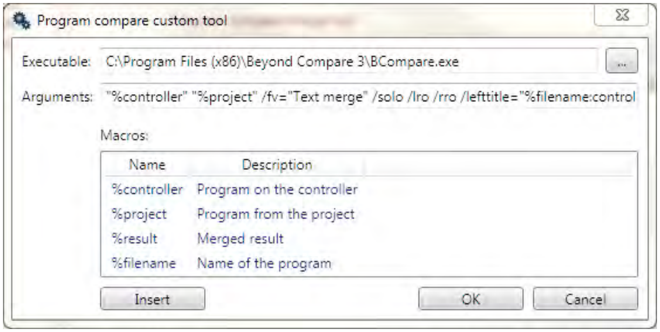
If this is checked then the printout is coloured using the same syntax highlighting colouring scheme as the editor screen display. Otherwise the printout is done in monochrome.

Options - Project Synchronization



This allows the user to select a program to use to compare the difference between the copy a program on the controller and the one in the project. I allows the user to configure any program which can compare text files. A list of common text file comparison programs is given in the drop down list.

Clicking on the “Custom” button will display the “Program Compare custom tool” dialog which allows the user to specify any suitable program already installed on the PC and which command line arguments are to be used.



If you do not have a suitable text file compare program installed on your computer, WinMerge can be downloaded free of charge from winmerge.org

Diagnostics

Motion Perfect has some built-in diagnostics which are designed to provide useful information in diagnosing some communications problems and possibly problems with *Motion* Perfect functionality. Diagnostic functions should not be used unless requested to do so by Trio Motion Technology, as enabling diagnostics increases the load on the application and can, in some cases, lead to unreliability.

See “Options – Diagnostics”

Jog Axes

The Jog Axes tool allows the user to move the axes on the *Motion Coordinator*.

This tool 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

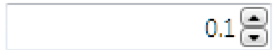
If there were no warnings set, then the message “Reverse jog set on axis?” is set in the warnings window, the **FAST _ JOG** input is invalidated for this axis, the **CREEP** is set to the value given in the jog speed control and finally the **JOG _ REV** output is turned off, thus enabling the reverse jog function.

JOG FORWARD

This button will initiate a forward jog. In order to do this, a check sequence identical to that used for **Jog**

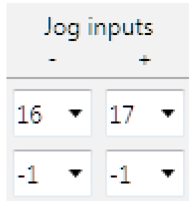
Reverse is performed.

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



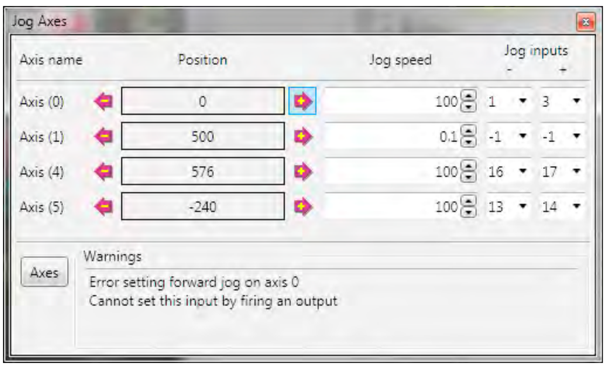
These are the inputs which will be associated with the forward / reverse jog functions.

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. Both real and virtual I/O lines can be used for jogging. The value -1 is shown when no input has been allocated for jogging.

The jog function depends on the state of the jog inputs as follows:

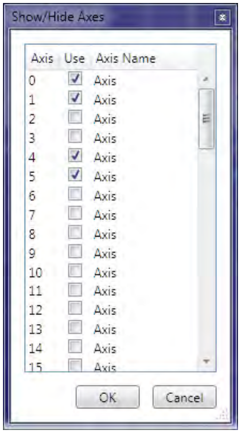
Jog -	Jog +	Function
OFF	OFF	Not defined
OFF	ON	Reverse Jog
ON	OFF	Forward Jog
ON	ON	No jog

WARNINGS



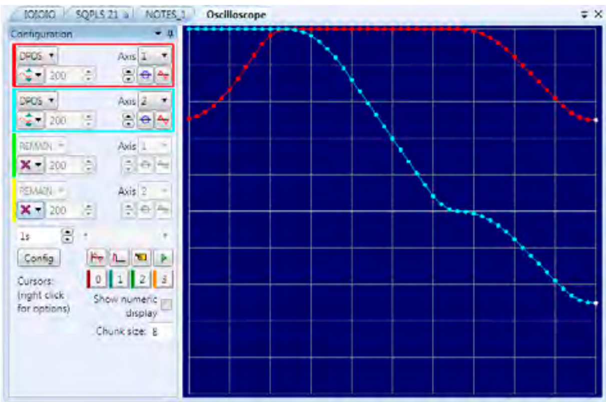
This shows the status of the last jog request. For example, the screen below shows axis 0 with IO channel 7 selected. This is an Input-only channel and therefore cannot be used in the jog screen.

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.

Oscilloscope



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

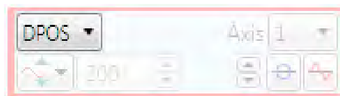
There are four groups of controls, one for each of the oscilloscope's four channels, a group of horizontal function controls and a group to control up to four cursors.

OSCILLOSCOPE CHANNEL CONTROLS

The controls for each of the four channels are grouped together and are surrounded by a coloured rectangle if the channel is ON, or a coloured bar to the left of the group if the channel is OFF. The colour is the same as the trace for that channel.

The group contains controls for channel operating mode, parameter selection and scaling.

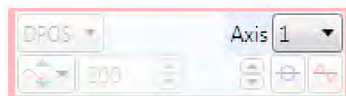
PARAMETER



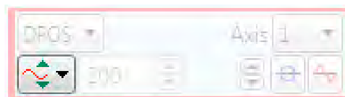
The parameters which the oscilloscope can record and display are selected using the pull-down list box in the upper left hand corner of each channel control block. Depending upon the parameter chosen, the next label switches between `axis' or `ch' (channel). This leads to the second pull-down list box which enables the user to select the required axis for a motion parameter, or channel for a digital input/output or analogue 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 analogue 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.



OPERATING MODE



The channel operating mode controls how the trace is displayed and scaled

	Trace off - no data gathered, trace not displayed
	Automatic Scaling - data gathered - trace automatically scaled to fit display
	Manual Scaling - data gathered - trace manually scaled
	Frozen - no data gathered - trace displayed as it was when frozen

VERTICAL SCALING

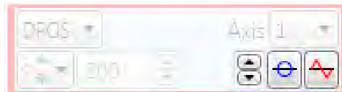
In automatic mode the oscilloscope calculates the most appropriate scale when it has finished recording, prior to displaying the trace. The value shown is the value calculated by the oscilloscope.

In manual mode the user selects the scale per grid division.

The vertical scale is changed by pressing the up/down scale buttons on the left side of the current scale text box.

CHANNEL TRACE VERTICAL OFFSET

There are three controls which control the vertical offset of the trace:



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.

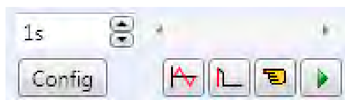
The Zero Offset button clears the vertical offset.

The auto-zero button, when active (in the down position), applies automatic vertical offset to the channel. The vertical offset and Zero Offset buttons are disabled (greyed out). This is equivalent to AC coupling on a conventional oscilloscope.

When not active the vertical offset manually set using the Vertical Offset buttons is applied. The vertical offset and Zero Offset buttons are enabled.

OSCILLOSCOPE HORIZONTAL CONTROLS

The oscilloscope horizontal controls appear towards the bottom of the oscilloscope control panel. From here you can control such aspect as the timebase, triggering modes and memory used for the captured data.



TIMEBASE

The required time base is selected using the up/down scale buttons on the left side of the current time base scale text box. 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.

HORIZONTAL DISPLAY MODE

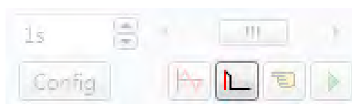
Button up  = x/t (timebase) mode.

This is the normal operation mode for an oscilloscope where each set of gathered data is plotted against time.

Button down  = x/y mode.

Channels are grouped in pairs and the values form one channel are plotted against the values of the other one in the pair.

ONE SHOT / REPEAT TRIGGER MODE



Button up  = 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 down  = Continuous (Auto-repeat) 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 up = Manual Trigger Mode:

In manual mode, the controller is triggered, and starts to record data immediately the oscilloscope trigger button is pressed.

Button down = 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.

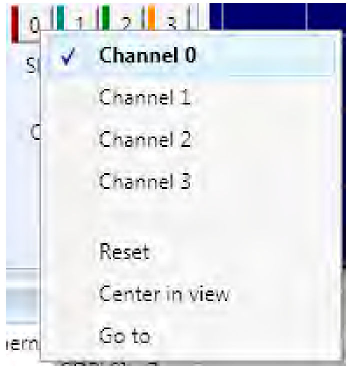
After the trigger button has been pressed, it changes to (stop) whilst the oscilloscope is running. If the oscilloscope is in the one-shot mode, then after the data has been recorded and plotted on the display, the trigger button returns to indicating that the operation has been completed. The oscilloscope can be halted at any time when it is running by pressing the button.

CONFIG. BUTTON

Clicking in the **Config.** button causes *Motion* Perfect to display the Capture Configuration Dialog.

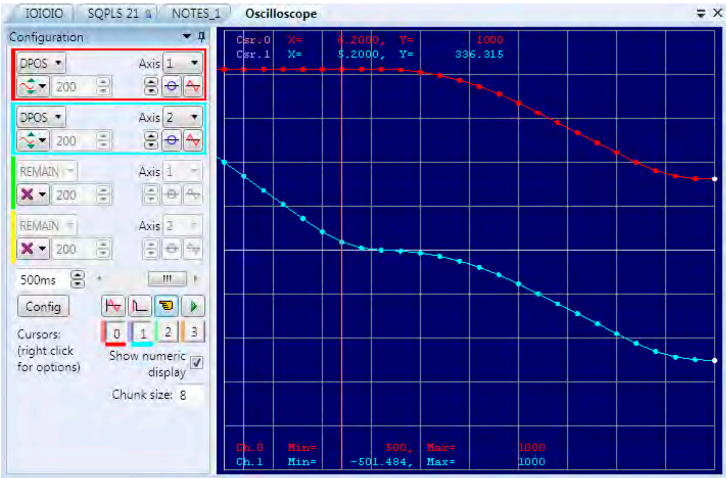
OSCILLOSCOPE CURSORS

The cursor bars are enabled/disabled by clicking on one of the cursor buttons which shows/hides the corresponding cursor. A cursor can 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. Cursors are automatically allocated to the first channel currently enabled. To allocate a cursor to a different channel, right click on its button and choose the desired channel from the pop-up menu. When a cursor is active a coloured bar representing the channel to which the cursor has been allocated is displayed under the cursor's button.



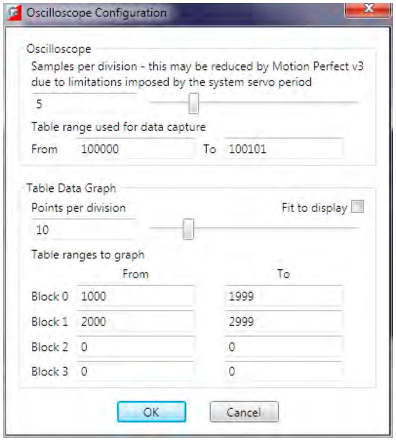
The cursor (right click) menu allows the user to assign the cursor to a channel and also contains **Reset** which resets the cursor position to a position close to the start of the display and **Go To** which scrolls the display so that the cursor is visible (only if zoomed in).

If the **Show numeric display** box is checked then the numeric display is enabled, this shows maximum and minimum values for all enabled traces at the bottom of the oscilloscope display and the positions of the active cursors at the top.



CAPTURE CONFIGURATION

When the **Config** button is pressed the oscilloscope capture configuration dialog is displayed, as shown below. Click the mouse button over the various controls to reveal further information.



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 analogue inputs are being recorded, then the fastest oscilloscope resolution (sample rate) is the number of analogue channels in milliseconds (i.e. 2 analogue 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 values 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.

Before the oscilloscope is triggered a sample quantization check is done to make sure that it is possible to gather the data at the sample interval requested. This may cause the number of samples per division to be adjusted so that the controller is able to gather the data at a sample period which is a whole number of servo cycles.

OPTIONS

The oscilloscope options are used to control the visual look of the oscilloscope display. Most colours and line thicknesses can be set, allowing the user to set up the oscilloscope to their own preference.

The **X/Y mode only** settings control the matching of the two channels used to capture X/Y data and the number of data sets buffered (and displayed) when in X/Y mode.

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.

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

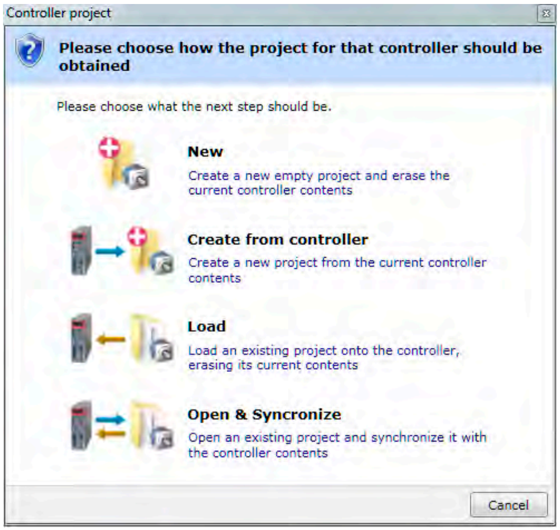
The controller records the parameter values at the required sample rate in the table, and then passes the information to the oscilloscope. Hence the trace displayed is accurate with respect to the selected time base. However, there is a delay between when the data is recorded by the controller and when it is displayed on the oscilloscope due to the time taken to upload the data via the communications link.

Intelligent Drives

Intelligent drive are drives which contain built-in control loops and are controlled via a digital interface, often over a data bus. *Motion* Perfect supports the configuration but means of add-ins. The following add-ins are currently available:

Add-in	Drives Supported

Controller Project Dialogue



The “Controller Project Dialog” is displayed when the user first attempts a Sync Mode connection to a controller. The options available are explained on the dialog.

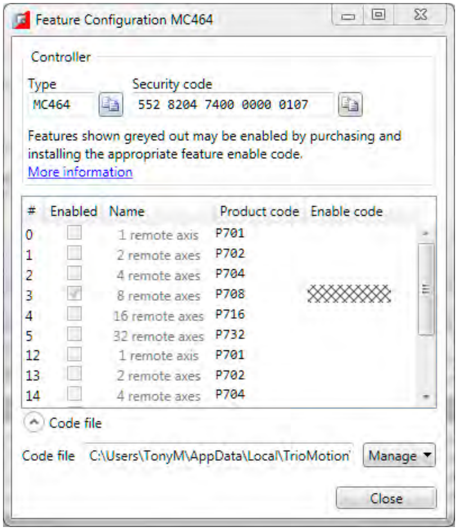
Controller Tools

Motion Perfect 3 has several tools which are used to configure the controller and interact with it. Most of these tools are available from the “Controller” section of the Main Menu.

Tool	Description
Connection Settings	Settings for the communications interface on the PC used by Motion Perfect to communicate with the controller
Reset Controller	Performs a soft reset on the controller
Interfaces	Settings for the communications interfaces on the controller
Enable Features	Enable or disable software configurable features on the controller
Memory Card	Manipulate files stored on the memory card in the controller
Load Firmware	Load system firmware onto the controller
Directory	Show a full directory listing of the programs on the controller
Processes	Show details of the processes currently running on the controller
Lock / Unlock Controller	Lock or unlock the controller
Date And Time	View or change the real-time clock on the controller.

Feature Configuration

Some *Motion Coordinator*s have features which can be enabled by the user. The features are enabled using the “Feature Configuration” tool.



FEATURE CODES

The features are made available by purchasing feature enable codes from Trio Motion Technology Ltd, each feature having a unique code, the codes also being different for every controller. Feature codes are stored on the computer in a special file on the computer which holds all feature codes entered. This file (default “FeatureCodes.tfc”) is normally located in the “TrioMotion \ MotionPerfectV3” sub directory of the current user’s local application data directory. The file used can be changed to another in a different location by clicking on “Manage” button and selecting “ Change from the drop-down list. It is also possible to import values from another Feature Code file by selecting “Import” from the same drop-down list.

To manually enter a new code select the appropriate “Enable” Code” cell in the feature grid and enter the code, being careful to get the case of the characters correct. If the code is entered correctly then the “Enabled” check box for the feature should become enabled and allow the user to enable and disable the feature.

When purchasing feature codes you will need to supply the Security code for your controller to ensure that you get the correct codes.

Feature codes are based on three factors: the feature number, an internal device code held in the controller, and the serial number of the controller. Each code is unique, so it is vital that the correct security code and feature number (or product code) are used when ordering a feature code.

Load System Firmware

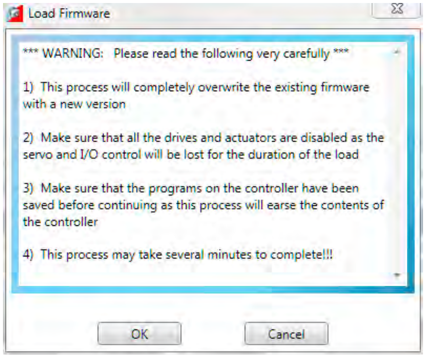
*Motion Coordinator*s feature a flash EPROM for storage of both user programs and the system firmware. Using

Motion Perfect it is possible to upgrade the system firmware to a newer version using a system file supplied by Trio.

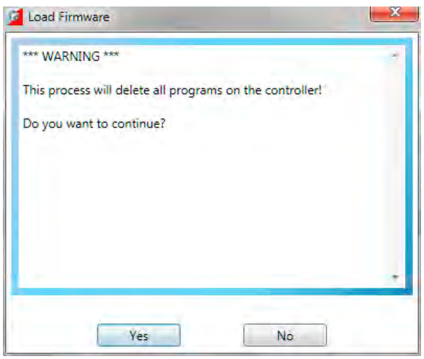
We do not advise that you load a new version of the system firmware unless you are specifically advised to do so by your distributor or by Trio.

The process of loading new system firmware will erase all programs stored on the controller. So make sure that they are backed up (in a project on the PC) before starting.

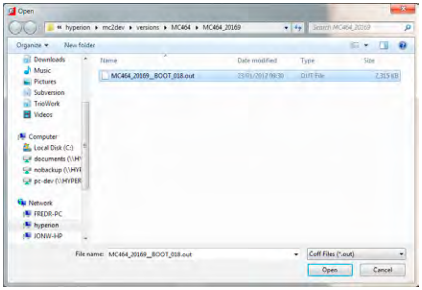
When you select the ‘Load Firmware’ option from the controller menu, you will first be presented with a warning dialog to ensure you have saved your project and are sure you wish to continue.



if you click on **OK** you will then be warned that the operation will delete all programs on the controller. This must be done because the programs are stored on the controller in a tokenized form and loading new system code may change the token list, consequently changing the commands in the programs.



When you click on **Yes** you will be presented with the standard Windows file selector to choose the file you wish to load.

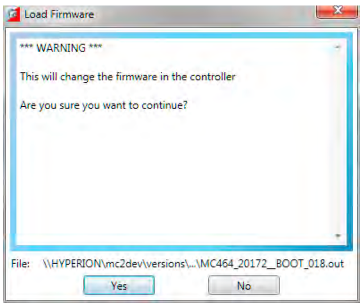


Each *Motion Coordinator* controller has its own system file, identified by the first characters of the file name.

System Code File Name	File Type	Controller Type
MC403*.OUT	**COFF**	MC403
MC405*.OUT	**COFF**	MC405
MC464*.OUT	**COFF**	MC464

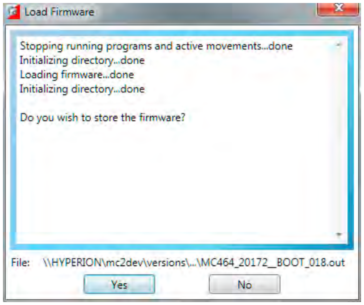
You must ensure that you load only software designed for your specific controller, other versions will not work and will probably make the controller unusable.

When you have chosen the appropriate file you will be prompted once again to check that you wish to continue. Click on **Yes** to start the download process.

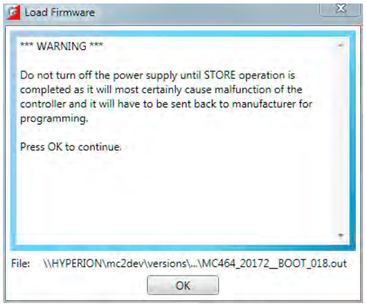


Downloading may take several minutes, depending on the speed of your PC, the controller and the communications link being used. During the download, you should see the names of each section displayed in the Output Window as they are loaded.

When the download is complete, a checksum check is performed to ensure that the download process was successful. If it passes the check you will be presented with a confirmation screen and asked if you wish to store the firmware into EPROM.



When you click on **Yes** a further warning dialog is displayed.



It will take a short time to fix the project into the EPROM and reconnect to the controller. You can then click on **Yes** and continue using *Motion* Perfect in the normal way.

It is advisable to check the controller configuration to confirm the new firmware version.

Lock / Unlock Controller

Locking the controller will prevent any unauthorised user from viewing or modifying the programs in memory, and also prevent *Motion* Perfect from connecting in Sync mode.

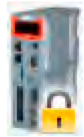
LOCKING

To Lock the currently connected controller, select “Controller / Lock Controller” from the main menu.

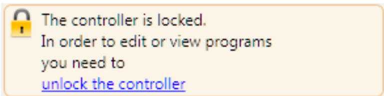
In the “Controller Lock” dialog, enter a numeric code (up to 7 digits) as a lock code. This value will be encoded by the system and used to lock the directory structure. The lock code is held in encrypted form in the flash memory of the controller.

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.

When the controller is locked the controller icon in the "Controller Tree" will have a lock symbol overlaid on it,



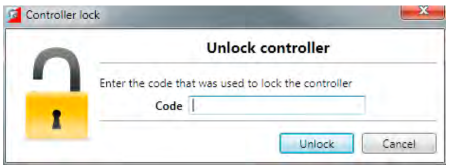
a message will be shown at the bottom of the controller tree,



and the controller name in the "Status Bar" will have a lock symbol next to it.

UNLOCKING

To Unlock the currently connected controller, select "Controller / Unlock Controller" from the main menu (only available when the controller is locked).

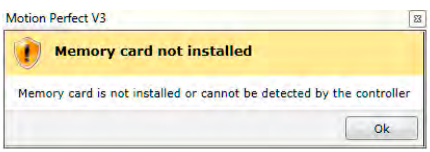


Enter the lock code with which the controller was previously locked. After the lock code has been accepted full access to the contents of the controller will be restored.

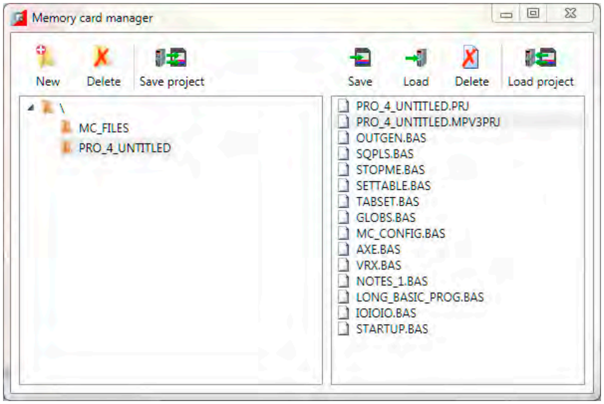
Memory Card Manager

The "Memory Card Manager" allows the user to manage the contents of the memory card in the controller. It is started by selecting "Controller / Memory Card" from the Main Menu.

If there is no memory card present a warning dialog is displayed.

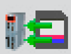


If a memory card is present the Memory Card Manager dialog is displayed.

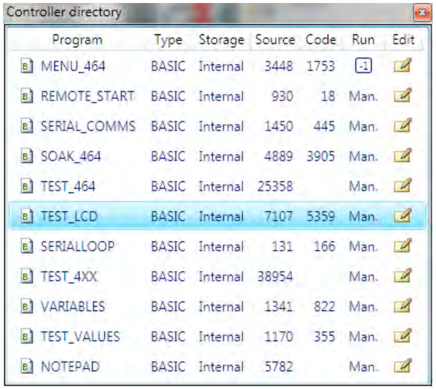


The panel on the left of the dialog shows the directory structure on the memory card and the panel on the right shows the files (not directories) in the currently selected directory.

The following operations are available:

Icon	Operation	Description
	New folder	Creates a new sub-folder in the selected folder
	Delete folder	Deletes the selected folder
	Save Project	Saves the project from the controller into the selected folder
	Save to Card	Saves one or more programs from the controller into the selected folder on the memory card
	Load from Card	Loads the selected program file onto the controller from the memory card
	Delete	Deletes the selected program
	Load Project	Loads the selected project onto the controller. This option is only available when a project file (extension .mpv3prj) is selected

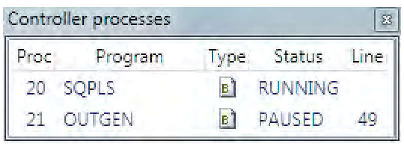
Directory Viewer



The Directory Viewer shows a more detailed directory view to that available in the “Controller Tree”. The information in the grid is as follows:

Column	Description
Program	Program name
Type	Program type
Storage	Storage location (Normally internal)
Source	Source code size in bytes
Code	Object code size in bytes
Run	Run method: Manual or Auto-run process number
Edit	Edit the program by clicking on the icon. If the icon is greyed-out then the program is not editable (running programs are not editable and some programs may be locked against editing for other reasons).

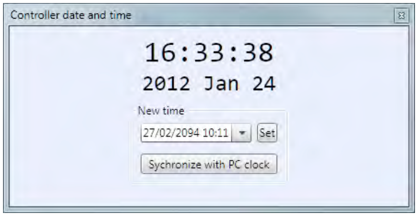
Process Viewer



The Process Viewer shows information about all currently running user processes on the controller. The information in the grid is as follows:

Column	Description
Proc.	Process number
Program	Program name
Type	Program type (See "Program Types")
Status	Run status (usually **RUNNING** or **PAUSED**)
Line	Current execution line in the program (if **PAUSED**)

Date And Time Tool



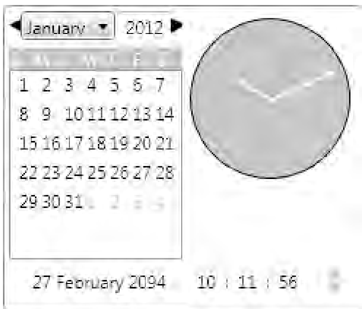
The Date and Time tool is used to monitor and set the real-time clock on the controller.

SETTING THE DATE AND TIME

The date and time can be set in two ways:

MANUAL SETTING

To set the date and time manually, click on the combo box to display a date and time selector dialog.



Select the date and time in the dialog then click outside it. The date and time selector dialog will close. Then click on the Set button in the Date and Time tool to update the controller.

AUTOMATIC SETTING FROM THE LOCAL PC CLOCK

To set the date and time on the controller to same time as the local PC clock, click on the "Synchronize with PC Clock" button.

STARTUP Program

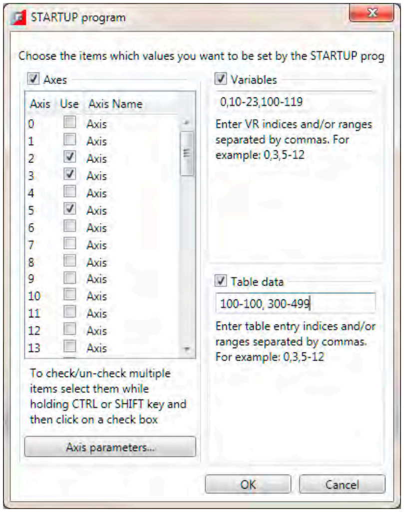
The **STARTUP** program is an automatically generated program designed to be run at system start to initialize the system. The **STARTUP** program is a standard TrioBASIC program which needs to be run as a user specified auto-run program (unlike the **MC _ CONFIG** program which always run at power-up).

The STARTUP program should not be edited manually as doing so may result in the manual additions being lost when the program is regenerated or wrong values being generated if code used by the automatic generation process is changed.

The file is divided up into sections each section being generated by a different tool. Some add-ins will generate a section in the **STARTUP** file for the configuration of external devices (such as intelligent drives).

Modify STARTUP Program

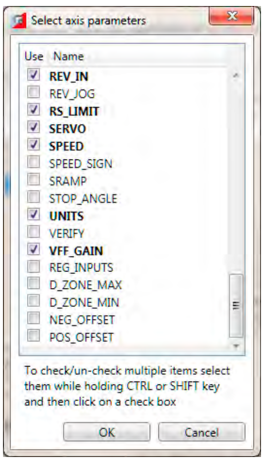
The **STARTUP** program is a user run TrioBASIC program used to initialize the system on power-up. It is commonly used to set up Axis Parameters, **TABLE** areas, VR Variables and Drive Parameters (when intelligent drive support is available).



The “Modify **STARTUP** Program” tool allows the user to save Axis Parameters, VR Variables and **TABLE** data in the **STARTUP** file so that it can be used to initialize the system. The storing of each type of data in enabled using a check box (check to enable).

AXES

The axes whose parameters need to be stored should be selected in the axis table. After doing this click on the “Axis Parameters” button to display the “Axis Parameters Selection Dialog” which allows the user to select which parameters should be stored. The same parameters are stored for all selected axes.



VARIABLES

VR variables can be stored by specifying variable numbers and ranges of variable numbers.

e.g. 1,4,6-9,12-23 will store VR(1), VR(4), VR(6) to VR(9) and VR(12) to VR(23)

TABLE DATA

TABLE values can be stored by specifying table indices and ranges of table indices.

e.g. 1,4,6-9,12-23 will store **TABLE**(1), **TABLE**(4), **TABLE** (6) to **TABLE** (9) and **TABLE** (12) to **TABLE** (23)

MC_CONFIG Program

The **MC _ CONFIG** program is a special program which can contain a small subset of TrioBASIC commands. It is automatically run at power-up and is used to set some basic configuration parameters on the controller.

MC _ **CONFIG**, if present, is always run at power-up and does not need to be specified as an auto-run program. It is always run before user specified auto-run programs.

If a parameter is not set in **MC _ CONFIG** then the value in the controller's flash EPROM memory is used.

The following system parameters can be written in the **MC _ CONFIG** program. No other **BASIC** commands or parameters are allowed. If an illegal parameter is put in the **MC _ CONFIG** program then it will cause a compiler error.

Parameter Name	Parameter Stored in
AUTO _ ETHERCAT	RAM
AXIS _ OFFSET	Flash EPROM
CANIO _ ADDRESS	Flash EPROM
CANIO _ MODE	Flash EPROM
IP _ ADDRESS	Flash EPROM
IP _ GATEWAY	Flash EPROM
IP _ NETMASK	Flash EPROM
MODULE _ IO _ MODE	Flash EPROM
REMOTE _ PROC	Flash EPROM
SCHEDULE _ TYPE	Flash EPROM
SERVO _ PERIOD	Flash EPROM
IP _ MEMORY _ CONFIG	RAM
IP _ PROTOCOL _ CONFIG	RAM

Parameter modifiers; SLOT and AXIS are allowed where appropriate.

PARAMETER DESCRIPTION

AUTO_ETHERCAT
Select the startup mode of EtherCAT. (Default: ON)

```
AUTO_ETHERCAT = OFF ‘ do not start EtherCAT network on power up
```

AXIS_OFFSET
Set the start address of an MC464 axis module. (Default: 0)

```
AXIS_OFFSET SLOT(1)=16 ‘ set start axis of module in slot 1
```

CANIO_ADDRESS
Set the operating mode of the built-in CAN port. (Default: 32)

```
CANIO_ADDRESS=40 ‘ set the CANIO_ADDRESS to use CANopen IO
```

CANIO_MODE
Determines the mode used with **CANIO** modules P317 (output), P318 (input) and P327 (relay).

Set to 0 to use the “up to 512” IO point mode. Set to 1 to use the mode compatible with MC2xx *Motion Coordinator*s. (Default: 0)

```
CANIO_MODE=1 ‘ set the CANIO to compatibility mode
```

IP_ADDRESS
Set the network IP address of the main Ethernet port. (Default: 192.168.0.250)

```
IP_ADDRESS = 192.168.0.110
```

IP_GATEWAY
Set the default gateway of the main Ethernet port. (Default: 192.168.0.255)

```
IP_GATEWAY = 192.168.0.103
```

IP_NETMASK
Set the subnet mask of the main Ethernet port. (Default: 255.255.255.0)

```
IP_NETMASK = 255.255.240.0
```

MODULE_IO_MODE
Define the operation and position of the axis module digital IO. (Default: 1)

```
MODULE_IO_MODE = 2 ‘ set so that module IO is after CAN IO
```

REMOTE_PROC
For use in systems with the TrioPC ActiveX. When the programmer needs to allocate the ActiveX synchronous connection to use a certain process number, set this value. (Default: -1)

```
REMOTE_PROC = 10 ‘ set the ActiveX to use process 10
```

SCHEDULE_TYPE
Alters the MC464 multi-tasking scheduler. See MC4xx Technical Reference Manual. (Default: 0)

```
SCHEDULE_TYPE = 0 ‘ WA() commands release their process for
                  ‘ other programs to use.
```

```
SCHEDULE_TYPE = 1 ‘ WA() commands use up all their process time
```

SERVO_PERIOD

Set the scan period of the servo loops and motion in microseconds. (Default: 1000)

```
SERVO_PERIOD = 500 ‘ set to half millisecond servo period.
```

IP_MEMORY_CONFIG

Set the Ethernet processor memory allocation. Buffer sizes can be increased to allow better processing of Ethernet Packets on a busy network. There is a trade-off between buffer size and the number of available protocols that can be connected. The default buffers are 2 for Tx and 2 for Rx. This allows all protocols to be used.

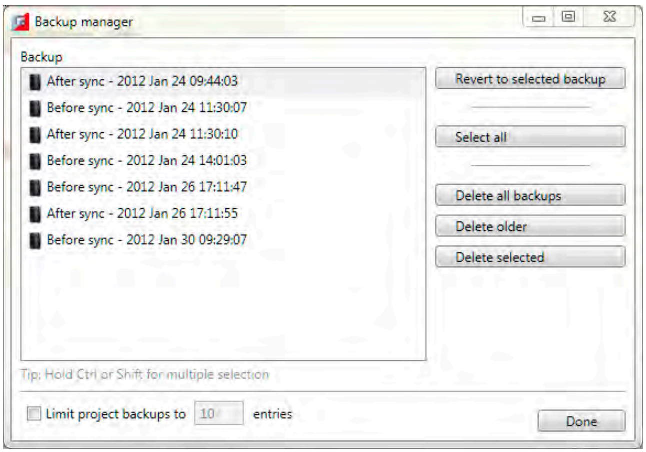
Increasing the buffers sizes must be done according to instructions from Trio *Motion* Technology, otherwise an unstable configuration may result.

IP_PROTOCOL_CONFIG

Set the available protocols ON or OFF. By default all protocols are available.

This should only be used under after taking advice from Trio Motion Technology.

Backup Manager



The “Backup Manager” is used to manage the backups automatically created before and after every synchronization operation.

As *Motion* Perfect is used the number of stored backups can become excessively large. The “Backup Manager” gives the user a way to limit these backups or to easily delete multiple backups if automatic limiting is not in use.

AUTOMATIC LIMITING

To automatically limit the number of backups stored check the “Limit Project Backups” check box and enter the number of entries you would like to keep. The backups kept are always the most recent ones. Although automatic limiting is good for saving disk space it is not good for keeping backup for any length of time.

MANUAL LIMITING

If the “Limit Project Backups” check box is not checked then no backups are deleted automatically. This means that the user should use the backup manager to remove unwanted backups in order to stop the number of stored backups growing excessively. Buttons allow the selection and deletion of individual and ranges of backups as well as the deletion of all backups.

It is possible to set the automatic limit to a high number to give an overall limit but to manage the backups manually.

REVERTING

To revert the project back to a given backup; Select the backup and click on the “Revert to Selected Backup” button.

6 IEC 61131-3 PROGRAMMING

IEC 61131-3 and *Motion* Perfect

This help file covers program using IEC 61131 languages using Trio Motion Technology’s *Motion* Perfect v3 application when used in conjunction with a compatible Trio 4 range of *Motion Coordinator*. The system supports several of the IEC 61131-3 defined languages providing both editing and debugging support.

Controller and Project Trees

IEC 61131 tasks are shown in the Controller and Project trees on the same level as a TrioBASIC program. This is because each represents an executable item which runs on a single controller process. All programs and spy lists in a task are shown as sub-items to the task in the tree.

The tree items have context menus to allow the user to perform associated operations.

CONTEXT MENUS

IEC TASK

Menu Item	Description
Add New IEC Program	Displays a dialog where the user can enter the new IEC program name, the IEC language and program run type
Add new spy list	Adds a new spy list to the IEC task
Open IEC variables	Opens the IEC variables editor tool
Open IEC types	Opens the IEC custom types editor tool
Compile IEC 61131-3 programs	Compiles all the IEC programs in the IEC task and creates an executable. The IEC Build results tool window is automatically shown
Run	Starts execution of the IEC task
Run on process	Displays a dialog where the IEC task can be started on a particular process

Menu Item	Description
Stop	Stops execution of the IEC task
Executable info	Displays information about executable – timestamps and version
Set AUTORUN	Displays a dialog where AUTORUN properties of the task can be specified
Delete	Deletes this IEC task
IEC 61131-3 settings	Displays IEC task settings window, where the user can modify different properties of the IEC task

IEC PROGRAM

Menu Item	Description
Edit	Opens the selected program for editing
Open local IEC variables	Opens an editor for local program variables
Open IEC variables	Opens the IEC variables editor tool, with the selected program variables grouped first
Open IEC types	Opens the IEC custom types editor tool
Rename	Opens a dialog, where a new name for the selected program can be specified. The program must not be open for editing in order to be renamed
Delete	Deletes the selected program from the IEC task

IEC SPY LIST

Menu Item	Description
Edit	Opens the selected spy list
Rename	Opens a dialog, where a new name for the selected spy list can be specified. The spy list must not be open for editing in order to be renamed
Delete	Deletes the selected spy list from the IEC task

DOUBLE CLICK ACTION

Double clicking on any IEC program or Spy List in the tree will open it for viewing or editing.

Languages

Motion Perfect v3 supports the following IEC 61131-3 defined languages:

- Ladder Diagram (**LD**)
- Structured Text (**ST**)

- Function Block Diagram (**SFD**)
- Sequential Function Chart (**SFC**)

Each of the languages has its own editor and can interact with the IEC 61131 environment shared between all programs running on the sane IEC 61131 task.

The IEC 61131 Environment

TASKS

Trio 4 range of *Motion Coordinator*s run programs in a pre-emptive multitasking environment with a limited number of processes. Normally IEC 61131 programs run on a single process (called a task) although it is possible to run more than one task in which case one process per task is used. Each task has its own IEC environment which holds “Task Variables” for that task.

VARIABLES

IEC variables are defined as “Local” which only apply to a single program or “Task” which apply to all the variables in a task.

“Task Variables” are not shared between different tasks. IEC 61131 programs which need to share “Task Variables” must all be run in the same task.

Run all IEC 61131 programs should be run in the same task unless there is a compelling reason to do otherwise.

During debugging variables can be monitored using task based “Spy Lists”, more than one of which can be defined for the each task.

COMPILATION

When an IEC 61131 program is compiled, all the programs in that task are compiled into a single executable entity which can be executed on the controller and controlled using the usual *Motion* Perfect RUN/**STOP**/**AUTORUN** etc. functionality.

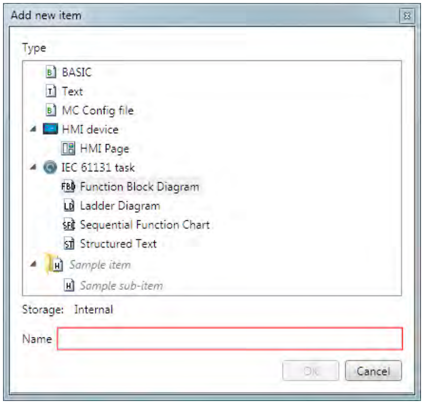
Adding a New IEC 61131 Program

ADDING VIA THE “ADD NEW PROGRAM” MENU

A new IEC 61131 program can be added to a *Motion* Perfect project in one of two ways:

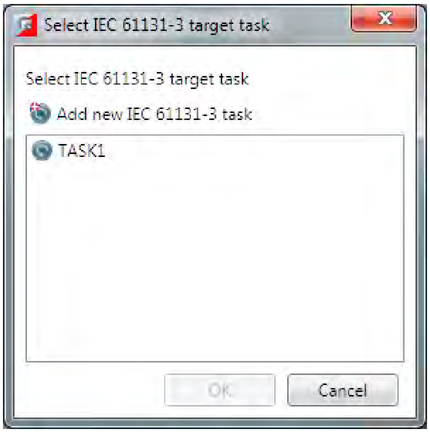
1. From the context menu associated with the “Programs” item in the Controller or Project tree, select “New...”
1. From Program main menu, select “New Program...”

The “Add New Program” dialog will be displayed.



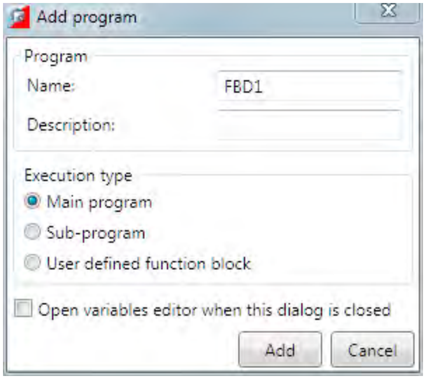
If IEC 61131 task is selected, this will add a new empty IEC task to the project.

If one of the IEC 61131 program types is selected the “Select Task” dialog is displayed.



This allows the user to create the program on an existing task (by selecting the task from the list) or a new one (by clicking the “Add New” button).

After selecting a task an closing the dialog the “Add Program” dialog will be displayed.

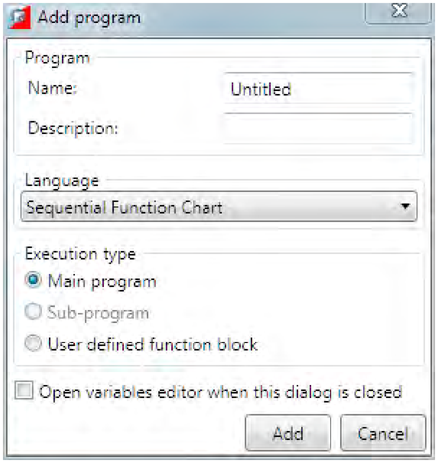


The fields and options in this dialog are as follows:

Field / Option		Description
Name		The name of the new IEC program
Description		Optional. Description of the new IEC program
Execution type	Main program	The program will be called on each cycle during IEC execution
	Sub-program	The program will be called by other programs in the IEC task. This type of execution is not allowed for SFC programs.
	User defined function block	The program will be custom “User defined function block”
Open variables editor when this dialog is closed		When checked, displays an editor for local variables for the new program. This editor is also available from the context menus of the program

ADDING TO AN EXISTING IEC 61131 TASK

To add a program to an existing IEC task right click on the task in the Controller or Project tree. This will display the “Add Program” dialog.



The fields and options in this dialog are as follows:

Field / Option		Description
Name		The name of the new IEC program
Description		Optional. Description of the new IEC program
Language		The IEC 61131 language used for the program
Execution type	Main program	The program will be called on each cycle during IEC execution
	Sub-program	The program will be called by other programs in the IEC task. This type of execution is not allowed for SFC programs.
	User defined function block	The program will be custom “User defined function block”
Open variables editor when this dialog is closed		When checked, displays an editor for local variables for the new program. This editor is also available from the context menus of the program

Editing Programs

To Edit an IEC program; double click on its entry in the Controller or Project Tree.

All IEC editors support standard edit operations, like CUT,**COPY** and **PASTE**. All of the editors support printing, which is available from the toolbar buttons.

When editing a larger program, it is sometimes useful to mark some pieces of code, so the user can easily navigate through the program. For this purpose, all IEC editors support Bookmarks.

All editors also support Find and Replace functionality. Find and replace window is accessible by pressing the “Ctrl+F” key combination on the keyboard.

All of the editors support drag and drop operations(from other IEC editors, from the variables tool and from spy lists). All of the editors, except SFC editor, support drag and drop of function blocks from the toolbox.

For information on editing a specific type of IEC program see one of the following:

- Editing **ST** Programs
- Editing **LD** Programs
- Editing **FBD** Programs
- Editing **SFC** Programs

Editing LD Programs

IEC 61131-3 LD language is a graphical programming language. **Ladder logic** is a programming language that represents a program by a graphical diagram based on the circuit diagrams of relay logic hardware.

The language itself can be seen as a set of connections between logical checkers (contacts) and actuators (coils). If a path can be traced between the left side of the rung and the output, through asserted (true or “closed”) contacts, the rung is true and the output coil storage bit is asserted (1) or true. If no path can be traced, then the output is false (0) and the “coil” by analogy to electro-mechanical relays is considered “de-energized”.

Ladder logic has contacts that make or break circuits to control coils.

Each rung of ladder language typically has one coil at the far right.

—( )— A regular coil, energized whenever its rung is closed.

—[ ]— A regular contact, closed whenever its corresponding coil or an input which controls it

The “coil” (output of a rung) may represent a physical output which operates some device connected to the controller, or may represent an internal storage bit for use elsewhere in the program.

Double-clicking on a contact or a coil displays a dialog for selecting the input/output for the element.

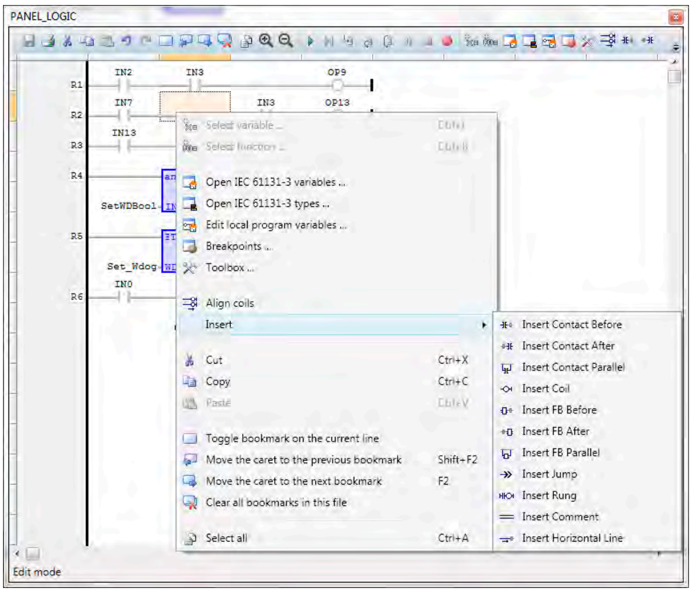
Double-clicking on a function/function block displays a dialog for selecting the function/functional block for the element.

The editor contents can be zoomed in and out via the toolbar buttons, or using the shortcut combinations “Ctrl +” for zoom in and “Ctrl -” for zoom out.

The LD editor context menu has the following functionality:

Menu Item	Action
Select variable	Displays a dialog for inserting/selecting a variable
Select function	Displays a dialog for inserting/selecting function block
Open IEC 61131-3 variables	Open the IEC variables tool
Open IEC 61131-3 types	Open the IEC types tool

Menu Item	Action
Edit local program variables	Open local variables editor
Breakpoints	Open breakpoints manager window
Toolbox	Open toolbox control, from where using drag and drop functions and function blocks can be added to the program
Align coils	Align the coils in program
Insert contact before	Inserts a contact before the selection
Insert contact after	Inserts a contact after the selection
Insert contact parallel	Inserts a contact parallel to the selection
Insert coil	Inserts new coil
Insert FB before	Inserts a new function block before selection
Insert FB after	Inserts a new function block after selection
Insert FB parallel	Inserts a new function block parallel to selection
Insert Jump	Inserts a new jump
Insert Rung	Inserts a new rung
Insert comment	Inserts a new comment
Insert horizontal line	Inserts a new horizontal line



Editing ST Programs

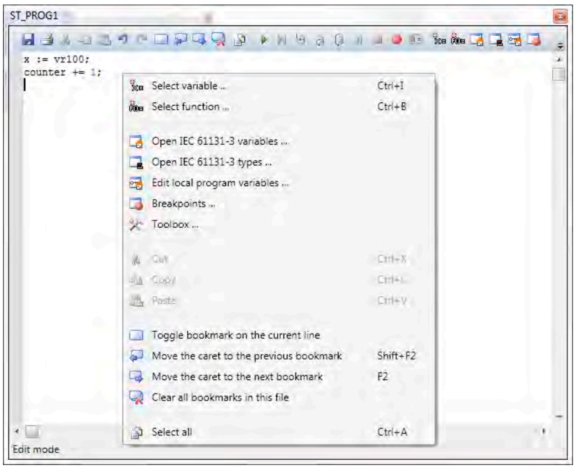
IEC 61131-3 ST language is a text-based programming language. It supports most of the traditional procedural programming language paradigms. t is a high level language that is block structured and syntactically resembles Pascal. All of the languages share IEC 61131 Common Elements. The variables and function calls are defined by the common elements so different languages can be used in the same program.

Complex statements and nested instructions are supported:

- Iteration loops (**REPEAT**-**UNTIL**; **WHILE**-**DO**)
- Conditional execution (**IF**-**THEN**-**ELSE**; **CASE**)
- Functions (**SQRT**(), **SIN**())

The ST editor's context menu has the following commands:

Menu Entry	Action
Select variable	Displays a dialog for inserting/selecting a variable
Select function	Displays a dialog for inserting/selecting function block
Open IEC 61131-3 variables	Open the IEC variables tool
Open IEC 61131-3 types	Open the IEC types tool
Edit local program variables	Open local variables editor
Breakpoints	Open breakpoints manager window
Toolbox	Open toolbox control, from where using drag and drop functions and function blocks can be added to the program



Editing FBD Programs

IEC 61131-3 FBD language is a graphical programming language. The FBD editor is a powerful graphical tool that enables you to enter and manages Function Block Diagrams according to the IEC 61131-3 standard. The editor supports advanced graphic features such as drag and drop, object resizing and connection lines routing features, so that you can rapidly and freely arrange the elements of your diagram. It also enables you to insert in a FBD diagram graphic elements of the LD (Ladder Diagram) language such as contacts and coils

A functional block diagram is a block diagram that describes a function between input variables and output variables. A function is described as a set of elementary blocks. Input and output variables are connected to blocks by connection lines. An output of a block may also be connected to an input of another block: Inputs and outputs of the blocks are wired together with connection lines, or links. Single lines may be used to connect two logical points of the diagram:

An input variable and an input of a block

An output of a block and an input of another block

An output of a block and an output variable

The connection is oriented, meaning that the line carries associated data from the left end to the right end. The left and right ends of the connection line must be of the same type.

Double-clicking on a contact or a coil displays a dialog for selecting the input/output for the element.

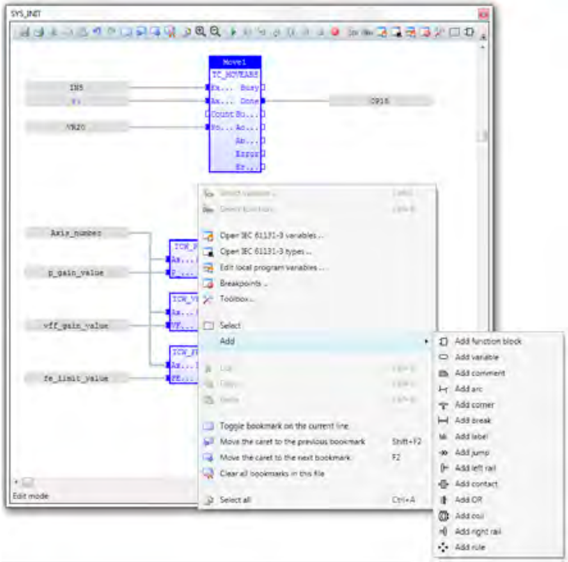
Double-clicking on a function/function block displays a dialog for selecting the function/functional block for the element.

The editor contents can be zoomed in and out via the toolbar buttons, or using the shortcut combinations “Ctrl +” for zoom in and “Ctrl -” for zoom out.

The FBD editor context menu has the following functionality:

Menu Entry	Action
Select variable	Displays a dialog for inserting/selecting a variable
Select function	Displays a dialog for inserting/selecting function block
Open IEC 61131-3 variables	Open the IEC variables tool
Open IEC 61131-3 types	Open the IEC types tool
Edit local program variables	Open local variables editor
Breakpoints	Open breakpoints manager window
Toolbox	Open toolbox control, from where using drag and drop functions and function blocks can be added to the program
Select	Enters in selection mode
Add function block	Enters in add function block mode
Add variable	Enters in add variable mode
Add comment	Enters in add comment mode
Add arc	Enters in add arc mode
Add corner	Enters in add corner mode

Menu Entry	Action
Add break	Enters in add break mode
Add label	Enters in add label mode
Add jump	Enters in add jump mode
Add left rail	Enters in add left rail mode
Add contact	Enters in add contact mode
Add OR	Enters in add OR mode
Add coil	Enters in add coil mode
Add right rail	Enters in add right rail mode
Add rule	Enters in add rule mode



Editing SFC Programs

IEC 61131-3 SFC language is a graphical programming language. Main components of SFC are:

- Steps with associated actions
- Transitions with associated logic conditions
- Directed links between steps and transitions

Steps in an SFC diagram can be active or inactive. Actions are only executed for active steps. A step can be active for one of two motives: (1) It is an initial step as specified by the programmer (2) It was activated during a scan cycle and not deactivated since

The editor contents can be zoomed in and out via the toolbar buttons, or using the shortcut combinations “Ctrl +” for zoom in and “Ctrl -” for zoom out.

The SFC editor context menu has the following functionality:

Menu Entry	Action
Open IEC 61131-3 variables	Open the IEC variables tool
Open IEC 61131-3 types	Open the IEC types tool
Edit local program variables	Open local variables editor
Breakpoints	Open breakpoints manager window
Insert step	Inserts a new step in the program
Insert transition	Inserts a new transition element in the program
Insert init step	Inserts an initialization step
Insert jump	Inserts a jump element in the program
Renumber	Renumbers the steps and transitions, starting from the selected one
Next item	Navigates to the next logical element of the program

SFC programs are divided into 2 levels:

LEVEL 1

level 1 is the main SFC chart, which describes the steps and transitions and is edited by the SFC editor.

A step represents a stable state. It is drawn as a square box in the SFC chart. At runtime a step can be either active or inactive. All actions linked to the steps are executed depending on the activity of the step. Initial steps represent the initial situation of the chart when program is started. There must be at least one initial step in each SFC chart.They are marked with a double line.

Transitions represent a condition the changes the program activity from one step to another. It is marked by a small horizontal line that crosses a link drawn between the two steps. The condition must be a **BOOL** expression. Transitions define the dynamic behaviour of the SFC chart, according to the following rules:

A transition in crossed if:

- its condition is **TRUE**.
- and if all steps linked to the top of the transition (before) are active.

When a transition is crossed:

- all steps linked to the top of the transition (before) are de-activated.
- all steps linked to the bottom of the transition (after) are activated.

DIVERGENCES

It is possible to link a step to several transitions and thus create a divergence. The divergence is represented by a horizontal line. Transitions after the divergence represent several possible changes in the situation of the program.

All conditions are considered as exclusive, according to a left to right priority order. It means that a transition is considered as **FALSE** if at least one of the transitions connected to the same divergence on its left side is **TRUE**

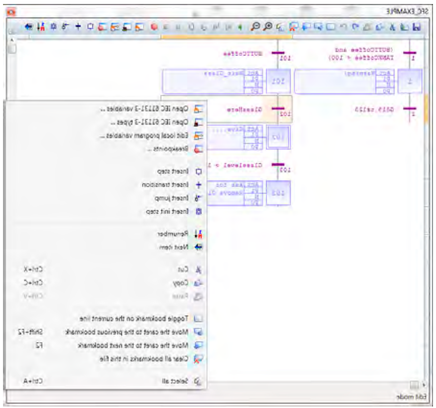
LEVEL 2

level 2 is the code for the actions, transitions and text for notes for level 1 elements

Each level 1 step has 5 level 2 elements, which can be open for editing by double-clicking on the corresponding element.

1. Actions – Simple actions entered as text
2. P1 actions, that can be programmed in **ST**,**LD** or **FBD**, are executed only once when the step becomes active
3. N actions, that can be programmed in **ST**,**LD** or **FBD**, are executed on each cycle while the step is active
4. P0 actions, that can be programmed in **ST**,**LD** or **FBD**, are executed only once when the step becomes inactive
5. Text notes

While a level 2 item is open for editing, the contents of the parent level 1 SFC program is locked for editing. This is done to prevent renumbering or deleting of the parent level 1 element, for which the level 2 editor is open. Once the editing of the level 2 element is complete, and the user closes the child editor, the SFC editor is unlocked and its normal operation is restored.



When editing a level 2 SFC program, an additional combo box will appear in the status bar of the program editor

From this combo box the language of the level 2 element can be chosen. The default is ST. When the language is changed, a prompt will appear, notifying that the current contents of the program will be cleared.

IEC Types Editor

The types editor tool is an editor, where the user can define, delete and modify custom types. It is a tab panel, which has 3 tabs : one for the IEC structures, one for the IEC enumerated types and one for the IEC bit fields.

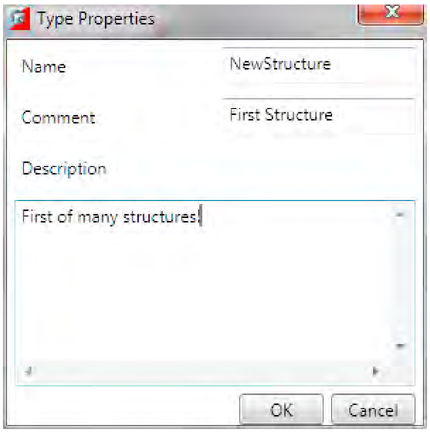
STRUCTURES TAB

The structures tab displays the custom structure types:

The description of the fields available for editing is the same as for the variables editor tool.

To add a new structure, press the “Insert new structure” button. To delete an existing structure, select it and press the “Delete” key on the keyboard, or press the “Remove” button.

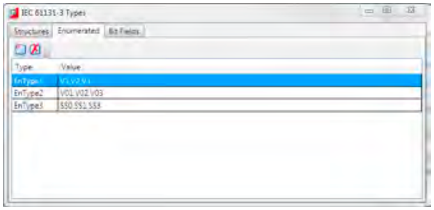
Double-click on a selected structure displays the “Type properties” dialog, where a type name, comment and description can be edited.



To add a new field in an existing structure, press the “Insert” key on the keyboard, or press the “Insert new variable” button. To delete an existing field in a structure, select it and press the “Delete” key on the keyboard, or press the “Remove” button.

ENUMERATED TAB

The enumerated tab displays the custom enumerated types:



This tab editor has 2 columns:

Column	Description
Name	The name for the enumerated type
Value	A coma separated list of symbolic values which will be the enumerated type values available for use in programs

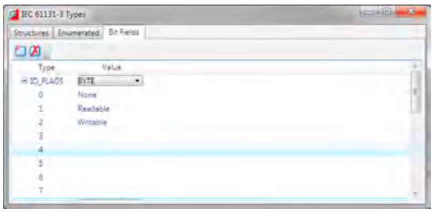
To add a new enumerated type, press the “Add new IEC type” button. To remove an existing enumerated type, select it and press the “Remove” button.

To edit the name of an existing enumerated type, double-click on the selected type’s Name column in the editor.

To edit the enumerated values, double-click on the selected type’s Value column.

1. BIT-FIELDS TAB

The bit-fields tab displays the custom bit-field types:



This tab editor has 2 columns:

Column	Description
Type	The name for the bit-field type. Below the name, is the list with bits(number of bits depends on the base type). The list can be expanded/collapsed via the “+” button in front of the type name.
Value	A combo box with the available base types for the bit-field type. Depending on the base type, the bit-field can have different number of bits. For example, a bit-field, based on INT, has 16 bits. A bit-field, based on SINT, has 8 bits. Each bit can be specified a symbolic name for use in code. For example, user-friendly names can be assigned, like “Shared”, “None”, etc.

To add a new bit-field type, press the “Add new IEC type” button. To remove an existing bit-field type, select it and press the “Remove” button.

To edit the name of an existing bit-field type, double-click on the selected type’s Type column in the editor.

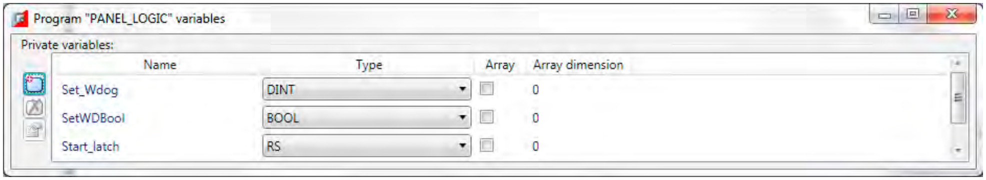
To change the base type of the selected bit-field type, use the combo box with available types.

To edit the bit-field names, double-click on the selected bit-field bit in the value column.

Program Local Variables

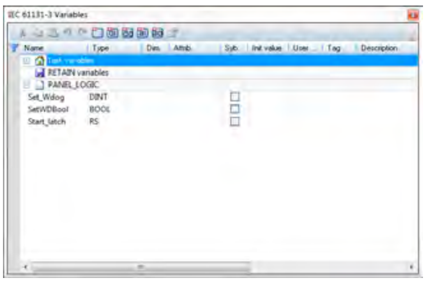
All IEC programs have local variables, which are “private” to the programs only. User defined function block programs, have also input and output variables, which are also local program variables.

The editor for the local variables, provides an easy way of adding/removing and setting properties of local variables.



For normal IEC programs, only the “Private variables” are available. For FBD programs additional sections for “Input Variables” and “Output Variables” are available.

Variable Editor



The Variable Editor displays all the variables that are in use in the IEC task. The variables are grouped in variables groups. There are 2 predefined variables groups – the “Task” and “Retain” variables. Then for each IEC program in the IEC task, a variable group with the same name as the program exists.

Variables in the “Task” group are accessible from all programs. The values of the variables in the “Retain” group are stored upon IEC execution stop and are restored upon next start of the IEC executable. The variables in the rest of the groups are "private" for the corresponding program.

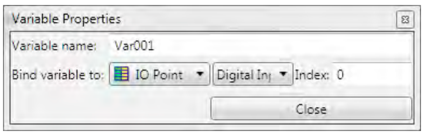
A new variable can be added, by selecting the corresponding group, and pressing the "Insert" key on the keyboard. A new variable will be inserted in the selected group and will have default name, type, initial value, etc.

The variable has the following properties, which are separated as columns in the variables editor:

Property	Description
Name	The name of the variable. To edit this property, double-click on it.
Type	The type of variable. Can be some of the predefined IEC types, or some user-defined type. To edit this property, double-click on it.
Dim	Dimensions of the variable. For example, arrays are created by specifying the size of the array in this field. To edit this property, double-click on it.
Attrib	Attributes of the variable. Depends on the variable type and profile. For example, an IO-mapped. To edit this property, double-click on it. variable can have the "Read-only" attribute set.
Syb	Embed variable symbol. Not supported(On-line change must be enabled). To edit this property, double-click on it.
Init value	The initial value of the variable, depending on its type. To edit this property, double-click on it.
User group	The user can specify additional grouping for a variable. To edit this property, double-click on it.
Tag	A short comment text for the variable. To edit this property, double-click on it.
Description	A long comment text for the variable. To edit this property, double-click on it.

Each variable has a set of properties attached. The properties editing dialog is displayed, when a variable is selected and the properties toolbar button is pressed, of from the context menu for the selected variable.

VARIABLE PROPERTIES EDITING



The Variables Properties dialog provides an editable text box, where the user can change the name of the variable and its mapping (if any) physical memory or I/O on the controller, by selecting one of the binding methods.

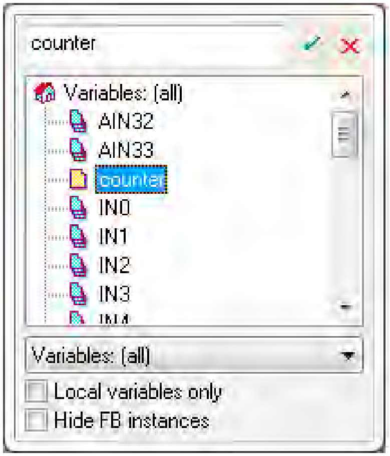
Property	Description
None	default – the variable is not mapped to anything
IO Point	the variable can be mapped to a Digital or Analogue Input or Output, by specifying the I/O point index

Property	Description
TABLE	the variable can be mapped to a TABLE location, by specifying the index in the table memory
VR	the variable can be mapped to a VR variable, by specifying the index in the VR memory

Selecting or Inserting a Variable

This applies to **ST**, **LD** and **FBD** programs.

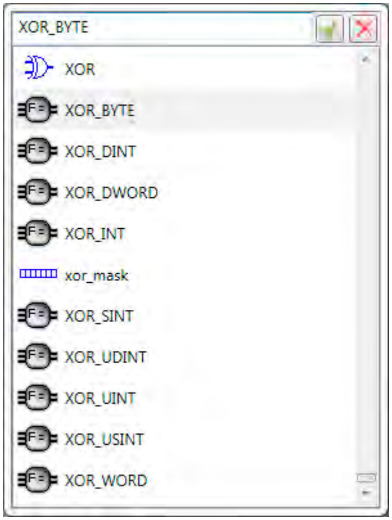
When the "Select variable" command is chosen from the context menu, a popup dialog appears in which the user can select an existing variable to replace the variable in the current selection, or to create a new variable. Type the name of the variable into the edit box and, if the variable already exists in the current scope, it will be selected. Pressing the Enter key, or the small green check on the dialog will replace the variable with the selected one. If a variable with the typed name does not already exist, a prompt will appear for creating this variable, setting its type and group.



Selecting or Inserting a Function Block

This applies to **ST**, **LD** and **FBD** programs.

When the "Select function" command is chosen from the context menu, a popup dialog appears, where the user can select from a list of available functions and function blocks. Type the name or symbol of the function/function block into the edit box and if it exists, it will be selected in the list. Pressing the Enter key or the small green check box will replace/insert the selected function in the editor with the selected one from the list box.



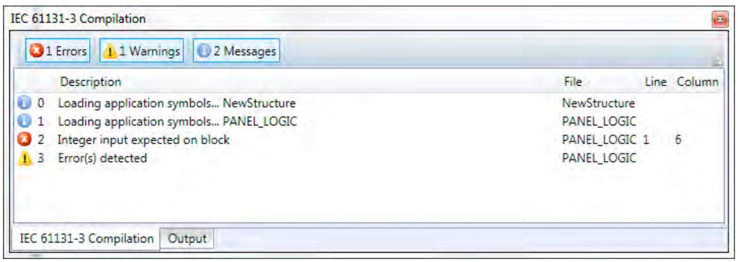
Compiling

When the “Build” command is executed, the “IEC 61131-3 Compilation” tool is automatically displayed. It contains a list with the build results from compiling the IEC task into an executable.

If the project compilation have been successful, there should be no errors, and the executable is downloaded on the controller.

If any errors occurred, the error description is displayed as a hint, so the error can be removed by the user.

Double-clicking on an item opens the source editor, relevant to the item. In the example below, double-clicking on the second line(Variable, constant expression or function call expected), will open an editor for the “LADDER1” program, and will position the caret on line 1, column 9 (which is the source of the error).



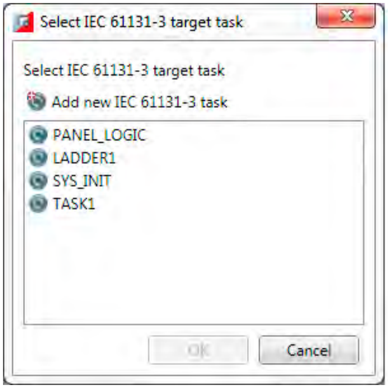
To show and hide different types of messages, the user can use the “Errors”, “Warnings” and “Messages” buttons respectively.

Running and Debugging a Program

When an IEC task is compiled, it can be executed by several ways:

1. From the toolbar of the IEC item in the project tree
2. From the context menu of the IEC item in the project tree
3. From the toolbar of some of the IEC programs, belonging to that IEC task
4. From the command line, by typing "**RUN** <IEC task name>"
5. From a BASIC program, using **RUN** basic command

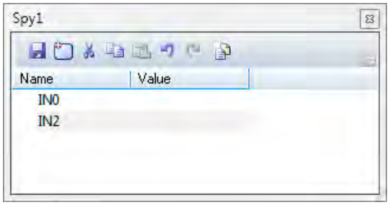
It is possible that an IEC task can be started more than once (e.g. from a **BASIC** program) but this is not a typical scenario. *Motion* Perfect's support for IEC programs is designed in a way that only one instance of an IEC task can be debugged at a time. Different IEC tasks can be debugged simultaneously, however, when connecting to a controller with more than one instance of the same IEC task running, *Motion* Perfect will prompt to which instance the debugger should connect.



It is also possible to set an IEC task to automatically start when the controller boots up, from the context menu of the IEC task, selecting the command "Set **AUTORUN**", or using the standard command **RUNTYPE**.

Spy List window

A spy list window can be opened for each spy list, defined in the IEC task by double-clicking on the spy list in the project tree, or from its context menu.



The Spy List is a list of variables and their values:

Column	Description
Name	The name of the variable to be spied
Value	The current value of the variable being spied

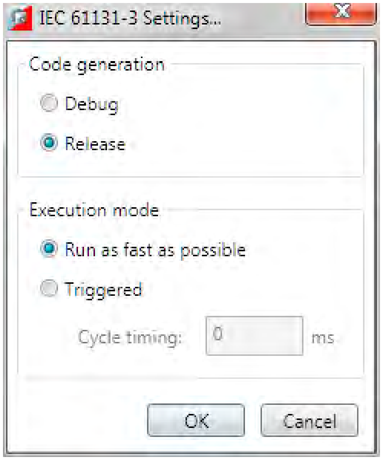
To add a new variable directly to the list of variables, drag and drop from an open editor, or the variables editor, or the structures editor. Alternatively press the “Insert” key on the keyboard or click on the Add Variable button in the toolbar, which will pop-up a dialog allowing the user to select the variable from a list.

To remove a variable from the list, select the variable, and press the “Delete” key on the keyboard.

As spy lists are part of the IEC task, when variables are added to, or removed from a spy list, the IEC task has to be recompiled.

IEC Settings

The IEC Settings dialog can be accessed from the context menu of task in the Controller or Project Tree. It allows the user to adjust what type of code is generated and how it is run.



CODE GENERATION

The Code Generation setting controls which type of code is produced:

- Debug Code: allows the user to use Spy Lists to view variables and to step through the code in order to debug it. The generated code is larger and will run more slowly than the release code.
- Release Code: contains no debugging information.

EXECUTION MODE

This determines how the code is executed:

- Run as fast as possible: Cycles are executed with the fastest possible speed of the hardware platform.
- Triggered: Cycles are executed with respect to the specified cycle time. The cycle time is the time between 2 consecutive cycles, in milliseconds. If for example, the user wants to execute code twice each second, the cycle timing should be specified as 500 ms(here the time needed for executing the instructions is ignored. In real-world scenarios, more precise timing might be needed)

7 MC400 SIMULATOR

Introduction to MC400 Simulator

The MC400 is a Microsoft Windows™ based application for the PC, designed to be used in conjunction with Trio Motion Technology's *Motion* Perfect development software. It provides a software simulation of one of Trio Motion Technology's series 4 range of multi-tasking motion controllers.

Running the Simulator

USING STORED CONNECTION PARAMETERS

To run the simulator, select "**Triomotion/MC400 Simulator**" from the "All programs" menu. This will cause both the simulator GUI and the simulator process to start up. The connection parameters used will be those last set in the application's "Options" dialog, of default parameters if none have been set.

The simulator consists of a **GUI** which is always running and a simulator process which mimics the internal processing of a real controller. The simulator process can be started and stopped by the user using the context menu.

Whilst the simulator process is running it is possible to connect to the simulator using an application such as *Motion* Perfect using a local Ethernet port (see Communications).

SPECIFYING CONNECTION PARAMETERS

If the application is run from the command line, the connection parameters may be specified as follows:

```
ExeFile MPE _ Port REMOTE _ Port HMI _ Port Flash _ File SD _ Card _ Dir
```

WHERE:

ExeFile is the full or relative path to the MC400Simulator executable file.

MPE_Port is the IP port used for communications with *Motion* Perfect (default 23).

REMOTE_Port is the IP port used for communications with the Trio PC Motion ActiveX control (default 3240).

HMI_Port is the IP port used for communications with an HMI device (default 10000).

Flash_File is the file which holds the image of the virtual flash memory.

SD_Card_Dir is the directory used for SD Memory Card images.

Starting the simulator using command line parameters allows more than one instance to run at the same time as long as the instances have different parameters from any other running instance.

IF AN INSTANCE USE ONE OR MORE PARAMETERS THE SAME AS THOSE USED BY A DIFFERENT INSTANCE THERE MAY BE CONNECTION PROBLEMS AND/OR CORRUPTION OF THE FLASH AND SD-CARD STORED DATA.

Communications

Communication between an application (such as Trio Motion Technology's *Motion* Perfect) and to simulator is done using a local Ethernet connection. The simulation acts a local server with the following parameters:

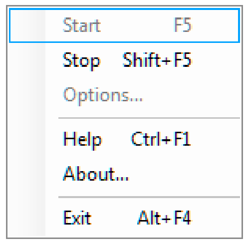
IP Address	127.0.0.1 (localhost)
Command Port	23
Token Port	3240

The command port is used for programs such as *Motion* Perfect.

The Token port is used with the Trio PC Motion ActiveX control.

Context Menu

The context menu is displayed when the user right-clicks on the main application window.



START

Starts the simulation process (only available when the simulation process is not running). This is equivalent to powering on the controller.

Only available when the simulator is stopped.

STOP

Stops the simulation process (only available when the simulation process is running). This is equivalent to powering off the controller.

Only available when the simulator is running.

HELP

Displays the help file.

OPTIONS

Displays the options for the simulator.

Only available when the simulator is stopped.

ABOUT

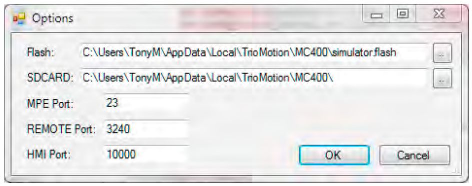
Displays information about this version of the simulator.

EXIT

Terminates the simulator program (both the simulator process and the GUI).

Options

The options dialog allows the user to set up the IP ports used for communications and the files used for saving images of the virtual flash memory and SD Memory Card.



FLASH

The file which holds the image of the virtual flash memory.

SDCARD

The directory used for SD Memory Card images.

MPE PORT

The IP port used for communications with *Motion* Perfect (default 23).

REMOTE PORT

The IP port used for communications with the Trio PC Motion ActiveX control (default 3240).

HMI PORT

The IP port used for communications with an HMI device (default 10000).

8

PC MOTION ACTIVEX CONTROL

TrioPC Motion ActiveX Control

The TrioPC ActiveX component provides a direct connection to the Trio MC controllers via a 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 USB and/or ethernet network support
- Windows XP, Windows Vista (32 bit verions) or Windows 7 (32 bit versions)
- Trio PCI driver - for PCI based *Motion Coordinator*s
- Trio USB driver - for *Motion Coordinator* with a USB interface.
- Knowledge of the Trio *Motion Coordinator* to which the TrioPC ActiveX controls will connect.
- Knowledge of the TrioBASIC programming language.

INSTALLATION OF THE ACTIVEX COMPONENT

The component and auxiliary documentation is provided as an MSI installer package. Double clicking on the .msi file will start the install process. It is recommended that any previous version should be uninstalled before the install process is initiated. The installer also installs the Trio USB and Trio PCI drivers and registers the ActiveX component.

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 Tools then Choose Toolbox Items.

When the Choose Toolbox Items dialogue box has opened, select the COM components tab, then scroll down until you find “TrioPC Control” 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 TrioPC Control.

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, Serial, 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 for 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:	See Connection Type.
Short PortMode:	See Communications Mode.

RETURN VALUE:
Boolean; **TRUE** if the connection is successfully established. For a USB connection, this means the Trio USB driver is active (an MC with a USB interface 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(PortId)`

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
    frmMain.Refresh
EndSub
```

IsOpen

DESCRIPTION:
Returns the state of the connection between the TrioPC ActiveX control and the *Motion Coordinator*.

SYNTAX:
`IsOpen(PortMode)`

PARAMETERS:

Short PortMode:	See Communications Mode.
Return Value:	Boolean; TRUE if the connection is open, FALSE if it is not .

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 IPV4 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:

String host:	host IP address as string (eg “192.168.0.250”).
Return Value:	None

EXAMPLE:
```
Rem Set up the Ethernet IPV4 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, See Connection Type.

EXAMPLE:
```
Rem Open a Synchronous connection
ConnectError = False
TrioPC _ Status = TrioPC1.Open(0, 0)
ConnectionType = TrioPC1.GetConnectionType()
```

```
If ConnectionType <> 0 Then
    ConnectError = True
End If
frmMain.Refresh
```

Properties

Board

DESCRIPTION

Sets the board number used to access a PCI card.

The PCI cards in a PC are always enumerated sequentially starting at 0. It must be set before the **OPEN** command is used.

TYPE:

Long

ACCESSREAD / 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 setting the IPV4 host address used to access a *Motion Coordinator* over 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
```

CmdProtocol

DESCRIPTION:
Used to specify the version of the ethernet communications protocol to use to be compatible with the firmware in the ethernet daughterboard. The following values should be used:

0: for ethernet daughterboard firmware version 1.0.4.0 or earlier.

1: for ethernet daughterboard firmware version 1.0.4.1 or later.

TYPE:
Long

ACCESS:
Read / Write

DEFAULT VALUE:
1

EXAMPLE:
```
Rem Set ethernet protocol for firmware 1.0.4.0
TrioPC.CmdProtocol = 0
```

Users of older daughterboards will need to update their programs to set the value of this proporty to 0.

FlushBeforeWrite

DESCRIPTION:
The USB and serial communications interfaces are error prone in electrically noisy environments. This means that spurious characters can be received on these interfaces which will cause errors in the OCX. If FlushBeforeWrite is non-zero then the OCX will flush the communications interface before sending a new request, so minimizing the consequences of a noisy environment. The flush routine clears the current contents of the communications buffer and waits 100ms to make sure that there are no other pending characters coming in.

TYPE:
Long

ACCESS:
Read / write

EXAMPLE:
```
    TrioPC1.FlushBeforeWrite = 0
```

FastSerialMode

DESCRIPTION:
The Trio *Motion Coordinator* have two standard RS232 communications modes: slow and fast. The slow mode has parameters 9600,7,e,1 whereas the fast mode has parameters 38400,8,e,1. If FastSerialMode is **FALSE** then the RS232 connection will use the slow mode parameters. If the FastSerialMode is **TRUE** then the RS232 connection will use the fast mode parameters.

ACCESS:
Read / write

TYPE:
Boolean

EXAMPLE:
```
    TrioPC1.FastSerialMode = True
```

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.
Double Distance:	Distance to be moved, can be a single numeric value or an array of numeric values that contain at least Axes values.
Short Axis:	Optional parameters that must be a single numeric value that specifies the base axis for this move.

RETURN VALUE:

See 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.
Short 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:

See TrioPC STATUS.

MoveAbs

DESCRIPTION:

Performs the corresponding **MOVEABS**(...) **AXIS**(...) command on the.

SYNTAX:

MoveAbs(Axes, Distance, [Axis])

PARAMETERS:

short Axes:	Number of axes involved in the **MOVEABS** command.
Double Distance:	Absolute position(s) that specify where the move must terminate. This can be a single numeric value or an array of numeric values that contain at least Axes values.
Short Axis:	Optional parameters that must be a single numeric value that specifies the base axis for this move.

RETURN VALUE:

See TrioPC STATUS.

MoveCirc

DESCRIPTION:

Performs the corresponding **MOVECIRC**(...) **AXIS**(...) command on the *Motion Coordinator*.

SYNTAX:

MoveCirc(EndBase, EndNext, CentreBase, CentreNext, Direction, [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.
Short Axis:	Optional parameters that must be a single numeric value that specifies the base axis for this move.

RETURN VALUE:
See 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.
short Axis:	Optional parameters that must be a single numeric value that specifies the base axis for this move.

RETURN VALUE:
See TrioPC STATUS.

CamBox

DESCRIPTION:
Performs the corresponding **CAMBOX**(…) command on the *Motion Coordinator*.

SYNTAX:
```
CamBox(TableStart, TableStop, Multiplier, LinkDist, LinkAxis, LinkOption, 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 LinkOption:	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. 8. Pattern Mode. 32. Link is only active during positive moves.
Double LinkPos:	The absolute position on the link axis where the cam will start.
Short Axis:	Optional parameters that must be a single numeric value that specifies the base axis for this move.

RETURN VALUE:
See 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.
Short Axis:	Optional parameters that must be a single numeric value that specifies the base axis for this move.

RETURN VALUE:
See 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.
Short Axis:	Optional parameters that must be a single numeric value that specifies the base axis for this move.

RETURN VALUE:
See 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.
Short Axis:	Optional parameters that must be a single numeric value that specifies the base axis for this move.

RETURN VALUE:
See TrioPC STATUS.

Datum

DESCRIPTION:
Performs the corresponding **DATUM**(…) **AXIS**(…) command on the *Motion Coordinator*.

SYNTAX:
`Datum(Sequence, [Axis])`

PARAMETERS:

The type of datum procedure to be performed:

Short sequence: 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.

Short Axis:

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.

Optional parameters that must be a single numeric value that specifies the base axis for this move

RETURN VALUE:

See TrioPC STATUS.

Forward

DESCRIPTION:

Performs the corresponding **FORWARD**(...) **AXIS**(...) command on the *Motion Coordinator*.

SYNTAX:

Forward([Axis])

PARAMETER:

Short Axis: Optional parameters that must be a single numeric value that specifies the base axis for this move.

RETURN VALUE:
See TrioPC STATUS.

Reverse

DESCRIPTION:
Performs the corresponding **REVERSE**(...) **AXIS**(...) command on the *Motion Coordinator*.

SYNTAX:
Reverse([Axis])

PARAMETERS:

Short Axis:	Optional parameters that must be a single numeric value that specifies the base axis for this move.

RETURN VALUE:
See 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.
Short Axis:	Optional parameters that must be a single numeric value that specifies the base axis for this move.

RETURN VALUE:
See TrioPC STATUS.

MoveLink

DESCRIPTION:
Performs the corresponding **MOVELINK**(...) **AXIS**(...) command on the *Motion Coordinator*.

SYNTAX:
`MoveLink(Distance, LinkDistance, LinkAcceleration, LinkDeceleration, LinkAxis, LinkOptions, LinkPosition, [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). **32** Link is only activee during positive moves on the link axis.
Double LinkPosition:	The absolute position on the link axis where the move will start.
Short Axis:	Optional parameters that must be a single numeric value that specifies the base axis for this move.

RETURN VALUE:
See 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.
Short Axis:	Optional parameters that must be a single numeric value that specifies the base axis for this move.

RETURN VALUE:

See TrioPC STATUS.

RapidStop

DESCRIPTION:

Performs the corresponding **RAPIDSTOP**(...) command on the *Motion Coordinator*.

PARAMETERS:

None

RETURN VALUE:

See TrioPC STATUS.

Process Control Commands

Run

DESCRIPTION:
Performs the corresponding RUN(...) command on the *Motion Coordinator*.

SYNTAX:
`Run(Program, Process)`

PARAMETERS:

String Program:	String that specifies the name of the program to be run.
Short Process:	Optional parameter that must be a single numeric value that specifies the process on which to run this program.

RETURN VALUE:
See TrioPC STATUS.

Stop

DESCRIPTION:
Performs the corresponding **STOP**(...) command on the *Motion Coordinator*.

SYNTAX:
`Stop(Program, Process)`

PARAMETERS:

String Program:	String that specifies the name of the program to be stopped.
Short Process:	Optional parameter that must be a single numeric value that specifies the process on which the program is running.

RETURN VALUE:
See 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.
Double 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:
See 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:

String Variable:	Name of the system variable to read.
Double Value:	Variable in which to store the value read.

RETURN VALUE:
See TrioPC STATUS.

GetVr

DESCRIPTION:
Returns the current value of the specified VR 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:
See 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.
Double 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:
See 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:

String Variable:	Name of the system variable to write.
Double Value:	Variable in which the value to write is stored.

RETURN VALUE:
See TrioPC STATUS.

SetVr

DESCRIPTION:
Sets the value of the specified Global variable.

SYNTAX:
`SetVr(Variable, Value)`

PARAMETERS:

Short Variable:	Number of the VR variable to write.
Double Value:	Variable in which the value to write is stored.

RETURN VALUE:
See TrioPC STATUS.

GetProcessVariable

DESCRIPTION:
Returns the current value of a variable from a currently running process. It is quite difficult to calculate the VariableIndex as the storage for the named variables is assigned during the program compilation, but it is not stored due to memory restrictions on the *Motion Coordinator*s. To make things worse, if a program is modified in such a way the named variables it uses are changed (added, removed, or changed in order of use) then the indices may change.

SYNTAX:

`GetProcessVariable(VariableIndex, Process, Value)`

PARAMETERS:

Short VariableIndex:	The index of the variable in the process variables table.
Short Process:	The process number of the running process.
Double Value:	Variable in which to store the value read.

EXAMPLE:

Let us assume that there is the program “T1” on the *Motion Coordinator* which has the following contents:

```
y=2
x=1
```

If this program is run on process 1 by the command RUN “T1”,1 then we could use the following code in VisualBASIC to read the contents of the x and y variables.

```
Dim x As Double
Dim y As Double
If Not AxTrioPC1.GetProcessVariable(1, 1, x) Then Exit Sub
If Not AxTrioPC1.GetProcessVariable(0, 1, y) Then Exit Sub
MsgBox(“X has value “ + Format(x))
MsgBox(“Y has value “ + Format(y))
```

RETURN VALUE:

See TrioPC STATUS.

GetAxisVariable

DESCRIPTION:

For a system variable that accepts the **AXIS** modifier this method will return the value of the that system variable on the given axis. If the system variable does not exist, or does not accept the **AXIS** modifier, then this method will fail.

SYNTAX:

`GetAxisVariable(VariableIndex, Axis, Value)`

PARAMETERS:

String Variable:	The name of the variable.
Short Axis:	The axis number.
Double Value:	Variable in which to store the value read.

RETURN VALUE:
See TrioPC STATUS.

SetAxisVariable

DESCRIPTION:
For a system variable that accepts the **AXIS** modifier this method will set the value of the that system variable on the given axis. If the system variable does not exist, or does not accept the **AXIS** modifier, then this method will fail.

SYNTAX:
SetAxisVariable(VariableIndex, Axis, Value)

PARAMETERS:

String Variable:	The name of the variable.
Short Axis:	The axis number.
Double Value:	Value to set.

RETURN VALUE:
See TrioPC STATUS.

GetProcVariable

DESCRIPTION:
For a system variable that accepts the **PROC** modifier this method will return the value of the that system variable on the given process. If the system variable does not exist, or does not accept the **PROC** modifier, then this method will fail.

SYNTAX:
GetProcVariable(Variable, Process, Value)

PARAMETERS:

String Variable:	The name of the variable.
Short Process:	The process number of the running process.
Double Value:	Variable in which to store the value read.

RETURN VALUE:
See TrioPC STATUS.

SetProcVariable

DESCRIPTION:
For a system variable that accepts the **PROC** modifier this method will set the value of the that system variable on the given process. If the system variable does not exist, or does not accept the **PROC** modifier, then this method will fail.

SYNTAX:
`SetProcVariable(Variable, Process, Value)`

PARAMETERS:

String Variable:	The name of the variable.
Short Process:	The process number of the running process.
Double Value:	Value to set.

RETURN VALUE:
See TrioPC STATUS.

GetSlotVariable

DESCRIPTION:
For a system variable that accepts the **SLOT** modifier this method will return the value of the that system variable on the given slot. If the system variable does not exist, or does not accept the **SLOT** modifier, then this method will fail.

SYNTAX:
`GetSlotVariable(Variable, Slot, Value)`

PARAMETERS:

String Variable:	The name of the variable.
Short Slot:	The slot number.
Double Value:	Variable in which to store the value read.

RETURN VALUE:
See TrioPC STATUS.

SetSlotVariable

DESCRIPTION:
For a system variable that accepts the **SLOT** modifier this method will set the value of the that system variable on the given slot. If the system variable does not exist, or does not accept the **SLOT** modifier, then this method will fail.

SYNTAX:
`SetSlotVariable(Variable, Slot, Value)`

PARAMETERS:

String Variable:	The name of the variable.
Short Slot:	The slot number.
Double Value:	Value to set.

RETURN VALUE:
See TrioPC STATUS.

GetPortVariable

DESCRIPTION:
For a system variable that accepts the **PORT** modifier this method will return the value of the that system variable on the given port. If the system variable does not exist, or does not accept the **PORT** modifier, then this method will fail.

SYNTAX:
`GetPortVariable(Variable, Port, Value)`

PARAMETERS:

String Variable:	The name of the variable.
Short Port:	The port number.
Double Value:	Variable in which to store the value read.

RETURN VALUE:
See TrioPC STATUS.

SetPortVariable

DESCRIPTION:
For a system variable that accepts the **PORT** modifier this method will set the value of the that system variable on the given port. If the system variable does not exist, or does not accept the **PORT** modifier, then this method will fail.

SYNTAX:
`SetPortVariable(Variable, Port, Value)`

PARAMETERS:

String Variable:	The name of the variable.
Short Port:	The port number.
Double Value:	Value to set.

RETURN VALUE:
See 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:
See 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.
Short Value:	Variable in which to store the value read.

RETURN VALUE:
See 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:
See 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:
See 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:
See 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:
See TrioPC STATUS.

Mark

DESCRIPTION:
Performs the corresponding **MARK**(...) command on the *Motion Coordinator*.

SYNTAX:
`Mark(Axis, Value)`

PARAMETERS:

Short Axis number:	Axis number.
Short Value:	The stored capture value for a registration first event.

RETURN VALUE:
See TrioPC STATUS. **FALSE** if no value has been captured (no registration first event has occurred).

MarkB

DESCRIPTION:
Performs the corresponding **MARKB**(...) command on the *Motion Coordinator*.

SYNTAX:
`MarkB(Axis, Value)`

PARAMETERS:

Short Axis number:	Axis number.
Short Value:	The stored capture value for a registration second event.

RETURN VALUE:
See TrioPC STATUS. **FALSE** if no value has been captured (no registration second event has occurred).

Op

DESCRIPTION:
Performs the corresponding OP(...) command on the *Motion Coordinator*.

SYNTAX:
`Op(Output, [State])`

PARAMETERS:

Long 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.
Short State:	Optional numeric value that specifies the desired status of the output, 0 implies off, not-0 implies on.

RETURN VALUE:
See 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.
Short Axis:	Optional numeric value that specifies the base axis for this command.
Short OutputNumber:	Optional numeric value that specifies the number of the output to set.
Short OutputStatus:	Optional numeric value that specifies the signalled status of the output, 0 implies off, not-0 implies on.
Double SetPosition:	Optional numeric value that specifies the position at which to signal the output.
Double ResetPosition:	Optional numeric value that specifies the position at which to reset the output.

RETURN VALUE:
See 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:
See TrioPC STATUS.

Record

DESCRIPTION:
This method is no longer supported by any current *Motion Coordinator*.

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. Unused.
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.

Double Dist: Only used in pattern recognition mode and specifies the distance over which to record the transitions.

RETURN VALUE:
See 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:

See 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).
Short Port:	Optional numeric value that specifies the port to set (0..3).
Short Control:	Optional numeric value that specifies whether to enable or disable handshaking on this port.

RETURN VALUE:

See TrioPC STATUS.

General commands

Execute

DESCRIPTION:
Performs the corresponding **EXECUTE**... command on the *Motion Coordinator*.

SYNTAX:
`Execute(Command)`

PARAMETERS:
String Command: String that contains a valid TrioBASIC command.

RETURN VALUE:
Boolean; **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. Otherwise **FALSE**.

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 (0,5,6,7, or 9).
String data:	data received by the control from the *Motion Coordinator*.

RETURN VALUE:
Boolean; **TRUE** - if the given channel is valid, the connection open and the data read correctly from the buffer. Otherwise **FALSE**.

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 (0,5,6,7, or 9).
String data:	data to be written to the *Motion Coordinator*.

RETURN VALUE:
Boolean; **TRUE** - if the given channel is valid, the connection open, and the data written out correctly. Otherwise **FALSE**.

Scope

DESCRIPTION:
Initialises the data capture system in the *Motion Coordinator* for future data capture on a trigger event by executing a **SCOPE** command on the *Motion Coordinator*. A trigger event occurrs when the *Motion Coordinator* executes a **TRIGGER** command.

SYNTAX:
`Scope(OnOff, [SamplePeriod, TableStart, TableEnd, CaptureParams])`

PARAMETERS:

Boolean OnOff:	TRUE to set up and enable data capture, FALSE to disable it.
Long SamplePeriod:	Data sample period (in servo periods).
Long TableStart:	The table index for the start of the block of TABLE memory which will be used to hold captured data.
Long TableEnd:	The table index for the start of the block of TABLE memory which will be used to hold captured data.
String CaptureParams:	A string of up to 4 comma seperated parameters to capture.

EXAMPLE:

```
Rem Set up to capture MPOS and DOPS on axis 5
TrioPC _ Status = TrioPC1.Scope(True, 10, 0, 1000, “MPOS AXIS(5), DPOS
```

`AXIS(5)””)`

RETURN VALUE:
See TrioPC STATUS.

Trigger

DESCRIPTION:
Sends a **TRIGGER** command to the *Motion Coordinator* to start data capture previously configured using a **SCOPE** command.

SYNTAX:
`Trigger()`

PARAMETERS:
None.

RETURN VALUE:
See TrioPC STATUS.

Events

OnBufferOverrunChannel0/5/6/7/9

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()`
The channel number (x) can be any of the following: 0, 5, 6, 7 or 9.

PARAMETERS:
None.

RETURN VALUE:
None.

OnReceiveChannel0/5/6/7/9

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()`
The channel number (x) can be any of the following: 0, 5, 6, 7 or 9.

PARAMETERS:
None.

RETURN VALUE:
None.

OnProgress

DESCRIPTION:

The file operations LoadProgram, LoadProject and LoadSystem can take a long time to complete. To give some feedback on this process the OnProgress event is fired periodically during the file operation.

SYNTAX:

`OnOnProgress`

PARAMETERS:

Description:	Textual description of the associated process
Percentage:	Progress of the process in percent.

Intelligent Drive Commands

MechatroLink

DESCRIPTION:
Performs the corresponding **MECHATROLINK**(...) command on the *Motion Coordinator*. For more information on the **MECHATROLINK** command please see the corresponding *Motion Coordinator* user manual. This method will only work on those *Motion Coordinator*s that support the MehchatroLink interface.

SYNTAX:
MechatroLink(Module, Function, NumberOfParameters, MLParameters, Result)

PARAMETERS:

Short Module:	Number of the MechatroLink interface module.
Short Function:	MechatroLink function number.
Short NumberOfParameters:	Number of parameters to use in the MECHATROLINK command.
Double MLParameters:	Array of parameters to use for the MECHATROLINK command.
Double Result:	Variable in which the return value is stored.

RETURN VALUE:
See TrioPC STATUS.

Program Manipulation Commands

LoadProject

DESCRIPTION:
Not implemented.

LoadSystem

DESCRIPTION:
Not implemented.

LoadProgram

DESCRIPTION:
Not implemented.

New

DESCRIPTION:
Deletes a program on the *Motion Coordinator*.

SYNTAX:
`New(Program)`

PARAMETERS:

String Program:	The name of the program to be deleted.

RETURN VALUE:
See TrioPC STATUS.

Select

DESCRIPTION:
Selects a program on the *Motion Coordinator*.

SYNTAX:
`Select(Program)`

PARAMETERS:

String Program:	The name of the program to be selected.

RETURN VALUE:
See TrioPC STATUS.

Dir

DESCRIPTION:
Gets a directory listing from the *Motion Coordinator*.

SYNTAX:
`Dir(Directory)`

PARAMETERS:

String Program:	A string object used to return the directory listing.

RETURN VALUE:
See TrioPC STATUS.

InsertLine

DESCRIPTION:
Inserts a line into a program onto the *Motion Coordinator*. This will first Select the given program on the controller and then insert the line text at the given line number.

SYNTAX:
`InsertLine(Program, Line, LineText)`

PARAMETERS:

String Program:	The name of the program.
Short Line:	The line number at which the new line will be inserted.
String LineText:	The text of the line to be inserted.

RETURN VALUE:

See TrioPC STATUS.

Data Types

The following data types are used by the PC Motion control interface:

Connection Type

ALSO KNOWN AS:
Port Type.

DESCRIPTION:
An enumeration representing communication port type.

Values:	-1:	No connection .
	0:	USB.
	1:	Serial.
	2:	Ethernet.
	3:	PCI.
	4:	Path.
	5:	FINS (Not used on Trio controllers).

Communications Mode

ALSO KNOWN AS:
Port Mode.

DESCRIPTION:
An enumeration representing the operating mode of a communications link.

VALUES:

Interface	Mode	Description
USB:	0	Synchronous.
	1	Asynchronous.
Serial:	>0	Synchronous on specified port number.
	<0	Asynchronous on specified port number.
Ethernet:	0 3240	Synchronous on specified port number.
	23	Asynchronous on specified port number (default 23).

	other	
PCI:	0	Synchronous.
	1	Asynchronous.

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.

9 AUTO LOADER AND MCLOADER ACTIVEX

Project Autoloader

Trio Project Autoloader is a stand-alone program to load projects created using *Motion* Perfect 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 controller 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 primarily 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 by default is in the LoaderFiles directory. The commands are executed in sequence until either the script completes or an error occurs.

PROJECT

The project to be loaded using **LOADPROJECT** or **FASTLOADPROJECT** is in the form of a normal Motion Perfect 2 project. This consists of a directory containing a project definition file and Trio **BASIC** 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.

TIMEOUT

If there are large programs in the project the command timeout may need to be increased from its default value of 10 seconds otherwise the project load may fail due to the long time it takes to select a new program on the controller. The **TIMEOUT** command should appear in the script file before any **LOADPROJECT** command.

TABLES

Any tables to be loaded must be in the form of *.lst files produced by *Motion* Perfect.

Normally these table files will be in the LoaderFiles directory.

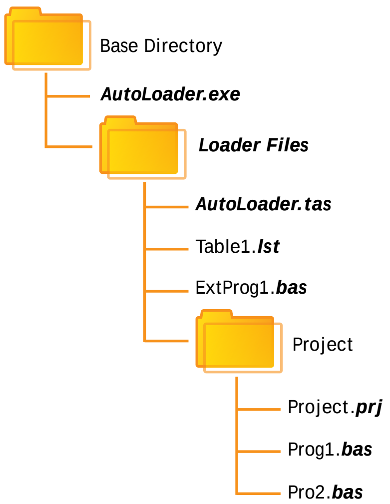
EXTRA PROGRAMS

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

By default the autoloader is designed to work with the following file structure (fixed names are shown in bold type).



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.

Any or all of the objects in the LoaderFiles directory can be located elsewhere as long as the file (or directory) name is specified using a full path. The script file can be specified as a single argument to the AutoLoader program.

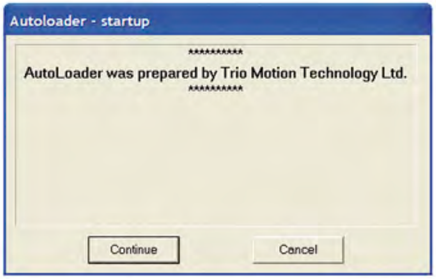
RUNNING THE PROGRAM

The program can be started in the same way as any other Windows program, in which case the LoaderFiles directory must be in the same directory as the AutoLoader executable file.

It can also be started from the command line with an optional argument which specifies the script file to process. e.g.

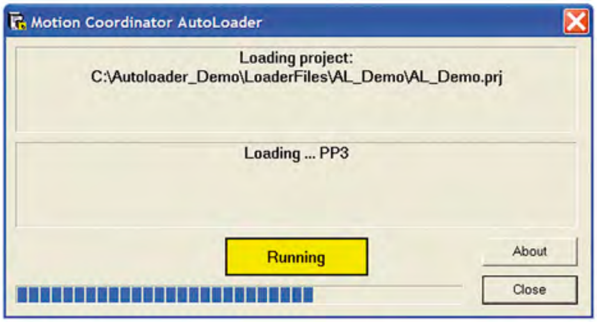
```
AutoLoader E:\MXUpdate\20051203\UpDate1.tas
```

START DIALOG



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



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:

AUTORUN
CHECKPROJECT
CHECKTYPE
CHECKUNLOCKED

CHECKVERSION
COMMLINK (alternative **COMMPORT**)
COMPILEALL
COMPILEPROGRAM
DELETEALL (alternative **NEWALL**)
DELETEPROGRAM
DELTABLE
EPROM
FASTLOADPROGRAM
FASTLOADPROJECT
HALTPROGRAMS
LOADPROGRAM
LOADPROJECT
LOADTABLE
SETDECRYPTIONKEY
SETPROJECT
SETRUNFROMEPROM
TIMEOUT

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.

AUTORUN

PURPOSE:
To run the programs on the controller which are set to run automatically at power-on.

SYNTAX:
AUTORUN

CHECKPROJECT

PURPOSE:
To check the programs on a controller against a project on disk.

SYNTAX:
CHECKPROJECT [<ProjectName>]

Where <ProjectName> is the optional path of the project directory. If the project directory is in the same directory as the ALoader.exe executable then it is just the name of the of the project directory. If no <ProjectName> is specified then the current project, set by a previous **SETPROJECT** or **LOADPROJECT** command, is used. This operation is automatically performed by a **LOADPROJECT** operation.

EXAMPLES:

```
CHECKPROJECT
```

```
CHECKPROJECT TestProj
```

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
MC2	2
MC202	202
MC204	204
Euro205	205
Euro205x	255
MC206	206
PCI208	208
MC216	216
MC224	224
MC402 (Omron)	250
MC402e (Omron)	251
MCW151 (Omron)	260
TJ1-MC16 (Omron)	262
MC302L	292
Euro205XL	254
MC206X	207

MC302X	293
TJ1_MC04 (Omron)	263
MTX205	294
MC464	464
MC209	209
Euro209	259
CJ1_MCH72	264
TJ2_MC64 (Omron)	266
PCI214	214
TJ2_MC04	267
TJ2_MC16	268
MC405	405
MC403	403
MC400	400
P157	305

The ID numbers are used in the **CHECKTYPE** command.

CHECKUNLOCKED

PURPOSE:
To check that the controller is not locked.

SYNTAX:
CHECKUNLOCKED

CHECKVERSION

PURPOSE:
To check the version of the controller system code.

SYNTAX:
```
CHECKVERSION <Operator><Version>
CHECKVERSION <LowVersion>-<HighVersion>
```

EXAMPLES:
```
    CHECKVERSION > 1.49
    CHECKVERSION >= 1.51
```

```
CHECKVERSION 1.42-1.50
```

‘ Comment

PURPOSE:
To allow the user to put descriptive comments into a script.

SYNTAX:
‘ <Text>
Where <Text> is any text.

EXAMPLES:

```
‘ This is a comment line
```

COMMLINK (alternative COMMPORT)

PURPOSE:
To set the communications port and parameters.

SYNTAX:
COMMLINK <PortSpec>
Where <PortSpec> is a string specifying a communications port and the connection parameters.

SERIAL
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.

USB
For a USB connection the string is **USB:0 as only a single USB connection (0) is supported.**

Ethernet

For an Ethernet connection the string is similar to Ethernet:192.168.0.123:23 which specifies an Ethernet connection to IP address 192.168.0.123 on port 23. The final ‘:’ and the port number can be omitted, in which case the port number defaults to 23.

PCI
For a PCI connection the string is similar to **PCI:0 which specifies a connection to PCI card 0.**

EXAMPLES:

```
COMMLINK COM2:9600,7,e,2
COMMLINK USB:0
COMMLINK Ethernet:192.168.0.111
COMMLINK PCI:1
```

COMPILEALL

PURPOSE:
To compile all the programs on the controller.

SYNTAX:
`COMPILEALL`

COMPILEPROGRAM

PURPOSE:
To compile a program on the controller.

SYNTAX:
`COMPILEPROGRAM <Program>`

Where <Program> is the program name.

EXAMPLES:

```
COMPILEPROGRAM Prog
```

The **LOADPROGRAM** command automatically compiles programs after they are loaded so under normal circumstances there is no need to use this command.

DELETEALL (alternative NEWALL)

PURPOSE:
To delete all programs on the controller.

SYNTAX:
`DELETEALL`

DELETEPROGRAM

PURPOSE:
To delete a program on the controller.

SYNTAX:
`DELETEPROGRAM <ProgramName>`
Where <ProgramName> is the name of a program on the controller.

EXAMPLES:
```
DELETEPROGRAM Prog.bas
```

DELETEPROGRAM may fail if programs are running. It will also indicate an error if the specified program is not present on the controller.

DELTABLE

PURPOSE:
To delete the table on the controller.

SYNTAX:
`DELTABLE`
This command should always be used before the **LOADTABLE** command.

This command has no effect on controllers with statically allocated table memory.

EPROM

PURPOSE:
To store the project currently in controller RAM into EPROM

SYNTAX:
`EPROM`

FASTLOADPROGRAM

PURPOSE:
To load a program not in a project onto the controller using the fast method.

SYNTAX:
FASTLOADPROGRAM <ProgramFile>

Where <ProgramFile> is the path of the program file. If the program file is in the same directory as the AutoLoader.exe executable then this is just the file name of the program file.

EXAMPLES:

```
FASTLOADPROGRAM Prog.bas
```

FASTLOADPROGRAM will only work on series 2 Motion Coordinators with system version 1.6653 or later and series 4 Motion Coordinators with system version 2.0010 or later.

FASTLOADPROJECT

PURPOSE:
To load a project from disk onto the controller.

DESCRIPTION:
FASTLOADPROJECT is a faster alternative to **LOADPROJECT**. It is only compatible with system software version 1.63 or later for series 2 *Motion Coordinator*s, and version 1.9013 or later for series 3 *Motion Coordinator*s.

FASTLOADPROJECT must be used if a project contains encrypted programs.

SYNTAX:
FASTLOADPROJECT [<ProjectName>]

Where <ProjectName> is the optional path of the project directory. If the project directory is in the same directory as the ALoader.exe executable then it is just the name of the of the project directory. If no <ProjectName> is specified then the current project, set by a previous **SETPROJECT** command, is used.

EXAMPLES:

```
FASTLOADPROJECT
FASTLOADPROJECT TestProj
```

If **FASTLOADPROJECT** fails and the project only contains Trio BASIC source files then using

LOADPROJECT may work

HALTPROGRAMS

PURPOSE:
To halt all programs on the controller.

SYNTAX:
HALTPROGRAMS

This operation is automatically performed as part of **LOADPROJECT**, **LOADPROGRAM** and **DELTABLE** commands.

LOADPROGRAM

PURPOSE:
To load a program not in a project onto the controller.

SYNTAX:
LOADPROGRAM <ProgramFile>

Where <ProgramFile> is the path of the program file. If the program file is in the same directory as the ALoader.exe executable then this is just the file name of the program file.

EXAMPLES:

```
LOADPROGRAM Prog.bas
```

LOADPROGRAM will only load TrioBASIC source files.

LOADPROJECT

PURPOSE:
To load a project from disk onto the controller.

SYNTAX:
LOADPROJECT [<ProjectName>]

Where <ProjectName> is the optional path of the project directory. If the project directory is in the same directory as the ALoader.exe executable then it is just the name of the of the project directory. If no <ProjectName> is specified then the current project, set by a previous **SETPROJECT** command, is used.

EXAMPLES:

```
LOADPROJECT
LOADPROJECT TestProj
```

LOADPROJECT will only load projects which only contain Trio **BASIC** source files. If a project contains other types of file (i.e. encrypted programs) then **FASTLOADPROJECT** must be used

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 LoaderFiles 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
```

SETDECRYPTIONKEY

PURPOSE:

To set the decryption key required when load an encrypted project from disk onto the controller.

DESCRIPTION:

SETDECRYPTIONKEY sets the decryption key for a subsequent **FASTLOADPROJECT** operation. The decryption key is only used when a project containing one or more encrypted programs is loaded onto a controller using **FASTLOADPROJECT**.

If a project contains encrypted programs, it can only be loaded using **FASTLOADPROJECT**.

SYNTAX:
SETDECRYPTIONKEY KeyString

EXAMPLES:
```
SETDECRYPTIONKEY 67dj0.fIcc
```

Decryption keys are a derived from the key string used to encrypt the program(s) and the security code of the target controller. Decryption keys can be generated using the Project Encryptor tool distributed with Motion Perfect.

SETPROJECT

PURPOSE:
To set the current project for following commands.

SYNTAX:
SETPROJECT <ProjectName>

Where <ProjectName> is the path of the project directory. If the project directory is in the same directory as the ALoader.exe executable then it is just the name of the of the project directory.

EXAMPLES:
```
SETPROJECT TestProj
```

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 @
```

This command only applies to controllers which have battery backed RAM (controllers with no battery backed RAM will always copy programs from EPROM).

TIMEOUT

PURPOSE:
To set the command timeout.

SYNTAX:
`TIMEOUT` *`time`*

Where time is the timeout value in seconds (default is 10).

EXAMPLE:

```
TIMEOUT 30
```

It will normally only be necessary to increase the timeout above 10 if there are large programs in the target controller or you are loading large programs onto it.

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 execution 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.

Trio MC Loader

INTRODUCTION

Trio MC Loader is a Windows ActiveX control which can load projects (produced with *Motion* Perfect) and programs onto a Trio *Motion Coordinator*. Communication with the *Motion Coordinator* can be via Serial link, USB, Ethernet or PCI depending on the *Motion Coordinator*.

PROPERTIES

The control has the following properties:

CommLink
ControllerSystemVersion
ControllerType
DecryptionKey
DisplayGaugeDuringProgramLoad
Locked
Open
ProjectFile
RunFromEPROM
Timeout

EVENTS

The control does not generate any events.

Property: CommLink

TYPE:

BSTR (string)

ACCESS:

Read / write

DESCRIPTION:

This property is used to get or set the configuration of the communications link. The format of the string depends on the type of communications link being used.

SERIAL

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 most controllers.

USB

For a USB connection the string is USB:0 as only a single USB connection (0) is supported.

ETHERNET
For an Ethernet connection the string is similar to Ethernet:192.168.0.123:23 which specifies an Ethernet connection to IP address 192.168.0.123 on port 23. The final ':' and the port number can be omitted, in which case the port number defaults to 23.

PCI
For a PCI connection the string is similar to **PCI:0 which specifies a connection to PCI card 0.**

EXAMPLES:

VISUAL BASIC:
```
axLoader.CommLink = "Ethernet:192.168.22.11"
```

VISUAL C#:
```
axLoader.CommLink = "Ethernet:192.168.22.11";
```

Property: ControllerSystemVersion

TYPE:
double

ACCESS:
Read

DESCRIPTION:
This is a read-only property which returns the controller system software version number.

EXAMPLES:

VISUAL BASIC:
```
Dim Version As Double

Version = axLoader.ControllerSystemVersion
```

VISUAL C#:
```
double dVersion;

dVersion = axLoader.ControllerSystemVersion;
```

Property: ControllerType

TYPE:
`unsigned long`

ACCESS:
Read

DESCRIPTION:
This is a read-only property which returns the Controller Type code.

EXAMPLES:

VISUAL BASIC:
```
    Dim ConType As Long

    ConType = axLoader.ControllerType
```

VISUAL C#:
```
    ulong ulConType;

    ulConType = axLoader.ControllerType;
```

Property: DecryptionKey

TYPE:
`BSTR  (string)`

ACCESS:
Read / write

DESCRIPTION:
The **`DecryptionKey`** property sets/gets the decryption key for a subsequent fast mode **`LoadProject`** operations. The decryption key is only used when a project containing one or more encrypted programs is loaded onto a controller using fast LoadProject.

EXAMPLES:

VISUAL BASIC:
```
    axLoader.DecryptionKey = “hjiHU87OOo”
```

VISUAL C#:
```
    axLoader.DecryptionKey = “hjiHU87OOo”;
```

Decryption keys are a derived from the key string used to encrypt the program(s) and the security code of the target controller. Decryption keys can be generated using the Project Encryptor tool distributed with *Motion* Perfect.

Property: DisplayGaugeDuringProgramLoad

TYPE:
`VARIANT_BOOL`

ACCESS:
Read / write

DESCRIPTION:
This property is used to control the display of a gauge (progress control) whilst a program is loading. When true, a gauge is displayed showing progress as a program is loaded. When false no gauge is displayed.

Displaying a gauge whilst a program is loaded gives some feedback to the user that something is happening. Otherwise there would potentially be a long period where nothing happens, which may give the impression that the program has hung up.

EXAMPLES:

VISUAL BASIC:
```
    If Not axLoader.DisplayGaugeDuringProgramLoad Then
        axLoader.DisplayGaugeDuringProgramLoad = True
```

VISUAL C#:
```
    if (!axLoader.DisplayGaugeDuringProgramLoad)
        axLoader.DisplayGaugeDuringProgramLoad = true;
```

Property: Locked

TYPE:
`VARIANT_BOOL`

ACCESS:
Read

DESCRIPTION:
This is a read-only property which returns the locked state of the controller (**true for locked, false for unlocked).**

EXAMPLES:

VISUAL BASIC:
```
Dim IsLocked As Boolean

IsLocked = axLoader.Locked
```

VISUAL C#:
```
bool bLocked;

bLocked = axLoader.Locked;
```

Property: Open

TYPE:
`bool`

ACCESS:
Read / write

DESCRIPTION:
The Open property sets/gets the state of the communications port used to communicate with the controller.

EXAMPLES:

VISUAL BASIC:
```
If Not axLoader.Open Then
  axLoader.Open = False
End If
```

VISUAL C#:
```
if (!axLoader.Open)
    axLoader.Open = false;
```

Any method or property which needs to communicate with the controller will automatically open a communications port if the parameters have been set. The communications port is not closed on completion of a command so the primary use of this property is to close the communications link rather than to open it.

Property: ProjectFile

TYPE:
`BSTR  (string)`

ACCESS:
Read / write

DESCRIPTION:
This property is used to get or set the current project file.   The full path to the project file should be used when setting this property.

EXAMPLES:

VISUAL BASIC:
```
    If axLoader.ProjectFile.Length = 0 then
        axLoader.ProjectFile = “C:\Projects\PPX\PPX.prj”
    End If
```

VISUAL C#:
```
    if (axLoader.ProjectFile.Length == 0)
        axLoader.ProjectFile = “C:\\Projects\\PPX\\PPX.prj”;
```

Property: RunFromEPROM

TYPE:
`VARIANT_BOOL`

ACCESS:
Read / write

DESCRIPTION:
This property is used to control how the controller starts up.   When set to **false it uses programs stored in its RAM memory. When set to true the controller uses programs stored in its EPROM memory (overwriting the programs in RAM).**

EXAMPLES:

VISUAL BASIC:
```
    If not axLoader.RunFromEPROM then
```

```
        axLoader.RunFromEPROM = True
    End If
```

VISUAL C#:

```
    if (!axLoader.RunFromEPROM)
        axLoader.RunFromEPROM = true;
```

Property: Timeout

TYPE:

`unsigned long`

ACCESS:

Read / write

DESCRIPTION:

This property is used to set the command timeout for communications with the controller. The default value is 10 (seconds) but may need to be increased if you are using large programs or have a large project.

EXAMPLES:

VISUAL BASIC:

```
    If axLoader.Timeout < 20 Then
        axLoader.Timeout = 25
    End If
```

VISUAL C#:

IF (AXLOADER.TIMEOUT < 20)

```
        axLoader.Timeout = 25;
```

Methods

The control has the following methods:

```
AutoRun
CheckProject
ClearGaugePosition
CompileAll
CompileProgram
DeleteAll
DeleteProgram
DeleteTable
FastLoadProgram
GetLastError
GetLastErrorString
HaltPrograms
LoadProgram
LoadProject
LoadTable
Lock
SetGaugePosition
StoreInEPROM
Unlock
```

Method: AutoRun

PARAMETERS:
none

RETURN TYPE:
VARIANT_BOOL

DESCRIPTION:
This method is used to run any programs on the controller which are set to auto-run on startup.

The return value is true if the method call succeeded and false if it failed. Further error information can be obtained by calling the **GetLastError** and **GetLastError**String methods.

EXAMPLES:

VISUAL BASIC:

```
If Not axLoader.AutoRun Then
    DisplayError(axLoader.GetLastError, axLoader.GetLastErrorString)
End If
```

VISUAL C#:

```
if (!axLoader.AutoRun())
```

```
        DisplayError(axLoader.GetLastError, axLoader.GetLastErrorString);
```

Method: CheckProject

PARAMETERS:
none

RETURN TYPE:
VARIANT_BOOL

DESCRIPTION:
This method is used to check the programs on the controller against the project previously set using the ProjectFile.

The return value is true if the method call succeeded and false if it failed. Further error information can be obtained by calling the **GetLastError** and **GetLastError**String methods.

EXAMPLES:

VISUAL BASIC:
```
    If Not axLoader.CheckProject Then
        DisplayError(axLoader.GetLastError, axLoader.GetLastErrorString)
    End If
```

VISUAL C#:
```
    if (!axLoader.CheckProject())
        DisplayError(axLoader.GetLastError, axLoader.GetLastErrorString);
```

Method: ClearGaugePosition

PARAMETERS:
None.

RETURN TYPE:
VOID

DESCRIPTION:
This method is used to clear the position of the gauge dialog which is displayed while a program is being loaded, which has been previously set using the SetGaugePosition method. This causes the gauge dialog to be displayed in its default position (the centre of the screen).

EXAMPLES:

VISUAL BASIC:
```
    ClearGaugePosition
```

VISUAL C#:
```
    ClearGaugePosition();
```

Method: CompileAll

PARAMETERS:
none

RETURN TYPE:
VARIANT_BOOL

DESCRIPTION:
This method is used to compile all programs on the controller.

The return value is true if the method call succeeded and false if it failed. Further error information can be obtained by calling the **GetLastError** and **GetLastError**String methods.

EXAMPLES:

VISUAL BASIC:
```
    If Not axLoader.CompileAll Then
        DisplayError(axLoader.GetLastError, axLoader.GetLastErrorString)
    End If
```

VISUAL C#:
```
    if (!axLoader.CompileAll())
        DisplayError(axLoader.GetLastError, axLoader.GetLastErrorString);
```

Method: CompileProgram

PARAMETERS:
BSTR (string): ProgramName

RETURN TYPE:
VARIANT_BOOL

DESCRIPTION:
This method is used to compile a single program on the controller.

The return value is true if the method call succeeded and false if it failed. Further error information can be obtained by calling the **GetLastError** and **GetLastError**String methods.

EXAMPLES:

VISUAL BASIC:
```
    If Not axLoader.CompileProgram(“PROG”) Then
        DisplayError(axLoader.GetLastError, axLoader.GetLastErrorString)
    End If
```

VISUAL C#:
```
    if (!axLoader.CompileProgram(“PROG”))
        DisplayError(axLoader.GetLastError, axLoader.GetLastErrorString);
```

Method: DeleteAll

PARAMETERS:
none

RETURN TYPE:
VARIANT_BOOL

DESCRIPTION:
This method is used to delete the all the programs on the controller.

The return value is true if the method call succeeded and false if it failed. Further error information can be obtained by calling the **GetLastError** and **GetLastError**String methods.

EXAMPLES:

VISUAL BASIC:
```
    If Not axLoader.DeleteAll Then
        DisplayError(axLoader.GetLastError, axLoader.GetLastErrorString)
    End If
```

VISUAL C#:
```
    if (!axLoader.DeleteAll())
```

```
        DisplayError(axLoader.GetLastError, axLoader.GetLastErrorString);
```

Method: DeleteProgram

PARAMETERS:
`BSTR (string): ProgramName`

RETURN TYPE:
`VARIANT_BOOL`

DESCRIPTION:
This method is used to delete a single program from the controller.

The return value is true if the method call succeeded and false if it failed. Further error information can be obtained by calling the **`GetLastError`** and **`GetLastErrorString`** methods.

EXAMPLES:

VISUAL BASIC:
```
    If Not axLoader.DeleteProgram("PROG") Then
        DisplayError(axLoader.GetLastError, axLoader.GetLastErrorString)
    End If
```

VISUAL C#:
```
    if (!axLoader.DeleteProgram("PROG"))
        DisplayError(axLoader.GetLastError, axLoader.GetLastErrorString);
```

Method: DeleteTable

PARAMETERS:
none

RETURN TYPE:
`VARIANT_BOOL`

DESCRIPTION:
This method is used to delete the table on the controller. It only works on controllers which do not have dedicated table memory.

The return value is true if the method call succeeded and false if it failed. Further error information can be obtained by calling the **GetLastError** and **GetLastErrorString** methods.

EXAMPLES:

VISUAL BASIC:

```
If Not axLoader.DeleteTable Then
    DisplayError(axLoader.GetLastError, axLoader.GetLastErrorString)
End If
```

VISUAL C#:

```
if (!axLoader.DeleteTable())
    DisplayError(axLoader.GetLastError, axLoader.GetLastErrorString);
```

Method: FastLoadProgram

PARAMETERS:

BSTR (string): ProgramFileName
VARIANT_BOOL: Compile

RETURN TYPE:

VARIANT_BOOL

DESCRIPTION:

This method is used to load a single program onto the controller using the fast load method. If Compile is true, the program will be compiled after it has been loaded (it is generally good practice to do this).

The return value is true if the method call succeeded and false if it failed. Further error information can be obtained by calling the **GetLastError** and **GetLastErrorString** methods.

EXAMPLES:

VISUAL BASIC:

```
If Not axLoader.FastLoadProgram("C:\Programs\Prog.bas", True) Then
    DisplayError(axLoader.GetLastError, axLoader.GetLastErrorString)
End If
```

VISUAL C#:

```
if (!axLoader.FastLoadProgram("C:\\Programs\\Prog.bas", true))
    DisplayError(axLoader.GetLastError, axLoader.GetLastErrorString);
```

FASTLOADPROGRAM will only work on series 2 Motion Coordinators with system version 1.6653 or later and series 4 Motion Coordinators with system version 2.0010 or later.

Method: GetLastError

PARAMETERS:
none

RETURN TYPE:
`unsigned long`

DESCRIPTION:
This method is used to retrieve the error code after a method call has failed (returned false). The returned error code is only valid for the previous method call.

The following error codes can be returned:

Code	Error Description
0	No error
1	File does not exist
2	Error opening file
3	Invalid IP address
4	Invalid IP port
5	Invalid integer
6	Invalid communications port
7	Invalid communications parameters
8	Communications error
9	Communications echo error
10	Invalid controller system version
11	Invalid controller type
12	Controller type not found
13	Invalid range
14	Failed version check
15	Controller locked
16	Failed to set project
17	Invalid command
18	Directory does not exist
19	No file specified
20	Program not in project
21	Program not on controller
22	CRC mismatch
23	Invalid directory
24	Failed to create directory
25	Invalid program file name
26	Error writing to file
27	Error reading CRC
28	Error calculating CRC
29	File not in project
30	Invalid program name
31	Failed to halt programs
32	Error reading directory
33	Program failed to compile

Code	Error Description
34	Failed to set communications parameters
35	Failed to get communications parameters
36	Transmit failure
37	Invalid connection type
38	Internal pointer error
39	Error sending string
40	Error sending command
41	Failed to select program
42	Program not loadable
43	Program does not exist
44	Project failed to load
45	Program failed to load
46	Program not compilable
47	Error deleting program
48	Error opening communications port
49	Error locking controller
50	Error unlocking controller

Further error information can be obtained by calling the **GetLastError**String method.

EXAMPLES:

VISUAL BASIC:
```
    If Not axLoader.CompileAll Then
        DisplayError(axLoader.GetLastError, axLoader.GetLastErrorString)
    End If
```

VISUAL C#:
```
    if (!axLoader.CompileAll())
        DisplayError(axLoader.GetLastError, axLoader.GetLastErrorString);
```

Method: GetLastErrorString

PARAMETERS:
none

RETURN TYPE:
BSTR (string)

DESCRIPTION:
This method is used to retrieve additional information from the controller. The string contains extra information which can be used in conjunction with the error code returned by the **GetLastError** method.

EXAMPLES:

VISUAL BASIC:
```
If Not axLoader.CompileAll Then
    DisplayError(axLoader.GetLastError, axLoader.GetLastErrorString)
End If
```

VISUAL C#:
```
if (!axLoader.CompileAll())
    DisplayError(axLoader.GetLastError, axLoader.GetLastErrorString);
```

Method: HaltPrograms

PARAMETERS:
none

RETURN TYPE:
VARIANT_BOOL

DESCRIPTION:
This method is used to halt all programs currently running on the controller.

The return value is true if the method call succeeded and false if it failed. Further error information can be obtained by calling the **GetLastError** and **GetLastError**String methods.

EXAMPLES:

VISUAL BASIC:
```
If Not axLoader.HaltPrograms Then
    DisplayError(axLoader.GetLastError, axLoader.GetLastErrorString)
End If
```

VISUAL C#:
```
if (!axLoader.HaltPrograms())
    DisplayError(axLoader.GetLastError, axLoader.GetLastErrorString);
```

Method: LoadProgram

PARAMETERS:
BSTR (string): ProgramFileName
VARIANT_BOOL: Compile

RETURN TYPE:
VARIANT_BOOL

DESCRIPTION:
This method is used to load a single program onto the controller. If Compile is true, the program will be compiled after it has been loaded (it is generally good practice to do this).

The return value is true if the method call succeeded and false if it failed. Further error information can be obtained by calling the **GetLastError** and **GetLastError**String methods.

EXAMPLES:

VISUAL BASIC:
```
    If Not axLoader.LoadProgram("C:\Programs\Prog.bas", True) Then
        DisplayError(axLoader.GetLastError, axLoader.GetLastErrorString)
    End If
```

VISUAL C#:
```
    if (!axLoader.LoadProgram("C:\\Programs\\Prog.bas", true))
        DisplayError(axLoader.GetLastError, axLoader.GetLastErrorString);
```

Method: LoadProject

PARAMETERS:
VARIANT_BOOL: FastLoad

RETURN TYPE:
VARIANT_BOOL

DESCRIPTION:
This method is used to load the project previously set using the ProjectFile property onto the controller. If FastLoad is true, the loader will use the fast loading algorithm. Fast loading is not available some controllers and is only available in more recent versions of system software. All controllers will perform a normal (slow) load. Fast load must be used if the project contains one or more encrypted programs.

The return value is true if the method call succeeded and false if it failed. Further error information can be obtained by calling the **GetLastError** and **GetLastError**String methods.

EXAMPLES:

VISUAL BASIC:
```
    If Not axLoader.LoadProject(False) Then
        DisplayError(axLoader.GetLastError, axLoader.GetLastErrorString)
```

```
    End If
```

VISUAL C#:

```
    if (!axLoader.LoadProject(false))
        DisplayError(axLoader.GetLastError, axLoader.GetLastErrorString);
```

Method: LoadTable

PARAMETERS:

`BSTR (string):  TableFileName`

RETURN TYPE:

`VARIANT_BOOL`

DESCRIPTION:

This method is used to load data into the table on the controller from a table list file (usually saved by *Motion* Perfect).

The return value is true if the method call succeeded and false if it failed. Further error information can be obtained by calling the GetLastError and GetLastErrorString methods.

EXAMPLES:

VISUAL BASIC:

```
    If Not axLoader.LoadTable("C:\Tables\ThisTable.lst") Then
        DisplayError(axLoader.GetLastError, axLoader.GetLastErrorString)
    End If
```

VISUAL C#:

```
    if (!axLoader.LoadTable("C:\\Tables\\ThisTable.lst"))
        DisplayError(axLoader.GetLastError, axLoader.GetLastErrorString);
```

Method: Lock

PARAMETERS:

`unsigned long: Lock Code`

RETURN TYPE:

`VARIANT_BOOL`

DESCRIPTION:

This method is used to lock the controller so that programs cannot be edited. The lock code used here must also be used if the controller is unlocked using the **Unlock** method.

The return value is true if the method call succeeded and false if it failed. Further error information can be obtained by calling the **GetLastError** and **GetLastError**String methods.

EXAMPLES:

VISUAL BASIC:

```
If Not axLoader.Lock(1234) Then
    DisplayError(axLoader.GetLastError, axLoader.GetLastErrorString)
End If
```

VISUAL C#:

```
if (!axLoader.Lock(1234))
    DisplayError(axLoader.GetLastError, axLoader.GetLastErrorString);
```

Method: SetGaugePosition

PARAMETERS:

LONG: x
LONG: y

RETURN TYPE:

VOID

DESCRIPTION:

This method is used to position the gauge dialog which is displayed while a program is being loaded. The parameters x and y are the screen coordinates of the top, left corner of the gauge dialog.

The gauge display position can be reset to default using the **ClearGaugePosition** method.

EXAMPLES:

VISUAL BASIC:

```
SetGaugePosition(10, 20)
```

VISUAL C#:

```
SetGaugePosition(10, 20);
```

Method: StoreInEPROM

PARAMETERS:
None

RETURN TYPE:
`VARIANT_BOOL`

DESCRIPTION:
This method is used to store the programs already loaded onto the controller into the controller's EPROM memory.

The return value is true if the method call succeeded and false if it failed. Further error information can be obtained by calling the **GetLastError** and **GetLastError**String methods.

EXAMPLES:

VISUAL BASIC:
```
    If Not axLoader.StoreInEPROM Then
        DisplayError(axLoader.GetLastError, axLoader.GetLastErrorString)
    End If
```

VISUAL C#:
```
    if (!axLoader.StoreInEPROM())
        DisplayError(axLoader.GetLastError, axLoader.GetLastErrorString);
```

Method: Unock

PARAMETERS:
`unsigned long:   LockCode`

RETURN TYPE:
`VARIANT_BOOL`

DESCRIPTION:
This method is used to unlock a locked controller so that programs can be edited. The lock code used here must be the same as the code used to lock the controller.

The return value is true if the method call succeeded and false if it failed. Further error information can be obtained by calling the **GetLastError** and **GetLastError**String methods.

EXAMPLES:

VISUAL BASIC:

```
If Not axLoader.Unlock(1234) Then
    DisplayError(axLoader.GetLastError, axLoader.GetLastErrorString)
End If
```

VISUAL C#:

```
if (!axLoader.Unlock(1234))
    DisplayError(axLoader.GetLastError, axLoader.GetLastErrorString);
```

INDEX

Index

N

S