

**Motion
Programming
using
MotionPRO
Developer**



ElectroCraft Document Number
A11229 Rev 1

User Manual

ELECTROCRAFT

**Motion Programming
using
MotionPRO
User Manual**

ElectroCraft Document No.

A11229

ElectroCraft
4480 Varsity Drive
Suite G
Ann Arbor, MI 48108

Tel.: (734) 662-7771

www.electrocraft.com

Read This First

Whilst ElectroCraft believes that the information and guidance given in this manual is correct, all parties must rely upon their own skill and judgment when making use of it. ElectroCraft does not assume any liability to anyone for any loss or damage caused by any error or omission in the work, whether such error or omission is the result of negligence or any other cause. Any and all such liability is disclaimed.

All rights reserved. No part or parts of this document may be reproduced or transmitted in any form or by any means, electrical or mechanical including photocopying, recording or by any information-retrieval system without permission in writing from ElectroCraft.

The information in this document is subject to change without notice.

About This Manual

This book is a technical reference manual for the MotionPRO Developer software.

In order to operate any ElectroCraft drive, you need to pass through 3 steps:

- ❑ **Step 1 Hardware installation**
- ❑ **Step 2 Drive setup** using ElectroCraft **PROconfig** or **MotionPRO Developer** software for drive commissioning
- ❑ **Step 3 Motion Programming** using one of the options:
 - ❑ A **CANOpen** or **EtherCAT** master
 - ❑ The drive **built-in motion controller** executing a **ElectroCraft Motion Program Language (MPL)** program developed using ElectroCraft **MotionPRO Developer** software
 - ❑ A **MPL_LIB motion library for PCs** (Windows or Linux)
 - ❑ A **MPL_LIB motion library for PLCs**
 - ❑ A **distributed control** approach which combines the above options, like for example a host calling motion functions programmed on the drives in MPL

This manual covers **Steps 2 and 3** in detail. For detailed information regarding the first step, refer to the specific documentation of each drive.

Notational Conventions

This document uses the following conventions:

MPL – ElectroCraft Motion Program Language

Faxx – Firmware versions with A = 0, 1, 2, 3, 4 or 9; Examples: F005K, F120B, F900H

FBxx – Firmware versions with B = 5, 6, 7, 8; Examples: F500B, F600C, F800I

Programmable drive – a drive with an embedded motion controller capable to execute high-level motion language programs

Programmable motor – a motor with an embedded programmable drive

SI units – International standard units (meter for length, seconds for time, etc.)

IU units – Internal units of the drive

Related Documentation

Help of the PROconfig software – describes how to use **PROconfig** to quickly setup any ElectroCraft drive for your application using only 2 dialogues. The output of PROconfig is a set of setup data that can be downloaded into the drive EEPROM or saved on a PC file. At power-on, the drive is initialized with the setup data read from its EEPROM. With PROconfig it is also possible to retrieve the complete setup information from a drive previously programmed. PROconfig includes a firmware programmer with allows you to update your drive firmware to the latest revision. PROconfig is part of the ElectroCraft Motion PRO Suite. Motion PRO Suite is available as part of a PRO Series Drive Evaluation Kit. Please contact ElectroCraft or your local ElectroCraft sales representative for more information on obtaining MotionPRO Suite or an evaluation kit.

CANopen Programming (Document No. A11226) – explains how to program the ElectroCraft programmable drives using **CANopen** protocol. Describes the associated **DS-301** communication profile and **CiA-402** device profile.

CANopen over EtherCAT Programming (Document No. A11227) – explains how to program the ElectroCraft Programmable drives with EtherCAT interface using **CANopen over EtherCAT** protocol. Presents the **CiA-402** associated drive profile and object dictionary.

Help of the MotionPRO Developer software – describes how to use MotionPRO Developer to create motion programs using ElectroCraft Motion Program Language (MPL). MotionPRO Developer platform includes **PROconfig** for the drive/motor setup, and a Motion Editor for the motion Programming. The Motion Editor provides a simple way of creating motion programs and automatically generates all the MPL instructions. *With MotionPRO Developer you can fully benefit from a key advantage of ElectroCraft drives – their capability to execute complex motions without requiring an external motion controller, thanks to their built-in motion controller.*

PRO Series and LIB v2.0 (Document Number A11230) – explains how to program in **C, C++,C#, Visual Basic or Delphi Pascal** a motion application for the ElectroCraft Programmable drives using ElectroCraft motion control library for PCs. The MPL_lib includes ready-to-run examples that can be executed on **Windows** or **Linux** (x86 and x64).

PRO Series and LabVIEW v2.0 (Document No. A11231) – explains how to program in **LabVIEW** a motion application for the ElectroCraft Programmable drives using MPL_LIB_Labview v2.0 motion control library for PCs. The MPL_Lib_LabVIEW includes over 40 ready-to-run examples.

PRO Series and LIB_S7 (Document No. A11232) – explains how to program, in a PLC **Siemens series S7-300 or S7-400**, a motion application for the ElectroCraft Programmable drives using MPL_LIB_S7 motion control library. The MPL_LIB_S7 library is **IEC61131-3 compatible**.

PRO Series and CJ1 (Document No. A11233) – explains how to program, in a PLC Omron series CJ1, a motion application for the ElectroCraft Programmable drives using MPL_LIB_CJ1 motion control library. The MPL_LIB_CJ1 library is **IEC61131-3 compatible**.

PRO Series and X20 (Document No. A11234) – explains how to program, in a PLC B&R series X20, a motion application for the ElectroCraft Programmable drives using MPL_LIB_X20 motion control library. The MPL_LIB_X20 library is **IEC61131-3 compatible**.

ElectroCAN (Document No. A11235) – presents ElectroCAN protocol – an extension of the CANopen communication profile used for MPL commands

If you Need Assistance ...

If you want to ...	Contact ElectroCraft at ...
Visit ElectroCraft online	World Wide Web: http://www.electrocrafter.com/
Receive general information or assistance (see Note)	World Wide Web: http://www.electrocrafter.com/ Email: drivesupport@electrocrafter.com
Ask questions about product operation or report suspected problems (see Note)	Fax: (41) 32 732 55 04 Email: : drivesupport@electrocrafter.com
Make suggestions about, or report errors in documentation.	Mail: ElectroCraft 4480 Varsity Drive Suite G Ann Arbor, MI 48108 Tel.: (734) 662-7771

This page is empty

Contents

Read This First	I
1. Overview	1
1.1. Getting Started with MotionPRO Developer	1
2. Project Management	9
2.1. Project File Concept	9
2.2. Memory Setting	12
2.3. Application - Setup	14
2.4. Application - Motion	16
2.4.1. Homing Modes	19
2.4.2. Homing Modes Edit	21
2.4.3. Functions	22
2.4.4. Functions Edit	23
2.4.5. Interrupts	24
2.4.6. Interrupts Edit	25
2.4.7. CAM Tables	26
2.4.8. CAM Tables Edit	28
3. MotionPRO Developer Workspace	30
3.1. Menu Bar	31
3.1.1. Project Menu	31
3.1.2. Application Menu	32
3.1.3. Application Setup Menu	33
3.1.4. Application Motion Menu	34
3.1.5. Communication Menu	35
3.1.6. View Menu	36
3.1.7. Logger	36
3.1.8. Control Panel	38
3.1.9. Help	39
3.2. Toolbar	40
4. Evaluation Tools	42
4.1. Data Logger	42

4.1.1.	Data Logger.....	42
4.1.2.	Data Logger - Start.....	43
4.1.3.	Data Logger - Plot Options.....	44
4.1.4.	Data Logger - Plot Setup.....	46
4.1.5.	Data Logger - Variables.....	48
4.1.6.	Data Logger - Other Options.....	50
4.2.	Control Panel.....	52
4.2.1.	Control Panel.....	52
4.2.2.	Control Panel - Show Value.....	58
4.2.3.	Control Panel - Scope.....	60
4.2.4.	Control Panel - Double Scope.....	62
4.2.5.	Control Panel - Y(X) Scope Object.....	64
4.2.6.	Control Panel - Gauge.....	66
4.2.7.	Control Panel - Slider.....	68
4.2.8.	Control Panel - Input.....	70
4.2.9.	Control Panel - Bit Value.....	71
4.2.10.	Control Panel - User Defined MPL Sequence Object.....	72
4.2.11.	Control Panel - Label.....	73
4.2.12.	Control Panel - Output.....	74
4.2.13.	Control Panel Properties.....	75
4.3.	Command Interpreter.....	75
4.4.	Binary Code Viewer.....	78
4.5.	Memory View.....	80
5.	Communication.....	81
5.1.	Communication Setup.....	81
5.1.1.	RS-232 Communication Setup.....	83
5.1.2.	RS-232 Communication Troubleshoots.....	85
5.1.3.	CAN-bus Communication Setup.....	86
5.1.4.	CAN-bus Communication Troubleshoots.....	89
5.1.5.	User implemented serial driver example.....	90
5.1.6.	User Implemented Serial Driver Setup.....	94
5.1.7.	User Implemented Serial Driver Troubleshoots.....	97
5.1.8.	Advanced Communication Setup.....	98
5.2.	Communication Protocols.....	101
5.2.1.	Message Structure. Axis ID and Group ID.....	103
5.2.2.	Serial communication. RS-232 and RS-485 protocols.....	106
5.2.3.	CAN-bus communication. ElectroCAN protocol.....	115

5.2.4.	CAN-bus communication. MPLCAN protocol.....	128
6.	Application Programming	134
6.1.	Motion Programming – drives with built-in Motion Controller.....	134
6.1.1.	Motion Programming Toolbars	138
6.1.2.	Motion Trapezoidal Profile.....	143
6.1.3.	Motion S-Curve Profile	146
6.1.4.	Motion PT	148
6.1.5.	Motion PVT.....	151
6.1.6.	Motion External	154
6.1.7.	Motion Electronic Gearing	157
6.1.8.	Motion Electronic Camming	161
6.1.9.	Motor Commands.....	166
6.1.10.	Motion Position Triggers	169
6.1.11.	Motion Homing.....	170
6.1.12.	Motion Contouring.....	172
6.1.13.	Motion Test	176
6.1.14.	Events Dialogue	178
6.1.15.	Jumps and Function Calls.....	192
6.1.16.	I/O General I/O (Firmware FAxx)	194
6.1.17.	I/O General I/O (Firmware FBxx)	197
6.1.18.	Assignment & Data Transfer - Setup 16-bit variable.....	199
6.1.19.	Assignment & Data Transfer - Setup 32-bit variable.....	200
6.1.20.	Assignment & Data Transfer - Arithmetic Operations	203
6.1.21.	Assignment & Data Transfer - Data Transfer Between Axes.....	205
6.1.22.	Send data to host.....	208
6.1.23.	Assignment & Data Transfer - Miscellaneous	210
6.1.24.	MPL Interrupt Settings	213
6.1.25.	Free text.....	218
6.2.	ElectroCraft Motion Language.....	219
6.2.1.	Basic Concepts	219
6.2.2.	MPL Description	239
6.2.3.	Electronic Gearing - MPL Programming Details.....	274
6.2.4.	MPL Instruction set.....	365
6.2.5.	Instructions descriptions.....	382
6.2.6.	MPL Registers.....	654
6.2.7.	Bit 9 SPDLP. Speed loop status.....	667
6.3.	Internal Units and Scaling Factors.....	687
6.4.	PRO EEPROM Programmer	687

6.4.1. PRO EEPROM Programmer	687
6.4.2. PRO EEPROM Programmer File Format.....	691
Appendix A : MPL Instructions List.....	693
7. Appendix B : MPL Data List.....	701

1. Overview

1.1. Getting Started with MotionPRO Developer

MotionPRO Developer is an integrated development environment for the setup and motion programming of ElectroCraft Programmable drives and motors. The output of MotionPRO Developer is a set of setup data and a motion program, which can be downloaded to the drive/motor EEPROM or saved on your PC for later use.

MotionPRO Developer includes a set of evaluation tools like the Data Logger, the Control Panel and the Command Interpreter which help you to quickly develop, test, measure and analyze your motion application.

MotionPRO Developer works with **projects**. A project contains one or several **Applications**.

Each application describes a motion system for one axis. It has 2 components: the **Setup** data and the **Motion** program and an associated axis number: an integer value between 1 and 255. An application may be used either to describe:

1. One axis in a multiple-axis system
2. An alternate configuration (set of parameters) for the same axis.

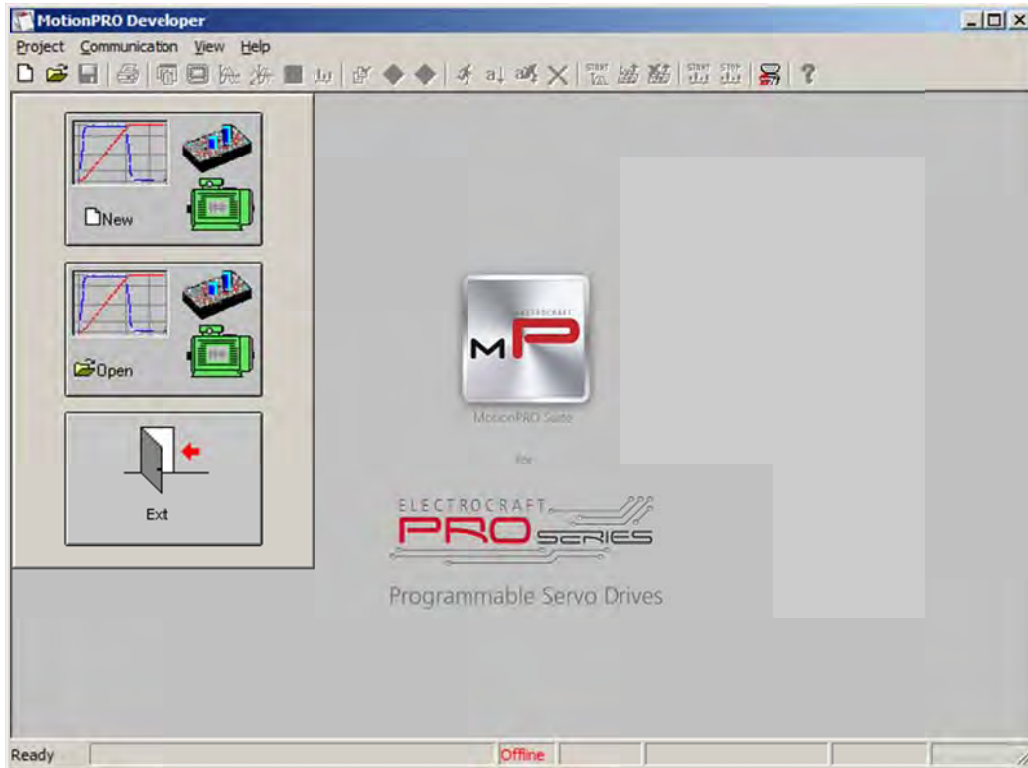
In the first case, each application has a different axis number corresponding to the axis ID of the drives/motors from the network. All data exchanges are done with the drive/motor having the same address as the selected application. In the second case, all the applications have the same axis number.

The setup component contains all the information needed to configure and parameterize a ElectroCraft drive/motor. This information is preserved in the drive/motor EEPROM in the *setup table*. The setup table is copied at power-on into the RAM memory of the drive/motor and is used during runtime.

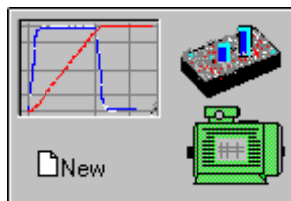
The motion component contains the motion sequences to do. These are described via a MPL (ElectroCraft Motion Program Language) program, which is executed by the drives/motors built-in motion controller.

Step 1 Create a new project

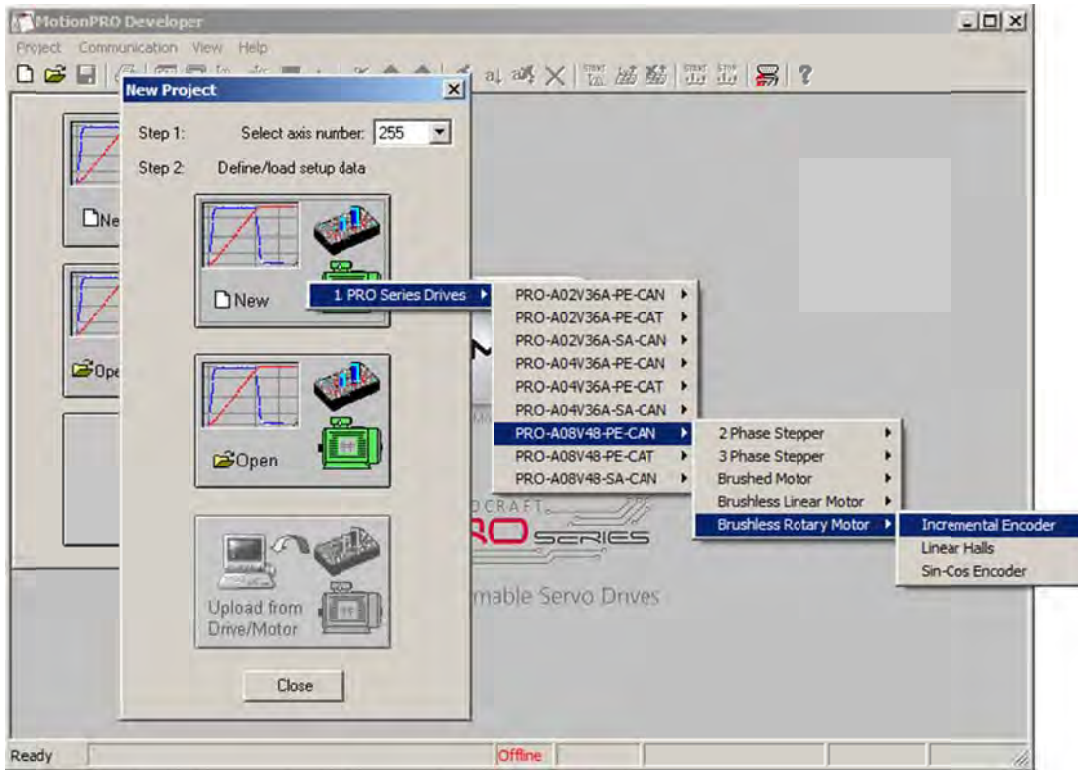
MotionPRO Developer starts with an empty window from where you can create a new project or open a previously created one.



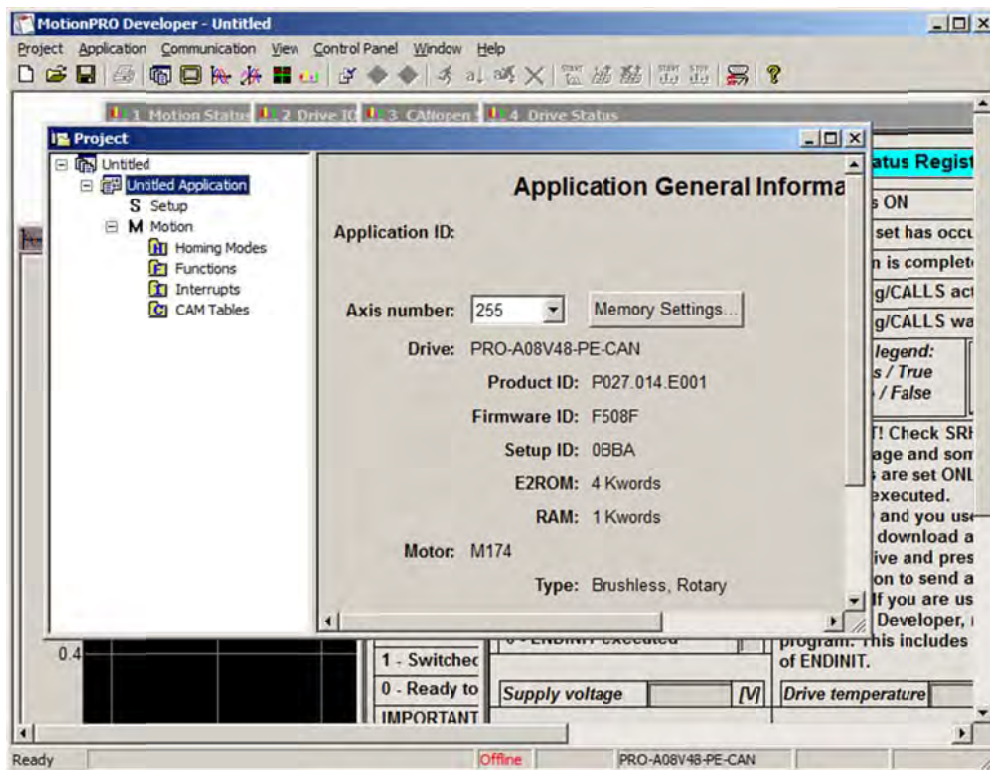
When you start a new project, MotionPRO Developer automatically creates a first application. Additional applications can be added later. You can duplicate an application or insert one defined in another project.



Press **New** button to open the “New Project” dialogue. Set the axis number for your first application equal with your drive/motor axis ID. The initial value proposed is 255 which is the default axis ID of the drives/motors. Press **New** button and select your drive/motor type. Depending on the product chosen, the selection may continue with the motor technology (for example: brushless motor, brushed motor, 3 phase stepper), the control mode (for example open-loop or closed-loop) and type of feedback device (for example: incremental encoder, SSI encoder)



Validate your selection with a mouse click. MotionPRO Developer opens the Project window where on the left side you can see the structure of a project. At beginning both the new project and its first application are named “Untitled”. The application has 2 components: **S** Setup and **M** Motion (program).



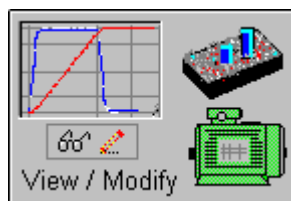
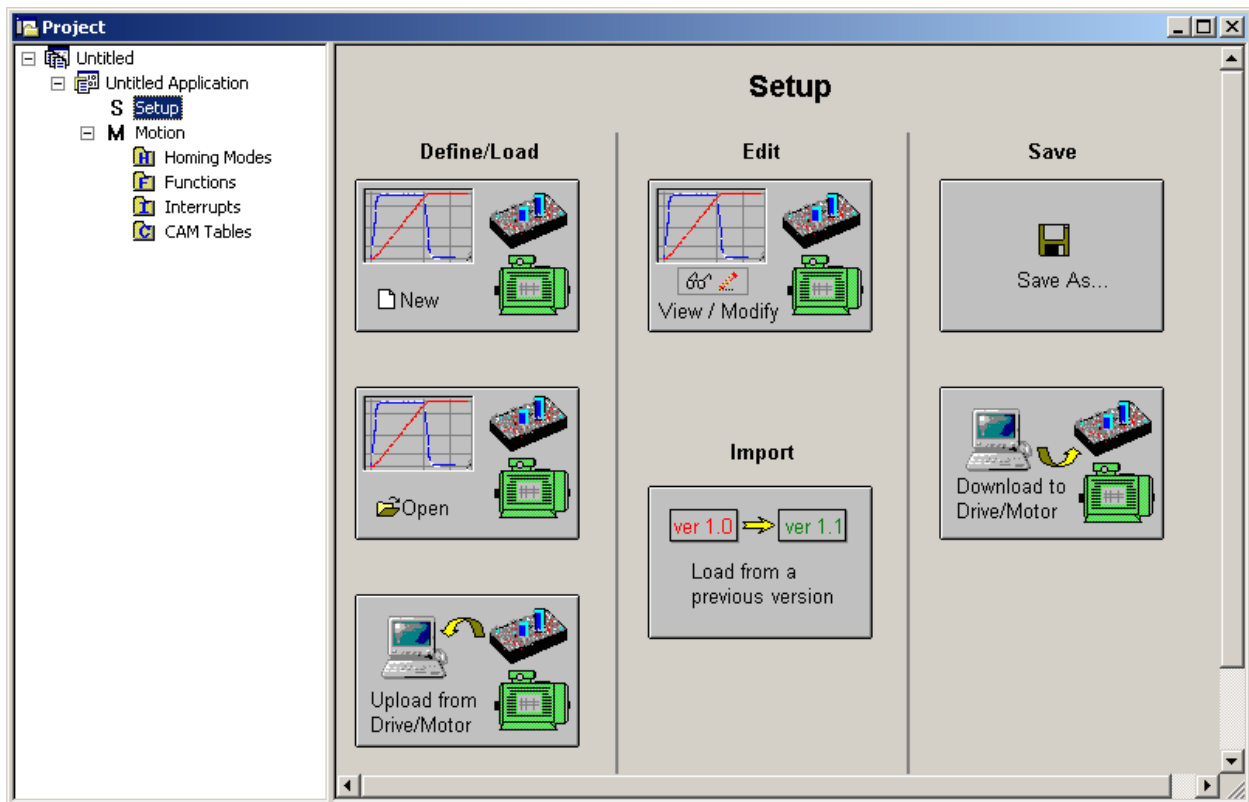
Step 2 Establish communication

If you have a drive/motor connected with your PC, now its time to check the communication. Use menu command **Communication | Setup** to check/change your PC communication settings. Press the **Help** button of the dialogue opened. Here you can find detailed information about how to setup your drive/motor and the connections. Power on the drive/motor, then close the Communication | Setup dialogue with OK. If the communication is established, MotionPRO Developer displays in the status bar (the bottom line) the text “**Online**” plus the axis ID of your drive/motor and its firmware version. Otherwise the text displayed is “**Offline**” and a communication error message tells you the error type. In this case, return to the Communication | Setup dialogue, press the Help button and check troubleshoots.

Remark: When first started, MotionPRO Developer tries to communicate with your drive/motor via RS-232 and COM1 (default communication settings). If your drive/motor is powered and connected to your PC port COM1 via an RS-232 cable, the communication can be automatically established.

Step 3 Setup drive/motor

In the project window left side, select “**S Setup**”, to access the setup data for your application.



Press **View/Modify** button. This opens 2 setup dialogues: for **Motor Setup** and for **Drive setup** through which you can configure and parameterize a ElectroCraft drive/motor. In the

Motor setup dialogue you can introduce the data of your motor and the associated sensors. Data introduction is accompanied by a series of tests having as goal to check the connections to the drive and/or to determine or validate a part of the motor and sensors parameters. In the **Drive setup** dialogue you can configure and parameterize the drive for your application. In each dialogue you will find a **Guideline Assistant**, which will guide you through the whole process of introducing and/or checking your data.



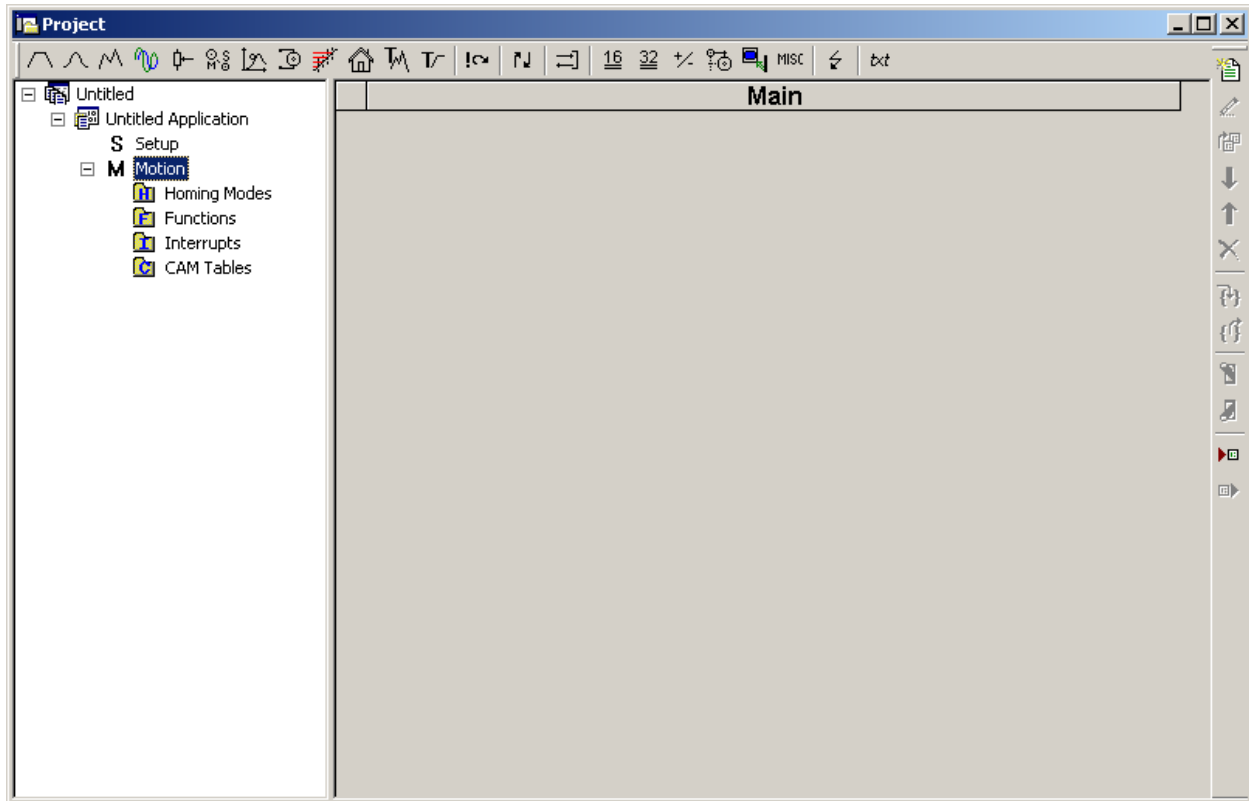
Press the **Download to Drive/Motor** button to download your setup data in the drive/motor EEPROM memory in the *setup table*. From now on, at each power-on, the setup data is copied into the drive/motor RAM memory which is used during runtime. It is also possible to save the setup data on your PC and use it in other applications. Note that you can upload the complete setup data from a drive/motor.

To summarize, you can define or change the setup data of an application in the following ways:

- create a new setup data by going through the motor and drive dialogues
- use setup data previously saved in the PC
- upload setup data from a drive/motor EEPROM memory

Step 4 Program motion

In the project window left side, select “**M Motion**”, for motion programming.



One of the key advantages of the ElectroCraft drives/motors is their capability to execute complex motions without requiring an external motion controller. This is possible because ElectroCraft drives/motors include both a state of art digital drive and a powerful motion controller.

Programming motion on a ElectroCraft drive/motor means to create and download a MPL (ElectroCraft Motion Program Language) program into the drive/motor memory. The MPL allows you to:

- Set various motion modes (profiles, PVT, PT, electronic gearing or camming, etc.)
- Change the motion modes and/or the motion parameters
- Execute homing sequences
- Control the program flow through:
 - Conditional jumps and calls of MPL functions
 - MPL interrupts generated on pre-defined or programmable conditions (protections triggered, transitions on limit switch or capture inputs, etc.)
 - Waits for programmed events to occur
- Handle digital I/O and analogue input signals
- Execute arithmetic and logic operations
- Perform data transfers between axes
- Control motion of an axis from another one via motion commands sent between axes
- Send commands to a group of axes (multicast). This includes the possibility to start simultaneously motion sequences on all the axes from the group
- Synchronize all the axes from a network

With MPL, you can really distribute the intelligence between the master and the drives/motors in complex multi-axis applications. Thus, instead of trying to command each step of an axis movement, you can program the drives/motors using MPL to execute complex tasks and inform the master when these are done. Thus for each axis the master task may be reduced at: calling MPL functions (with possibility to abort their execution if needed) and waiting for a message, which confirms the execution. If needed, the drives/motors may also be programmed to send periodically information messages to the master so it can monitor a task progress.

In order to help you create a MPL program, MotionPRO Developer includes a [Motion Editor](#). This offers you the possibility to program all the motion sequences using high level graphical dialogues which automatically generate the corresponding MPL instructions. With Motion Editor you can develop motion programs using almost all the MPL instructions without needing to learn them.

The Motion Editor is automatically activated when you select “**M Motion**” in the project window left side. When activated, Motion Editor adds a set of toolbar buttons in the project window just below the title. Each button opens a programming dialogue. When a programming dialogue is closed, the associated MPL instructions are automatically generated. Note that, the MPL instructions generated are not a simple text included in a file, but a motion object. Therefore with Motion Editor you define your motion program as a collection of motion objects.

The major advantage of encapsulating programming instructions in motion objects is that you can very easily manipulate them. For example, you can:

- Save and reuse a complete motion program or parts of it in other applications
- Add, delete, move, copy, insert, enable or disable one or more motion objects
- Group several motion objects and work with bigger objects that perform more complex functions

As a starting point, push for example the leftmost Motion Editor button – Trapezoidal profiles, and set a position or speed profile. Then press the **Run** button. At this point the following operations are done automatically:

- A MPL program is created by inserting your motion objects into a predefined template
- The MPL program is compiled and downloaded to the drive/motor
- The MPL program execution is started

For learning how to send commands from your host/master, check the Application | [Binary Code Viewer](#). This tool helps you to quickly find how to send MPL commands using one of the communication channels and protocols supported by the drives/motors. Using this tool, you can get the exact contents of the messages to send as well as of those expected to be received as answers.

Step 5 Evaluate motion application performances

MotionPRO Developer includes a set of evaluation tools like the [Data Logger](#), the [Control Panel](#) and the [Command Interpreter](#) which help you to quickly measure and analyze your motion application.

Step 6 Create an EEPROM image file for programming in production

Once you have validated your application, you can create with the menu command **Application | Create PRO EEPROM Programmer File** a software file (with extension **.sw**) which contains all the data to write in the EEPROM of your drive/motor. This includes both the setup data and the motion program. The **.sw** file can be programmed into a drive/motor, using the [PRO EEPROM Programmer](#) tool, which comes with MotionPRO Developer but may also be installed separately. The PRO EEPROM Programmer was specifically designed for repetitive fast and easy setup and programming of ElectroCraft drives/motors in production.

See also:

[MotionPRO Developer Workspace](#)

2. Project Management

2.1. Project File Concept

MotionPRO Developer works with **projects**. A project contains one or several **Applications**.

Each application describes a motion system for one axis. It has 2 main components: the **Setup** data and the **Motion** program and an associated axis number: an integer value between 1 and 255. Applications for ElectroCraft Motion Controller contain also a third component **Axis Selection**.

An application may be used either to describe:

1. One axis in a multiple-axis system
2. An alternate configuration (set of parameters) for the same axis.

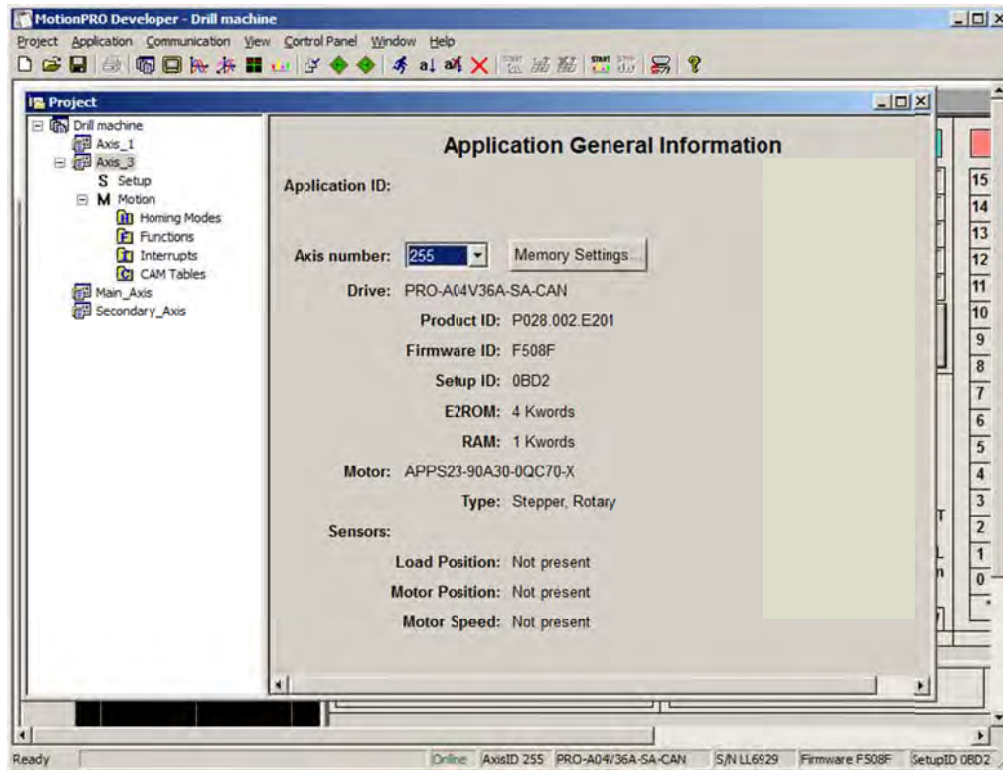
In the first case, each application has a different axis number corresponding to the axis ID of the drives/motors from the network. All data exchanges are done with the motion controller/programmable drive or motor having the same address as the selected application. In the second case, all the applications have the same axis number.

The setup component contains all the information needed to configure and parameterize a ElectroCraft drive/motor. This information is preserved in the drive/motor EEPROM in the *setup table*. The setup table is copied at power-on into the RAM memory of the drive/motor and is used during runtime.

The motion component contains the motion sequences to do. These are described via a MPL (ElectroCraft Motion Program Language) program, which is executed by the built-in motion controller.

In case of motion controller applications the Axis Selection allows multi-axes system description. The information is used by the motion controller to configure and command the slave axes.

When you start a new project, MotionPRO Developer automatically creates a first application. Additional applications can be added later. You can duplicate an application or insert one defined in another project.



When you select an application from the left side selection tree, the **Application General Information** view opens on the right, summarizing the basic data:

- **Application ID:** contains an array of characters you can create to quickly identify an application. The application ID is set in the setup component, the Drive Setup dialogue at Drive Info. The application ID is saved in the drive/motor EEPROM with the setup data
- **Axis number:** must match the axis Axis ID of the associated motion controller/programmable drive or motor.
- **Memory Settings:** shows how the associated motion controller/programmable drive or motor memory is used and allows you to modify the space reserved for different sections to match your application needs
- **Drive**
 - **Product ID:** displays the drive/motor execution/order code. ElectroCraft writes it in a reserved area of the EEPROM.
 - **Firmware ID:** shows the firmware required by the selected configuration. The actual firmware on the drive/motor must have the same number and a revision letter equal or higher.
 - **Setup ID:** displays the setup configuration
 - **E2ROM:** shows the size of the drive/motor E2ROM memory.
 - **RAM:** shows the size of drive/motor RAM memory.
- **Motor:** displays the name of the motor used
 - **Type:** presents the motor type: brushless, brushed, stepper: rotary or linear

-
- **Sensors:** presents the sensors used for the load and motor position and for the motor speed (when these sensors are present)
 - **Load Position:** type of position sensor for the load.
 - **Motor Position:** type of position sensor for the motor
 - **Motor Speed:** type of speed sensor for the motor

On the selection tree, for each application selected, you can access the 2 main components: the **Setup** data and the **Motion** program. The application tree for motion controller contains also the **Axis Selection**.

Continue with:

[Application – Setup](#)

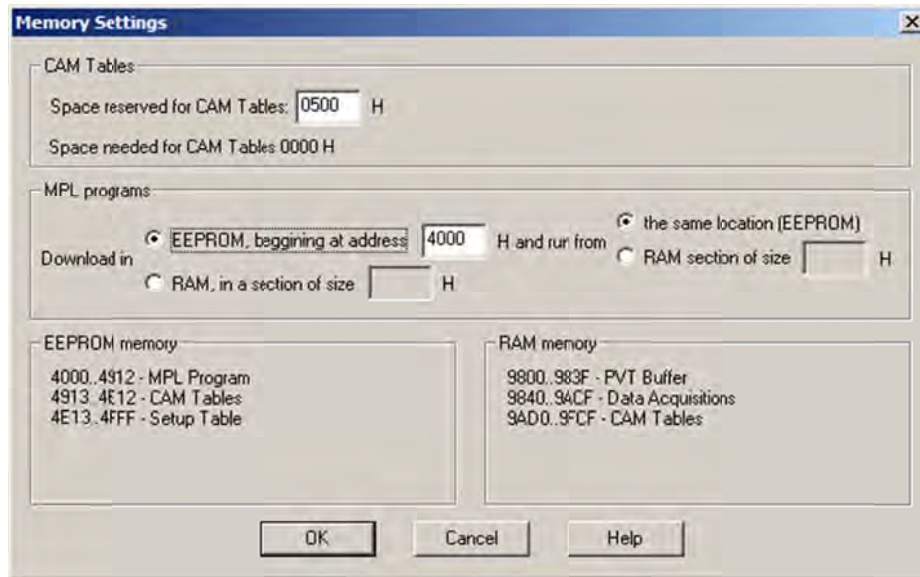
[Application – Motion](#)

See also :

[MotionPRO Developer Workspace](#)

2.2. Memory Setting

The Memory Settings dialogue allows you to customize the memory space reserved for different sections of your application as well as where the MPL program is loaded and executed. The memory settings refer to 2 types of memories: RAM and EEPROM.



The RAM memory has an area reserved for PVT / PT buffer followed by a customizable area. This is typically used for the [Logger](#) data acquisitions and to store the cam tables during runtime. It may also be used to temporarily store MPL programs.

The EEPROM memory has an area reserved for the setup table preceded by a customizable area. This is used to store the MPL programs and the cam tables.

The exact amount of EEPROM memory is specific for each drive/motor.

In the CAM Tables section, you can adjust the space reserved for the cam tables selected to be used in your application. The cam tables are first downloaded into the EEPROM memory and at runtime are copied into the RAM memory. Therefore, the cam tables' space is reserved in both RAM and EEPROM memories. You can find how much of the space reserved is really occupied by the cam tables from the [CAM Tables View](#) which shows you at **Buffer Free Space** the remaining space reserved for cam tables.

If your application doesn't use cam tables you can free the space reserved to increase the space allocated for data acquisition.

In the MPL program section, you can choose where to download and execute the MPL program. Typically you download and execute the MPL program in the non-volatile EEPROM from, starting from its first location (4000h) which is checked at power on in the [AUTORUN](#) mode.

If your configuration includes an absolute encoder with position read via SSI or EnDat protocols, the MPL program **MUST** be downloaded into the EEPROM and executed from the RAM. In these cases, when the MPL program executes the ENDINIT command, the EEPROM memory can no longer be accessed. Therefore, for these configurations, MotionPRO Developer automatically adds to your MPL program a copy sequence which is executed immediately as the MPL program starts to run. The copy sequence, copies your MPL program from the EEPROM memory into the RAM memory and then it passes the control. The whole process is transparent for the user, whose only obligation is to set the download address in the EEPROM and the run address in the RAM.

For test purposes, you can also download and run the MPL programs from the RAM memory. This option speeds up the download process and may be useful if your MPL program is large and you intend to execute a lot of tests.

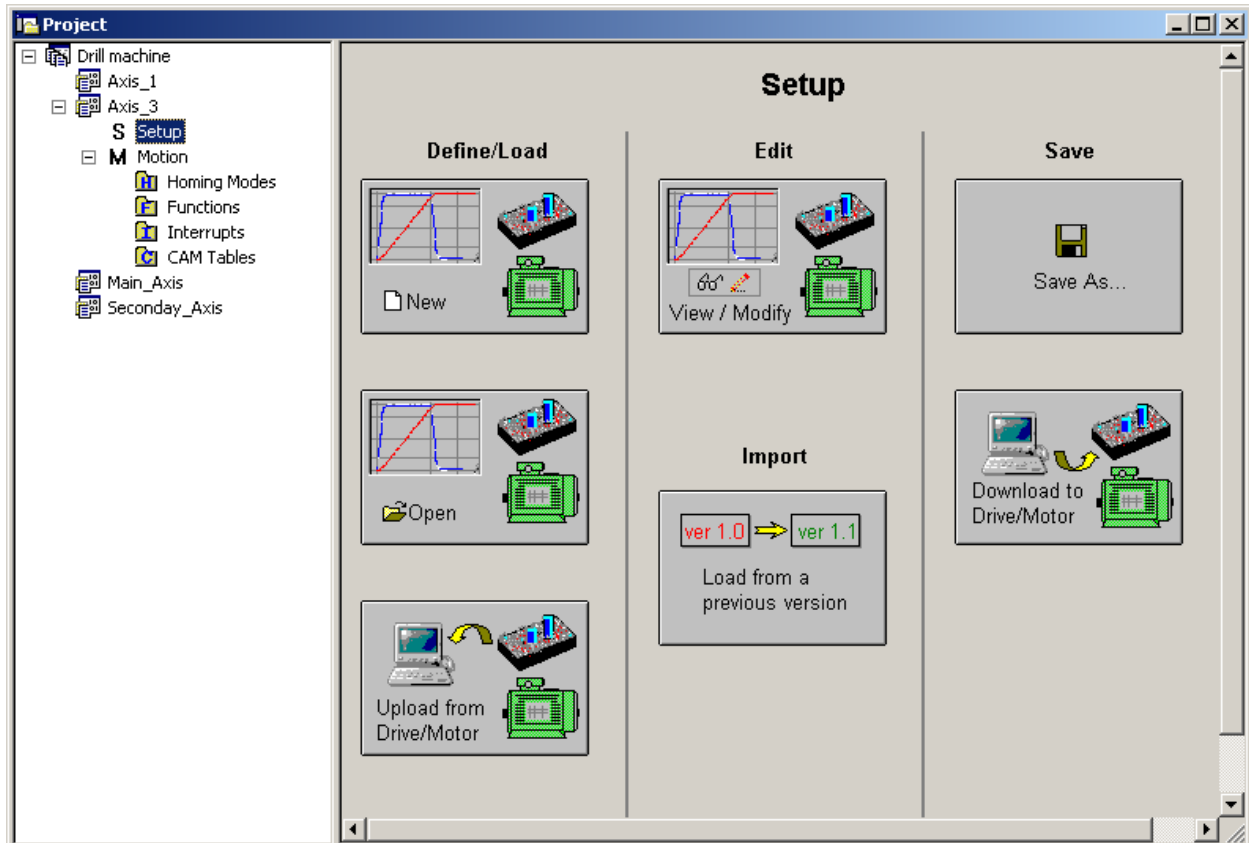
See also:

[Project Concept](#)

[Memory Map](#)

2.3. Application - Setup

In the **Setup** view you can create a new configuration, load a previously saved setup, upload from drive/motor, view or change the selected configuration, save or download the configuration.



The setup view is split in 3 sections as follows:

- In the left section you can define or load a setup configuration:
 - Create a **New** drive/motor setup. Opens the “**Select ElectroCraft Product**” dialogue. Select your drive/motor type. Depending on the product chosen, the selection may continue with the motor technology (for example: brushless motor, brushed motor) or the control mode (for example stepper – open-loop or stepper – closed-loop) and type of feedback device (for example: incremental encoder, SSI encoder).
 - **Open** an existing setup. Loads a drive/motor setup configuration, which was previously defined and saved on your PC. The command opens the “**Select Drive/Motor Setup**” dialogue, allowing you to select a drive/motor setup. Note that a setup is not a single file, but a collection of files which are saved together in the same folder. The folder name is the name of the setup. Hence you select a setup by choosing a folder. By default, **MotionPRO Developer** saves the setup data in “**Setup Files**” subdirectory of the **MotionPRO Developer** main folder
 - **Upload from Drive/Motor** the setup data.
- In the middle section you can view and edit the setup data. You may also load and convert setup data from a previous versions:

-
- **View/Modify** setup. Opens 2 setup dialogues: for **Motor Setup** and for **Drive setup** through which you can configure and parameterize a ElectroCraft drive/motor. In the **Motor setup** dialogue you can introduce the data of your motor and the associated sensors. Data introduction is accompanied by a series of tests having as goal to check the connections to the drive and/or to determine or validate a part of the motor and sensors parameters. In the **Drive setup** dialogue you can configure and parameterize the drive for your application.
 - **Load from a previous version.** Converts setup data from a previous version into the up-to-date version. The command is foreseen to provide migration of setup data defined long time ago into the latest version for that configuration. The command has sense only if there are differences between the user interface opened with the old setup data and that opened for the same configuration (drive, motor and sensor) with the command New.
 - In the right section, you can save the setup data on your PC or download it on the drive/motor
 - **Save.** Opens “Save Drive/Motor Setup” dialogue where you can select where to save the setup data. Note that a setup is not a single file, but a collection of files which are saved together in the same folder. The setup name gives the name of the associated folder. By default, **MotionPRO Developer** saves the setup data in "**Setup Files**" subdirectory of the **MotionPRO Developer** main folder
 - **Download to Drive/Motor.** The command will download the actual setup data into the drive EEPROM in the *setup table*.

See also:

[Application – Motion](#)

[Application – Axis Selection](#)

[Motion Project Concept](#)

2.4. Application - Motion

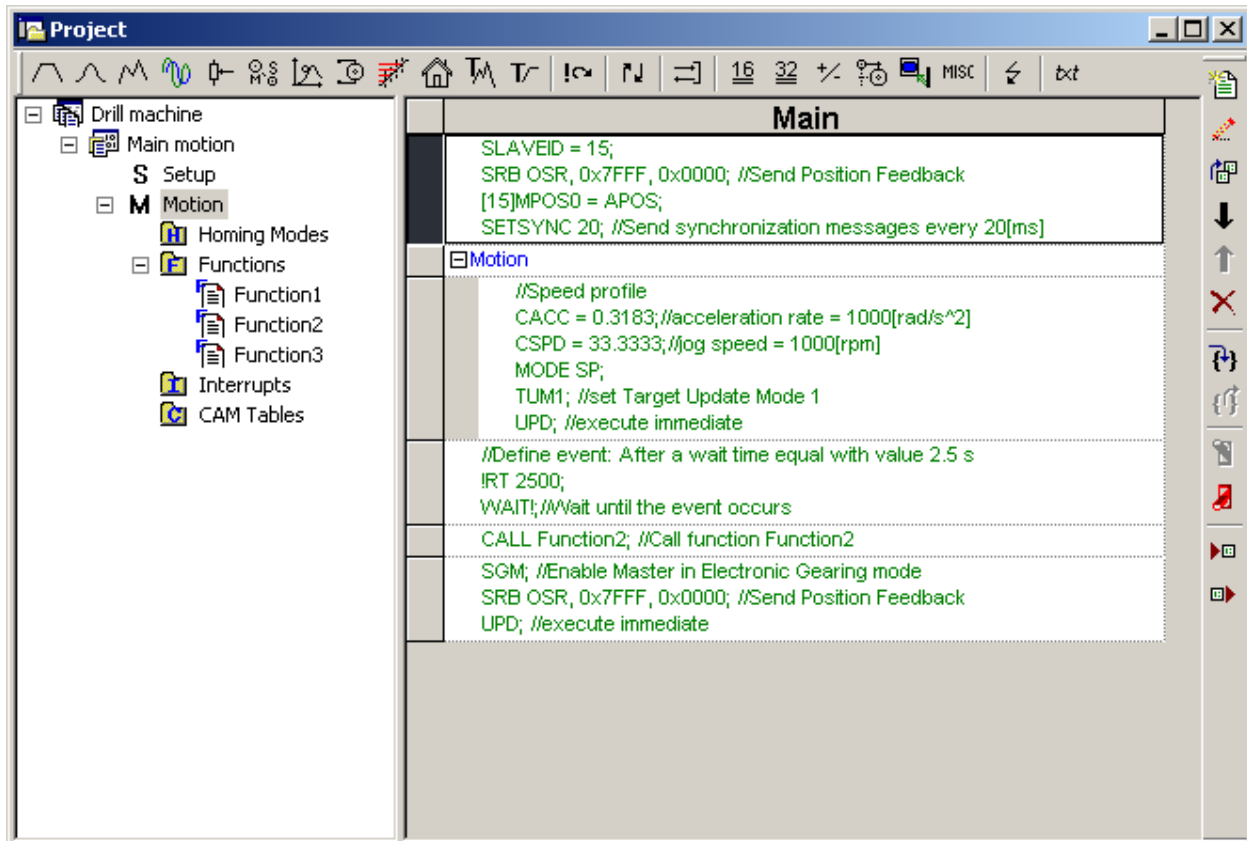
In the **Motion** part of an application, you can program the motion sequences to be executed by the motion controller (dedicated or built-in programmable drives/motors).

Programming motion on a ElectroCraft motion controller or programmable drive/motor means to create and download a MPL (ElectroCraft Motion Program Language) program into the motion controller/drive/motor memory. The MPL allows you to:

- Set independent motion modes (profiles, PVT, PT, electronic gearing or camming, etc.)
- Set 2D/3D coordinate motion modes (Vector Mode, Linear Interpolation)
- Change the motion modes and/or the motion parameters
- Execute homing sequences
- Control the program flow through:
 - Conditional jumps and calls of MPL functions
 - MPL interrupts generated on pre-defined or programmable conditions (protections triggered, transitions on limit switch or capture inputs, etc.)
 - Waits for programmed events to occur
- Handle digital I/O and
- analogue input signals
- Execute arithmetic and logic operations
- Perform data transfers between axes
- Slave axes management from Motion Controller
- Control motion of an axis from another one via motion commands sent between axes
- Send commands to a group of axes (multicast). This includes the possibility to start simultaneously motion sequences on all the axes from the group
- Synchronize all the axes from a network

A MPL program includes a **main** section, followed by the subroutines used: **functions**, **interrupt** service routines and **homing** procedures. The MPL program may also include **cam tables** used for electronic camming applications.

When you select the “**M** Motion” part of an application, you access the main section of your application MPL program.



You can select the other components of a MPL program too. Each has 2 types of access views:

- **Definition and/or selection view**, with the following purposes:
 - **Homing Modes:** select the homing procedure(s) to use from a list of already defined procedures.
 - **Functions:** create new MPL functions (initially void) and manipulate those defined: delete, rename, change their order in the program
 - **Interrupts:** choose the MPL interrupt service routines you want to view/change their default implementation
 - **Cam Tables:** create new cam tables loaded from other applications or imported from text files and manipulate those defined: select those to be downloaded and their order, delete or rename.

Remark: The Cam Table are available only in applications developed for programmable drive/motors.

- **Edit view** – for editing the contents. There is one edit view for each homing procedure and cam table selected, for each function defined and each interrupt chosen for view/edit.

When you start a new application the edit views of the above components are not present as there is none defined. After you have defined/selected the first homing procedure(s), function(s), interrupt(s) or cam table(s), select again the corresponding view in the project window left side tree. Below it, you'll see the component(s) defined/created. Choose one and on the right side you'll see the corresponding edit view.

In order to help you create a MPL program, MotionPRO Developer includes a **Motion Editor** which is automatically activated when you select “**M Motion**” – the main section view or an edit view for a homing procedure, function or interrupt service routine. The Motion Editor adds a set of [toolbar buttons](#) in the project window just below the title bar. Each button opens a programming dialogue. When a programming dialogue is closed, the associated MPL instructions are automatically generated. Note that, the MPL instructions generated are not a simple text included in a file, but a motion object. Therefore with Motion Editor you define your motion program as a collection of motion objects.

The major advantage of encapsulating programming instructions in motion objects is that you can very easily manipulate them. For example, you can:

- Save and reuse a complete motion program or parts of it in other applications
- Add, delete, move, copy, insert, enable or disable one or more motion objects
- Group several motion objects and work with bigger objects that perform more complex functions

See also:

[Motion Editor toolbar buttons for motion programming](#)

[MotionPRO Developer Workspace](#)

[Homing Modes](#)

[Functions](#)

[Interrupts](#)

[CAM Tables](#)

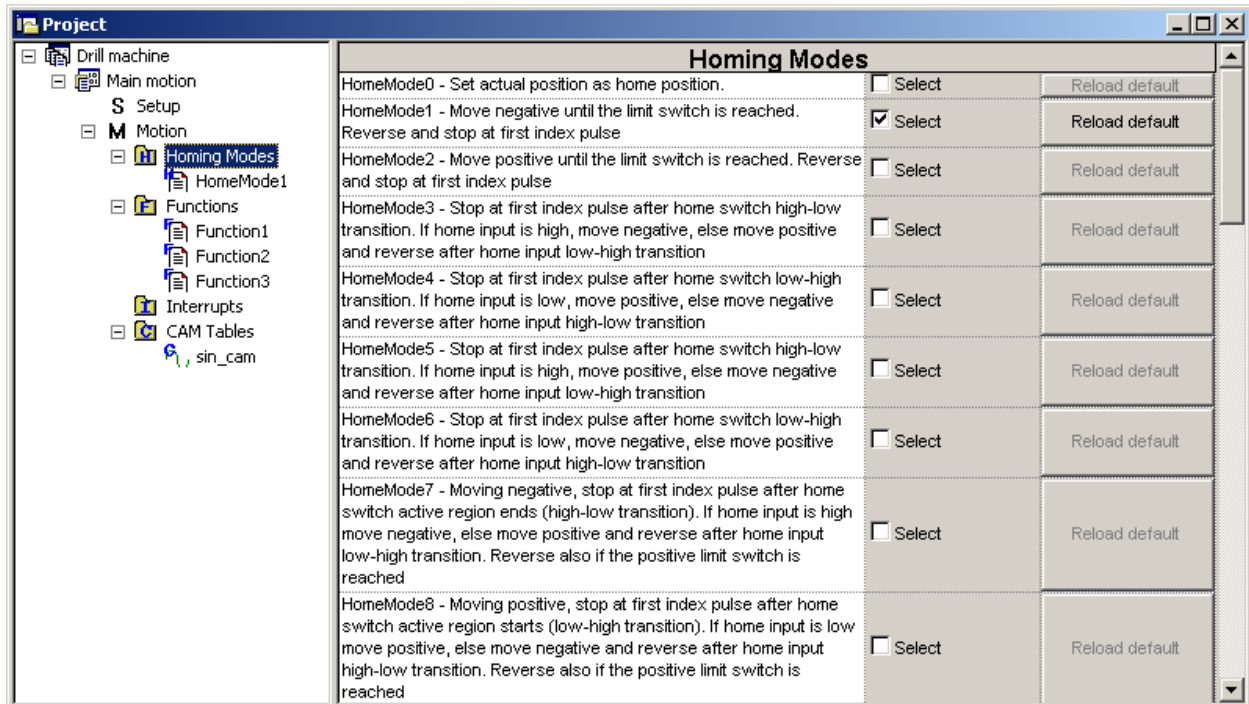
[Application – Setup](#)

[Application – Axis Selection](#)

[Motion Project Concept](#)

2.4.1. Homing Modes

This view allows you to choose the homing procedures associated with the selected application.



Electrocraft provides for each programmable drive/motor a collection of up to 32 homing procedures. These are predefined MPL functions, which you may call after setting the homing parameters. You may use these homing procedures as they are, or you may modify them according with your application needs.

In this view you can see all the homing procedures defined for your drive/motor, together with a short description of how it works. In order to select a homing procedure, check its associated button. You may choose more than homing procedure, if you intend to use execute different homing operations in the same application. The selected homing modes appear in the project window left side selection tree, in the current application, as a sub-tree of the **Homing Modes** section. Select a homing procedure from this list. On the right side you'll see the associated function in the [Homing Procedures Edit](#). Here you can check and modify the contents of the selected homing procedure(s).

Remark: Only the selected homing modes are available as options in the Motion – Homing dialogue.

Once modified, a homing procedure is memorized together with the application. However, if you'll create a new application, the homing procedure changes will are not preserved. If you want to preserve them, either create the new application by duplicating that with modified homing procedures, or load the entire motion from the application with modified homing procedures.

Press the **Reload Default** button to restore the default homing procedure.

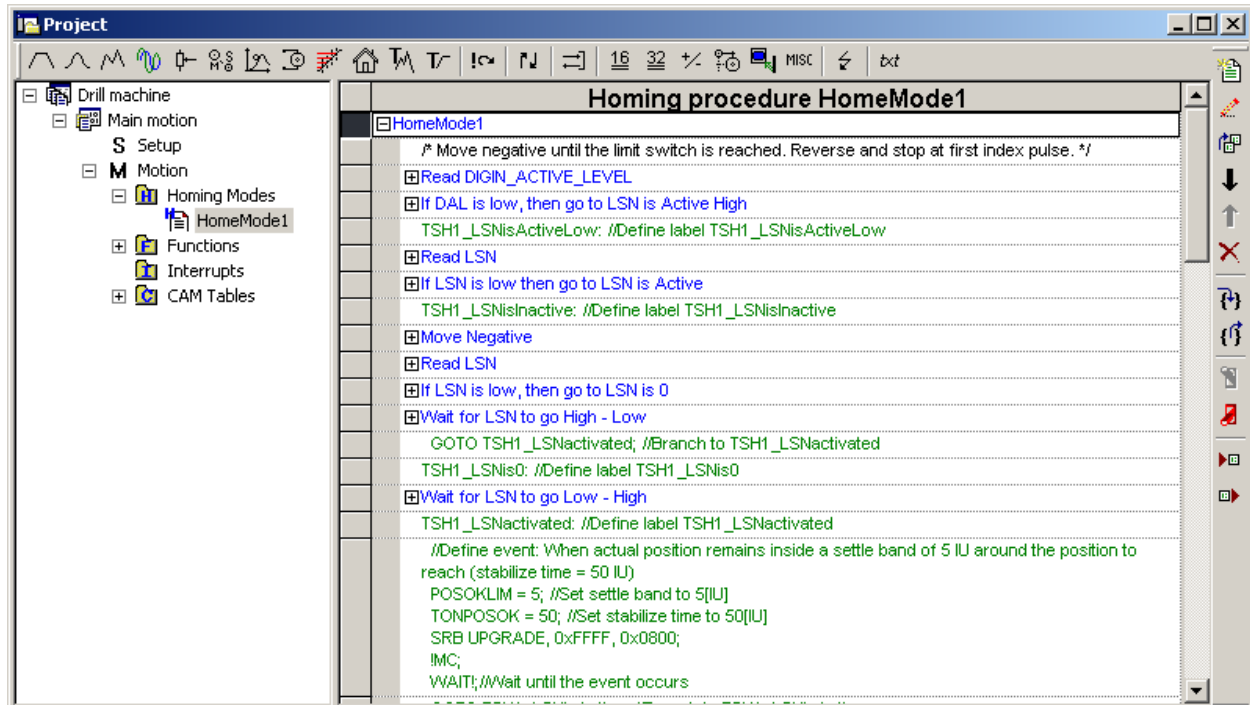
See also:

[Homing Modes Edit](#)

[Application – Motion](#)

2.4.2. Homing Modes Edit

In the Homing Procedures Edit, you can view and modify the contents of the homing procedure selected on the left-side tree. This is a standard motion view offering access to all the MPL programming features.



See also:

[Motion Editor toolbar buttons for motion programming](#)

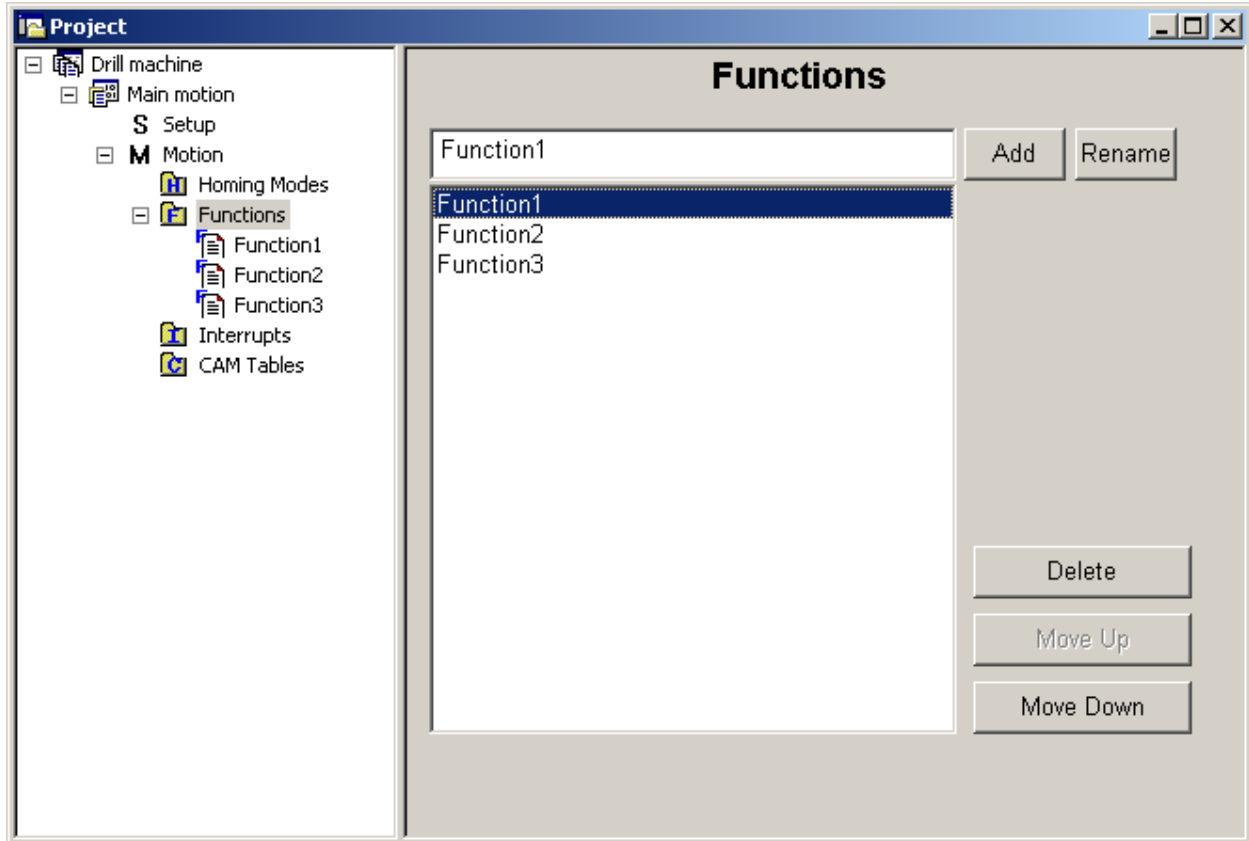
[MotionPRO Developer Workspace](#)

[Homing Modes](#)

[Application – Motion](#)

2.4.3. Functions

This view allows you to add and remove the MPL functions associated with the selected application. You may also rename and change the functions download order.



Type in the edit the name, and press the **Add** button to create a new function. Select a function from the list and press **Rename** to change its name, **Delete** to erase it, **Move Up** or **Move Down** to change its position in the list.

The MPL functions defined appear in the project window left side selection tree, in the current application, as a sub-tree of the **Functions** section. Select a function from this list. On the right side you'll see the function contents in the [Functions Edit](#).

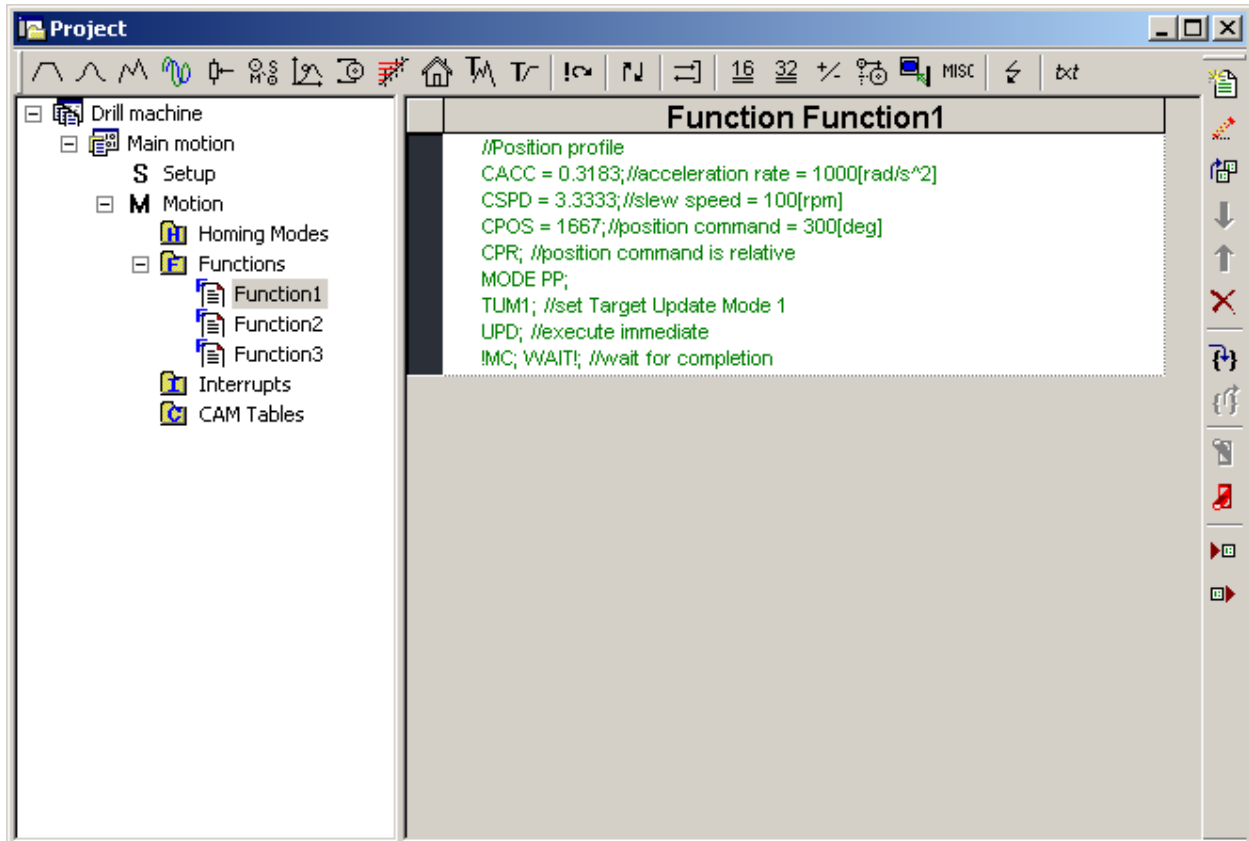
See also:

[Functions Edit](#)

[Application – Motion](#)

2.4.4. Functions Edit

In the Functions Edit, you can view and modify the contents of the MPL function selected on the left-side tree. This is a standard motion view offering access to all the MPL Programming features.



See also:

[Motion Editor toolbar buttons for motion programming](#)

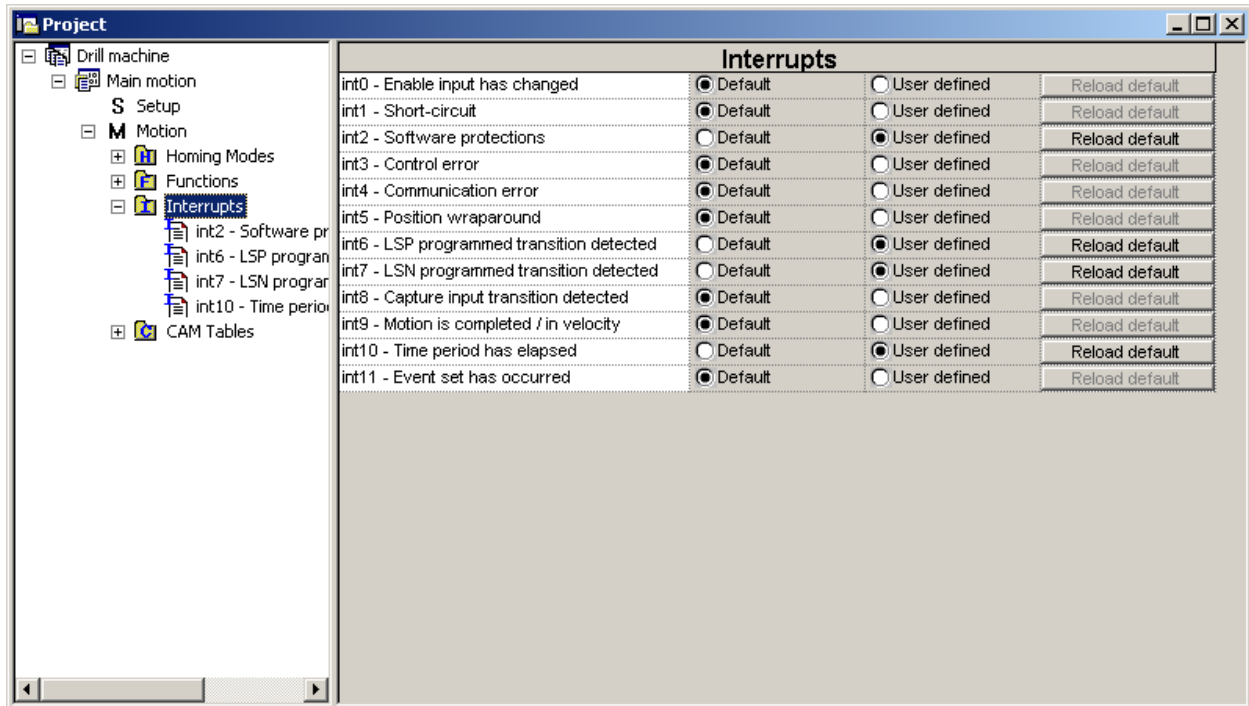
[MotionPRO Developer Workspace](#)

[Functions](#)

[Application – Motion](#)

2.4.5. Interrupts

This view allows you to see, define and modify the MPL interrupt service routines (ISR).



Each drive/motor has a default ISR for each of the 12 MPL interrupts. In order to use the default ISR, select **Default** for all the MPL interrupts. If you want to see or modify any of the default ISR, choose option **User defined**. The MPL interrupts with option **User defined** appear in the project window left side selection tree, in the current application, as a sub-tree of the **Interrupts** section. Select an interrupt from this list. On the right side you'll see the ISR contents in the [Interrupt Edit](#). Here you can check and modify the selected ISR according with your needs.

In can cancel your modifications and to return to the starting point i.e. the default ISR by pressing **Reload Default** button. You can also return at any moment to the default ISR by selecting again the **Default** option.

Remark: Some of the drive/motor protections may not work properly if the MPL Interrupts are handled incorrectly. In order to avoid this situation keep in mind the following rules:

- The MPL interrupts must be kept globally enabled to allow execution of the ISR for those MPL interrupts triggered by protections. As during a MPL interrupt execution, the MPL interrupts are globally disabled, you should keep the ISR as short as possible, without waiting loops. If this is not possible, you must globally enable the interrupts with EINT command during your ISR execution.
- If you modify the interrupt service routines for **Int 0** to **Int 4**, make sure that you keep the original MPL commands from the default ISR. Put in other words, you may add your own commands, but these should not interfere with the original MPL commands. Moreover, the original MPL commands must be present in all the ISR execution paths.

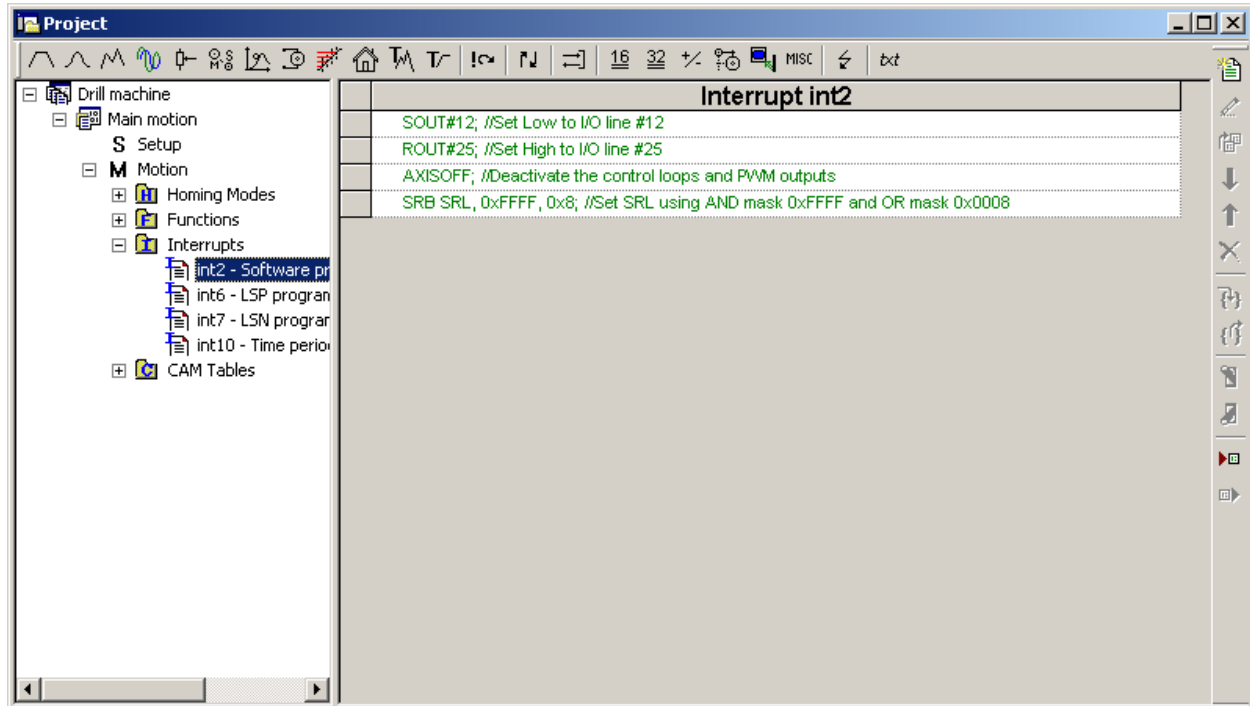
See also:

[Interrupts Edit](#)

[Application – Motion](#)

2.4.6. Interrupts Edit

In the ISR Edit, you can view and modify the contents of the service routine for the MPL interrupt selected on the left-side tree. This is a standard motion view offering access to all the MPL programming features.



See also:

[Motion Editor toolbar buttons for motion programming](#)

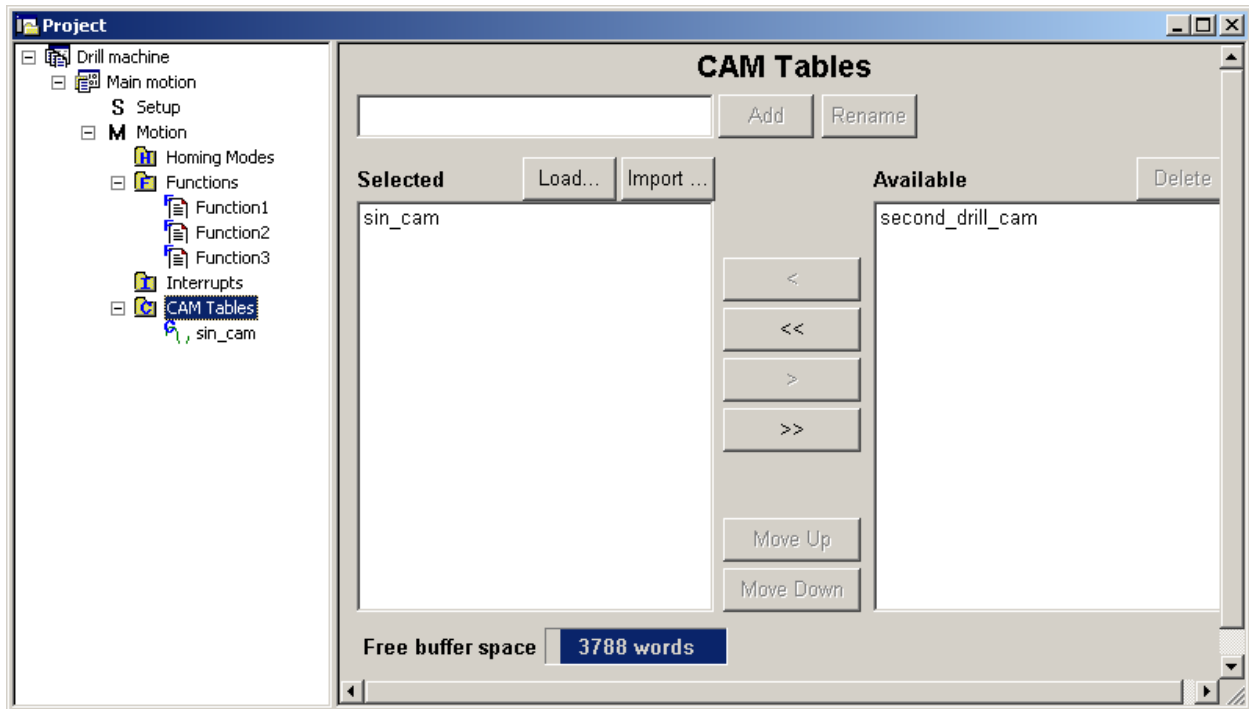
[MotionPRO Developer Workspace](#)

[Interrupts](#)

[Application – Motion](#)

2.4.7. CAM Tables

This view allows you to specify the cam tables associated with the selected application.



You can:

- **Load** cam files (with extension **.cam**) defined in other applications
- **Import** cam tables from text files with format: 2 columns, one for X, the other for Y, separated by space or tab
- **Add** new cam files

Once defined, a cam table can be placed in one of the following two lists: **Available** or **Selected**. Move in the **Selected** list those cam tables you intend to use in your application. You can have one or more cam tables, up to the limit of the memory space reserved for cams (the remaining memory for cam tables is shown by **Free Buffer Space** indicator). Use **Move Up** and **Move Down** buttons to change the cam tables order in the **Selected** list i.e. the order in which these cam tables will be downloaded. Move in the **Available** list all the cam tables you don't use now, but may use later. Use **Delete** to remove a cam table from the **Available** list. Select a cam from either list and change its name with **Rename** or use [<] or [>] to move it from one list to the other. Use [<<] or [>>] to move all the cams from one list to the other. By default, all the new added cam tables are placed in the category **Selected**.

Remark: Check the application [Memory Settings](#) if you want to change the space reserved for cam tables

First time when you run (i.e. press the **Run** button) a new application, the associated cam tables from the **Selected** list are automatically downloaded into the drive/motor together with the motion application. Later on, the cam tables download is repeated only if these are modified of the **Selected** list is changed.

There is also a dedicated menu command **Application | Motion | Download CAM Tables**, for the cam tables download.

Once defined, all the cams from the **Selected** list appear in the project window left side selection tree, in the current application, as a sub-tree of the **CAM Tables** section. Select a cam table from this list. On the right side you'll see graphically the cam profile in the [CAM Tables Edit](#), view, where you may edit the cam file.

When you create a new cam table, you must:

- Type its name in the edit field and press the **Add** button
- Select the cam table from the left side selection tree and edit or import the points

See also:

[CAM Tables Edit](#)

[Application – Motion](#)

2.4.8. CAM Tables Edit

In the CAM Tables Edit, you can view, modify, export or import a cam table. All these operations refer to the selected cam on the left-side tree.

The screenshot shows the 'sin_cam' window with the following data in the X,Y table:

X	Y
0	0
128	268
256	535
384	802
512	1067
640	1330
768	1591
896	1849
1024	2104
1152	2355
1280	2603
1408	2845
1536	3083
1664	3315

The cam tables are arrays of X, Y points, where X is the cam input i.e. the master position and Y is the cam output i.e. the slave position. The X points are expressed in the master internal position units, while the Y points are expressed in the slave internal [position units](#). Both X and Y points 32-bit long integer values. The X points must be positive (including 0) and equally spaced at: 1, 2, 4, 8, 16, 32, 64 or 128 i.e. having the interpolation step a power of 2 between 0 and 7. The maximum number of points for one cam table is 8192.

As the X points are equally spaced, these are completely defined by only 2 data: the **Master start value** or the first X point and the **Interpolation step** providing the distance between the X points.

When you create a new cam table, you may either import or edit its points.

Press the **Import...** button to import the cam table points from a simple text file (.txt), with 2 columns, first the X points and the column with Y points. A tab or a space must separate the columns.

In order to edit a cam table:

- Set the first X point value in **Master start value**
- Set a value between 0 and 7 in **Interpolation step 2[^]**
- Set the first **Y value** and press the **Insert** button. Repeat these operations until you define all the cam Y points.

Remark: *The X points are automatically calculated and displayed as you introduce the Y points.*

To navigate between the cam table points use [≤<], [>≥] buttons. Use **Remove** or **Update** to delete or change the currently selected cam table point.

You may also **Export** a cam table in the same text file format (.txt) used for import. When the project is saved, for each application, the associated cam files (.cam.) are saved in the application folder.

See also:

[CAM Tables](#)

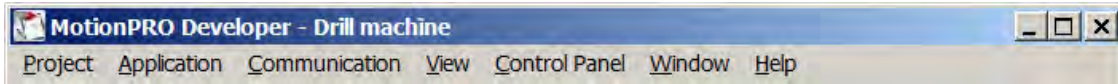
[Motion – Electronic Camming](#)

[Application – Motion](#)

3. MotionPRO Developer Workspace

Menu Bar

The **Menu Bar** is the toolbar at the top of the screen that contains all MotionPRO Developer menu commands.



When MotionPRO Developer creates a new project, besides the **Project** window, it opens also the **Logger** and 3 predefined **Control Panel** windows (**1_Motion Status** and **2_Drive IO** and **3_Drive Status**).

Toolbar

The buttons in the toolbar represent commonly used MotionPRO Developer commands.



Status bar

In the status bar you will find the following information:

- The communication status:
 - "Online" if the communication between the drive/motor associated with to the selected and MotionPRO Developer is established
 - "Offline" if the MotionPRO Developer can't communicate with the drive/motor associated to the selected application.
- Axis ID – the axis ID of the selected application if the communication between the drive/motor and MotionPRO Developer is established
- Product ID – the product ID of the drive/motor associated to the selected application
- Firmware ID – the version of the firmware found on the drive/motor
- Setup ID – the identification code for the setup configuration used in the selected application
- The coordinates of the mouse pointer in the data logger graphs. This information is available only when the Data Logger is selected.

See also:

[Motion Programming Toolbars](#)

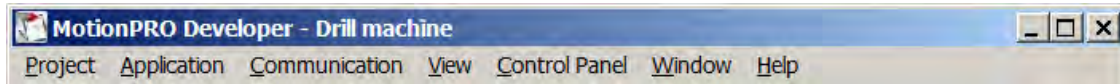
[Application – Motion](#)

[Functions Edit](#)

[Interrupts Edit](#)

3.1. Menu Bar

The **Menu Bar** is the toolbar that contains all MotionPRO Developer menu commands:



3.1.1. Project Menu

New. Use this command to create a new project. The "**New Project**" menu dialog will open.

Open. Use this command to open an already defined Motion System project that was previously saved and closed. An "**Open MotionPRO Developer project**" dialogue will open, allowing you to select a project name

Close. Use this command to close the current project.

Save. In order to save the changes done to the current project use this command.

Save As....Use this command to save the current project with a appropriate name. The "**Save Project**" window opens, allowing you to create a new folder where the project will be saved.

Archive. Use this command in order to compress all the files of a project. This command creates a unique file, having the name of the project and the suffix **.m.zip**.

Restore. An **.m.zip** file, saved by default in the /Archives subdirectory, can be copied into a different location (another computer), and then re-opened using this command. This will simplify the process of project transfer from one location to another, as the project consists from more files, which must be restored into specific sub-directories, in relation to the applications included in the project.

Print. to generate a printed image of some of the project windows (as logger, MPL source code), select the respective window, and use this command.

Print Preview. Use this menu command in order to preview the plot before printing

Print Setup. The command opens a dialogue where you configure the printer used by MotionPRO Developer

A list of the last projects opened.

Exit. In order to quit the **MotionPRO Developer** use this menu command.

3.1.2. Application Menu

New... Use this command in order to add a new application to your current project.

Duplicate. Use this command to duplicate the currently active application. This command creates an identical application in the project, asking you for a different application name. The duplicate operation copies all the contents of the directory associated to the copied application, and create the new one in the current project. This operation is useful if you want to modify an existing application, while keeping the original one unchanged. Selecting one application or another will allow you to execute and compare the two applications in a straightforward manner.

Insert... Using you can include an already defined application from a different project. This command opens a dialog that allows you to select the application to be copied. The import operation copies all the contents of the directory associated to the copied application, and create a new one in the current project. You can rename the application in order to change its name.

Edit..... This command opens the **Application Attributes** dialogue that allows you to rename the currently active application and/or to change the Axis ID.

Delete. You can delete the currently active application using this command.

[Setup](#)

[Motion](#)

Run. Use this menu command to start you application.

Axis On. This command enables the PWM signals of the drive/motor associated to the selected application.

Axis Off. This command disables the PWM signals of the drive/motor associated to the selected application.

Reset. Use this command to reset the drive/motor associated to the selected application.

Show slave errors. This command displays the errors reported by the slave axes to a multi-axes Motion Controller.

Binary Code Viewer The “Binary Code Viewer” is a tool included in MotionPRO Developer which offers you a quick way to find the binary code that must be sent / will be received by your host processor, when communicating with a ElectroCraft drive.

Create PRO EEPROM Program File. The option **Motion and Setup** creates a **.sw** file with complete information including setup data, MPL programs, cam tables (if present) and the drive/motor configuration ID. The option **Setup Only** produces a **.sw** file identical with that produced by PROconfig i.e. having only the setup data and the configuration ID.

Export to MPL_LIB. With this command you export the setup information, of the selected application, for use with MPL_LIB. The setup information will be stored in two files **setup.cfg** and **variables.cfg**.

3.1.3. Application | Setup Menu

New... Create a new drive/motor setup using this command. A new window “**Select ElectroCraft Product**” will open in which you need to select the template on which the new motion application will be based. The collection of **MotionPRO Developer** templates is organized for different configurations, based on the different types of ElectroCraft drives, and the associated types of motors that can be driven by these drives.

Open... Use this menu command for opening an already defined drive/motor setup that was previously saved and closed. A “**Select Drive/Motor Setup**” dialogue will open, allowing you to select a drive/motor setup name. By default, **MotionPRO Developer** saves each motion project as a separate directory having the same name as the project itself, in the “**Setup Files**” subdirectory of the **MotionPRO Developer** program directory.

Upload from Drive/Motor. Use this command to upload the setup from the drive/motor.

Import from a Previous Version... Converts setup data from a previous version into the up-to-date version. The command is foreseen to provide migration of setup data defined long time ago into the latest version for that configuration.

Edit/New. Using this menu command will open the drive setup window and motor setup window. In these windows you can view or change the drive and motor settings.

Download to Drive/Motor. Use this command to download the setup to drive/motor.

Save As... Use this command to save the setup on the disk. A “**Save Drive/Motor Setup**” window will open in which you can create a new folder for the setup to be saved in.

3.1.4. Application | Motion Menu

Build. Using this command you compile and link the MPL program, the result is a file with out extension ready to be downloaded to the drive/motor.

Download CAM Tables. Use the command to download on the drive/motor EEPROM the cam tables defined for the selected application.

Download Program. The command downloads on the drive/motor the out file created for the selected application. The out file is created using the command **Build**.

Load from Another Application... This command allows you to load the motion section defined in a different application. All motion section components (motion sequences, functions, ISR or homing sequence) of the current application are overwritten by this command.

Import Sequence... This command allows you to load/insert motion objects previously saved in *.msq files. These are appended below the current position e.g. the immediately after the selected motion object.

Export Sequence... Use this command to save a part of your program (one or more motion object) in a separate motion file .The operation saves the selected motion objects in a file with extension *.msq.

Import G-Code file... This command allows you to convert G-Code sequences into MPL motion language instructions for a multi-axis Motion Controller.

Edit. Use this command after select a motion sequence to change its parameters. The dialogue associated with the selected motion sequence opens.

Insert. Reserved for future developments.

View Generated MPL Code... This command allows you to view the MPL Code generated for the motion sequences selected in the Motion Editor dialogue.

Duplicate. This command duplicates the selected motion sequence.

Move Up. This command moves up the selected motion sequence.

Move Down. This command moves down the selected motion sequence.

Delete. This command allows you to delete the selected motion sequence.

Group. This command allows you to group a number of motion sequences in a new object containing all the selected motion objects

Ungroup. Use this command to restore the motion objects list instead of the group object.

Enable For debugging, you have the possibility to remove motion sequences (one or more motion objects) from the motion program like commenting lines in a text program. Use this menu command to uncomment (enable) the selected motion sequences.

Disable .Use this command to comment (disable) the selected motion sequences.

Add Function. This command creates a new function (named “Untitled”) and the “Function Window” will open. In this window you can insert the motion sequences to be executed when the function is called.

Delete Function. This command deletes the currently selected function.

3.1.5. Communication Menu

Setup... The “**communication setup**” dialogue will open which allows you to select the communication type between RS-232, RS-485 and CAN-bus with several PC to CAN interface boards, to choose the desired baud rate and to setup the communication parameters.

Refresh Select the command if during operation the communication is interrupted (for example if the drives power is turned off) in order to restore communication

Work Offline When this option is selected the MotionPRO Developer doesn't attempt to communicate with the drives/motors associated to the defined applications.

Show Info In Output View | The menu command allows selecting the information listed in the output view. The output view is showed/hided from menu **View | Output**.

None – when you select this option no information is presented in Output view.

Errors – use this option to view errors occurred during communication. Errors due to programming error, detected during program build, are automatically listed in Output View.

Warnings – when this option is selected in the Output view appears the warning messages of the selected communication channel.

Traffic – when this option is selected in the Output view are listed all messages send or received by MotionPRO Developer.

Unrequested messages – use this option to list messages send automatically by the drive connected to the PC.

EEPROM Write Protection. From this menu you access the options related to EEPROM write protection feature. You have the following options:

Do not protect EEPROM after download – when this option is selected the EEPROM is not protected

Write protect last ¼ of EEPROM after download – when this option is selected the last quarter of the EEPROM is write protected after the download of setup data or MPL program

Write protect last ½ of EEPROM after download – when this option is selected the last half of the EEPROM is write protected after the download of setup data or MPL program

Write protect entire EEPROM after download – when this option is selected the entire EEPROM is write protected after the download of setup data or MPL program

Scan Network. Use this command to detect online drives/motors, members of a CAN network. The drives/motors detect are listed in the Output View along with their axis ID and firmware version.

3.1.6. View Menu

Project. Use this command to visualize the “**Project Window**”

Command Interpreter. Use this command to visualize the “**Command Interpreter Window**”

Logger. Use this command to visualize the “**Logger Window**”

Multi-Axis Logger. Use this command to visualize the “**Multi-Axis Logger Window**”

Control Panel. Use this command to show/hide the “**Control Panel**” windows defined for the selected application. By default there are 3 control panels defined. Check the windows you want to show/hide from the list.

- **1_Motion Status**
- **2_Drive IO**
- **3_Drive Status**

Memory The command opens the Memory Window, within you can view/modify the drive/motor memory contents.

Remark: *This is a feature is a very low level function, it is **NOT** recommended to modify memory contents without a deep knowledge of the use made by the ElectroCraft drive of each memory location you intend to modify.*

Output. Use this command to visualize the “**Output Window**”. From Communication | Show Info In Output View menu you select what information is presented in the window.

Refresh The command refreshes the content of Memory window. It’s available only when the Memory window is active.

Toolbar. Use this command to hide/show the MotionPRO Developer toolbar.

Status bar. Use this command to hide/show the status bar from the bottom of MotionPRO Developer window.

View graph plot. Previously saved plots from Logger or during controllers tuning can be opened from this menu command.

3.1.7. Logger

Variables... This menu command opens the dialogue from where you manage the plotted variables.

Plot Setup... This menu command allows you to select and group on specific graphic subplots the variables which will be stored during the motion execution through the data logging procedure.

Plot Options... This menu command allows you to set the graphical parameters of all the variables selected to be plotted in any of the four subplots of Logger View, as colors, line width and pattern, background, axes colors, grid options and measurement units.

Arrange | From this menu entry you can define the position of the subplots on the Logger View. The command is effective if more than one subplot are defined

Auto: use a default disposal of the subplots, depending on their number (2, 3 or 4).

Horizontal: the plot window is divided in horizontal regions for sub-plotting. The subplots are displayed in a row, from left to right, on the graphic window.

Vertical: the plot window is divided in vertical regions for sub-plotting. The subplots are displayed one below the other

Zoom | This menu commands allows you to select fixed zoom areas of the *first subplot* on the Logger View

Zoom In: zoom-in the graphical image of the first subplot.

Zoom Prev: zoom-out one step the graphical image of the first subplot.

Zoom Out: zoom-out back to the initial graphical image of the first subplot.

Start Use this menu command to start storing data onto the **drive/motor** memory.

Upload Data. Use this menu command to get the data from the **drive/motor** memory and display them in the logger window.

Stop Data Upload. Use this menu command to stop the logged data uploading process

Import... Use this menu command in order to load a pre-defined logger configuration into a special format file. Thus, all logger settings, including selected variables, pre-defined sub-plots contents, and other preferences (colors, etc), can be loaded, replacing the actual logger settings.

Export...Use this menu command in order to save the actual logger configuration into a special format file. Thus, all logger settings, including selected variables, pre-defined sub-plots contents, and other preferences (colors, etc), can be saved on that file

Save graph as... This menu command allows saving the selected graph into ElectroCraft plot files format, with extension **TPT**. A dialog is opened which ask the user to indicate the name of the file. The saved file may be opened using the menu command **View | View Plot Graph...**

Export to WMF. This menu command will be used to save the actual graphic window contents to a file on the system disk, into a standard format, the Windows Metafile Format (or WMF). A special dialog is opened, similar to the **Export...** one, which asks the user to indicate the name of the metafile file (its default extension is **“.WMF”**). The saved file may then be imported in other Windows applications that have adequate graphic filters and recognize the metafile format. Thus, the graphics may be included in other documents; more text may be added to the plots, colors and other features may be changed

Export to ASCII. This menu command will be used to save the actual values of *all the uploaded variables values*, on a file on the system disk, into a standard ASCII text format. A special dialog is opened, similar to the **Export...** one, which asks you to indicate the name of the ASCII file (its default extension is **“.txt”**). The saved file may then be examined, and also read and imported in different other programs as Excel, Word, etc.

Print... The command opens the dialogue from where you can print the selected plot

Print Preview With this command you can preview the plot before printing

Print Setup...The command opens a dialogue where you configure the printer used by MotionPRO Developer

3.1.8. Control Panel

Start. Use this menu command to start the control panels of an application. From this moment, all the contents of all the objects contained in the visible control panels of that application will be updated and displayed on the screen.

Stop. Use this menu command in order to stop the update of information on the control panels of the application. Note that this command will delete all the information associated to that control panel

Customize. Use this menu command in order to be able to customize a control panel. A special toolbar will be displayed, containing all the possible objects, which can be added in a control panel. You'll be able to add, remove and parameterize all the objects of a control panel. Note that during the parameterization stage, all the control panels are stopped.

Rename... A name must be given to a control panel at the moment of its loading from an external file, or at its creation. This name is displayed in the window bar of the panel. You can change this name using this menu command.

Export to File. Use this menu command in order to save a defined control panel on an external file. This will allow you to load and use this control panel in a different application.

Edit Active Item... Choosing this menu command will open its specific parameterization dialog. This dialogue is automatically opened when a new object is defined.

Align to | Use this menu command in order to align all the objects which are selected, at left, right, top or bottom. Note that the reference position is taken from the LAST selected object in the currently selected objects.

Left - align the selected objects along their left side

Right - align the selected objects along their right side

Top - align the selected objects along their top edges

Bottom - align the selected objects along their bottom edges

Space evenly | Use this menu command in order to equally space all the objects which are selected, horizontally (across) or vertically (down). Note that the reference position is taken from the selected objects placed in the extremes of the currently selected objects.

Across. Use this menu command to space objects evenly between the leftmost and the rightmost control selected.

Down. Use this menu command to space objects evenly between the topmost and the bottommost object selected.

Make same | You can manually resize an object by using the specific resize mouse cursors and the mouse left-button. If more objects are selected, this menu commands allows you to make the same width, height or size (both width and height) for all these objects. Note that there are some limits when trying to resize some of the objects. Note that the reference size is taken from the LAST selected object in the currently selected objects.

Width - size objects with the same width as the dominant object;

Height - size objects with the same height as the dominant object;

Size - size objects with both the same height and the same width as the dominant object.

Send to back. Use this menu command to send to back the selected items.

Send to front. Use this menu command to send to front the selected items.

Add Control Panel... Use this menu command to define a new control panel.

Add Control Panel from file... Use this menu command to add into your current application control panels defined in another application (e.g. associated with another setup file).

Delete Control Panel | Use this menu command to delete a control panel.

- **1_Motion Status**
- **2_Drive IO**
- **3_Drive Status**
- Other Control Panels created

3.1.9. Help

Help Topics

Getting Started

About MotionPRO Developer... – The menu command opens a dialogue with information about MotionPRO Developer version

Enter registration info...


Check Updates – From this menu command you launch the PRO Update utility.


3.2. Toolbar


The buttons in the toolbar represent commonly used MotionPRO Developer commands.





 **New.** Use this icon to create a new project. The "**New Project**" menu dialog will open.


 **Open.** Use this icon to open an already defined motion project that was previously saved and closed. The "**Open MotionPRO Developer project**" dialogue will open, allowing you to select a project name.


 **Save.** In order to save the changes done to the current project use this icon. If the current project has not been named a "**Save Project**" window will open, allowing you to create a new folder where the project will be saved.


 **Print.** Use this icon in order to print the motion sequences from active window select some printing parameters such as the printer, the paper size and orientation.


 **View Project.** Use this icon to visualize the "**Project Window**"


 **Command Interpreter.** Use this icon to visualize the "**Command Interpreter**"


 **View Logger.** Use this icon to visualize the "**Logger**" window

 **View Multi-Axis Logger.** Use this icon to visualize the "**Multi-Axis Logger**" window


 **View Control Panel.** Use this icon to visualize the "**Control Panel**" windows. Check the windows you want to be open from the existing list.

 **Edit/View Setup.** Using this menu command will open the drive setup window and motor setup window. In these windows you can view or change the drive and motor settings.


 **Import Setup from Drive/Motor.** Use this icon to upload an existing setup from the drive/motor.


 **Download Setup to Drive/Motor.** Use this icon to download the setup to drive/motor.


 **Run.** Use this icon to download and run your application.


 **Axis On.** Enable PWM signals.

 **Axis Off.** Disable PWM signals.


 **Reset Active Drive/Motor.** Send a reset command to the selected drive/motor.


 **Start Logger.** Use this toolbar icon to start storing data onto the **drive/motor** memory. The button is active only when the logger window is active.


 **Upload Logger.** Use this button to upload data from drive/motor. The button is active only when the logger is started.

 **Stop Logger Upload.** Use this button to stop the upload of data from drive/motor. The button is active only when data from the drive is uploaded.

 **Start Control Panel.** Use this button to start the selected control panels.

 **Stop Control Panel.** Use this button to stop the selected control panels. The button becomes active when the control panels are started.

 **Refresh Communication.** Use this button to reestablish communication with the drive/motor.

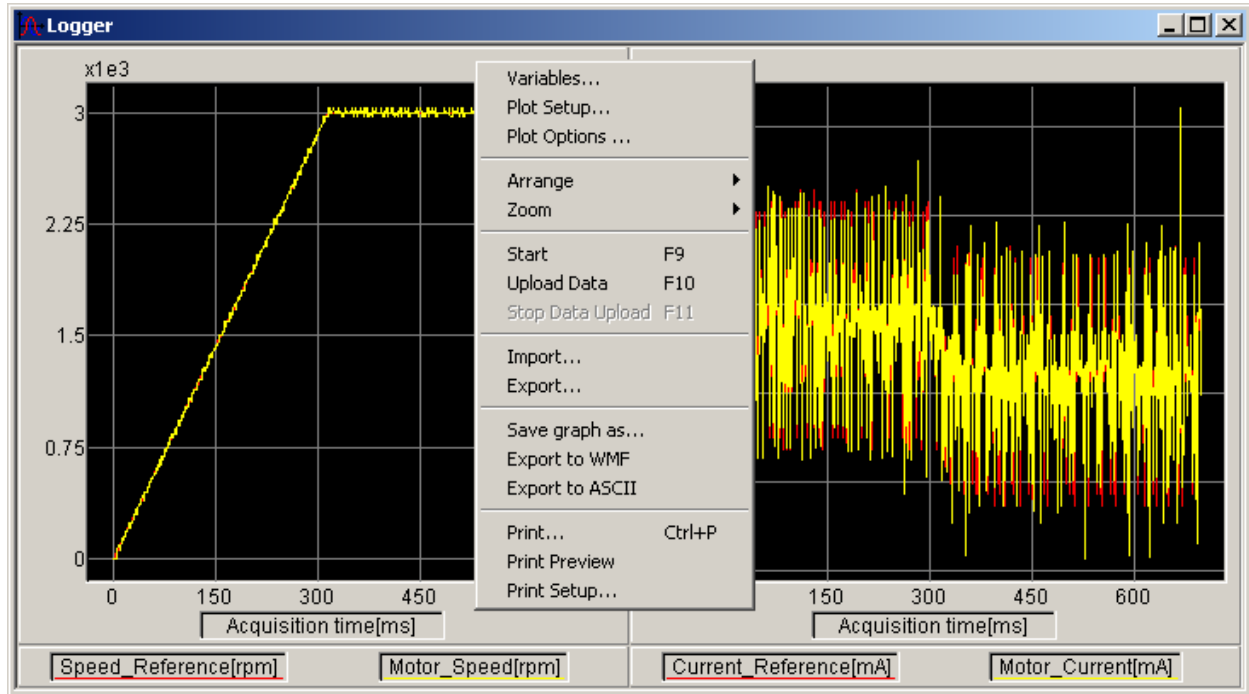
 **Help.** Opens the help page associated with the active window.


4. Evaluation Tools

4.1. Data Logger

4.1.1. Data Logger

The **Data Logger** is an advanced graphical analysis tool, allowing you to do data acquisitions on any variable of your drive / motor and plot the results.



In order to set up / manage the data logger module, simply select the **View | Logger** menu command (alternatively, use the associated toolbar icon ).


Once the **Logger** window is opened, you have access to its associated menu by clicking on the right mouse button when positioned in the logger window. This opens the **Logger pop-up menu**. This menu has the following menu sub-commands:

[Variables](#) / [Plot Setup...](#) / [Plot Options...](#) / [Arrange](#) / [Zoom](#) / [Start](#) / [Upload Data](#) / [Stop Data Upload](#) / [Import...](#) / [Export...](#) / [Export to WMF](#) / [Export to ASCII](#) / [Print...](#) / [Print Preview](#) / [Print Setup...](#)

Depending on the state of the **Logger**, some of these menu sub-commands will be enabled or not, hence you can execute only the allowed operations for a given situation.


4.1.2. Data Logger - Start

Start the logger


Use the **Logger | Start** menu command (or the associated toolbar icon ) to start a data acquisition for the selected variables. Data is saved in the **drive/motor** RAM memory. In MotionPRO Developer, if the logger window is opened, the data acquisition is started automatically when you press the **Run** button.

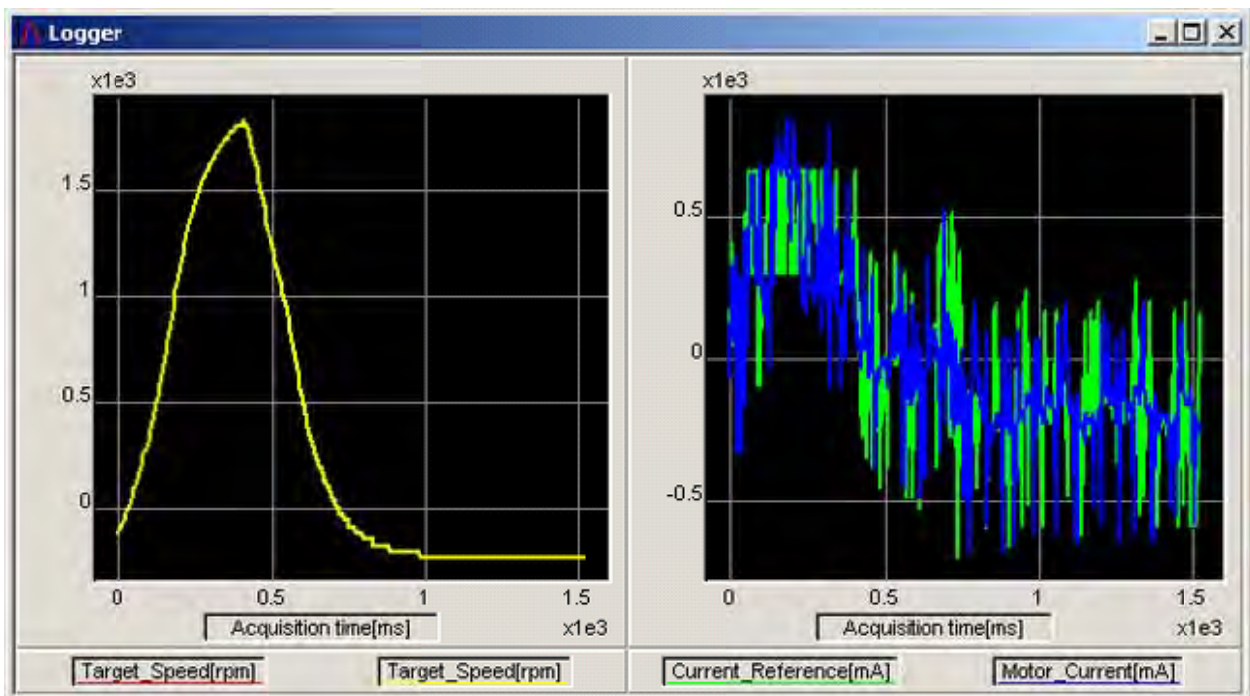
Each time when you execute a **Logger | Start** command, the data acquisition is restarted and will overwrite the previously stored data. Once the buffer is full, the data storage process is stopped.

Upload Data

Use the **Logger | Upload Data** menu command (or the associated toolbar icon ) to get the data from the **drive/motor** memory and display them in the logger window.

Stop Data Upload

Use the **Logger | Stop Data Upload** menu command (or the associated toolbar icon ) to stop the logged data uploading process

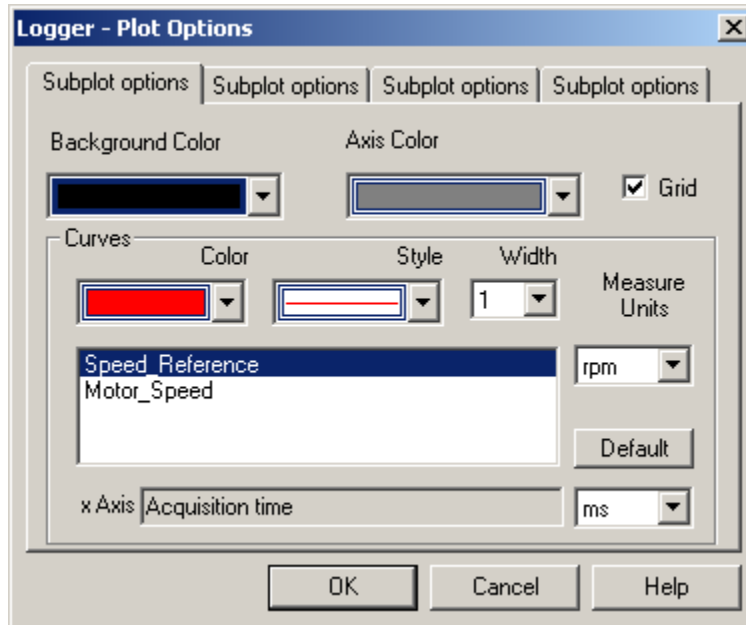


See also:

[Data Logger Utility](#)

4.1.3. Data Logger - Plot Options

The **Logger | Plot Options...** menu command allows you to set the graphical parameters of all the variables selected to be plotted in any of the four subplots of Logger View, as colors, line width and pattern, background, axes colors, grid options and measurement units.



The list of variables that will be stored during the data logging must be defined using the [Logger | Variables...](#) menu command. Once this list is defined, you may use the [Logger | Plot Setup](#) dialog in order to select the corresponding variables and distribute them on the graphics subplots for further visualization.

With the plot variables selected in the [Logger | Plot Setup](#) dialog, you may open the **Logger Plot Options** dialog in order to examine / modify the predefined graphic attributes associated to the curves, axes, etc.

The dialog contains the complete list of the variables selected to be stored for each of the possible four subplots that may be defined. For each subplot, any variable to be plotted on it may be selected from the list grouped under the title **Curves**.

You may switch between the subplots using the corresponding tabs associated to each subplot. By default, each subplot tab is named as **Subplot options** if no name was given to the curve. Otherwise, if that name was defined, it is used as the tab name. (You may freely define each of the subplot names in the [Logger | Plot Setup](#) dialog).

For each subplot, you may select any of the variables from the **Curves** list. Once a variable is selected (outlined) in the list, its graphical attributes are displayed and may be examined and/or modified by you.

Thus, you may modify:

- The curve color, using the **Color** drop-down list of available colors (up to 28 colors may be used);
- The curve style, using the **Style** drop-down list of available line styles (up to 5 line styles may be used);
- The curve width, using the **Width** drop-down list of available line widths (1 to 4 line widths may be used)

You can change in the **Measure Units** drop-down list the units in which to display the variables stored.

The background color may be defined for each subplot, using the **Background Color** drop-down list (up to 28 colors may be used).

The axis color may be defined for each subplot, using the **Axis Color** drop-down list (up to 28 colors may be used).

The grid option for each subplot may be set/reset using the **Grid** check button.

You may also define the X-axis label and measurement unit, by editing the **X-Axis** edit control field and respectively, by selecting the measurement unit from the associated drop-down list of possible units.

Use the **Default** button to reset all the selected measurement units for the curves.

Use the **OK** button to effectively apply the defined settings and exit back to the Logger View, by closing the **Logger - Plot Options** dialog.

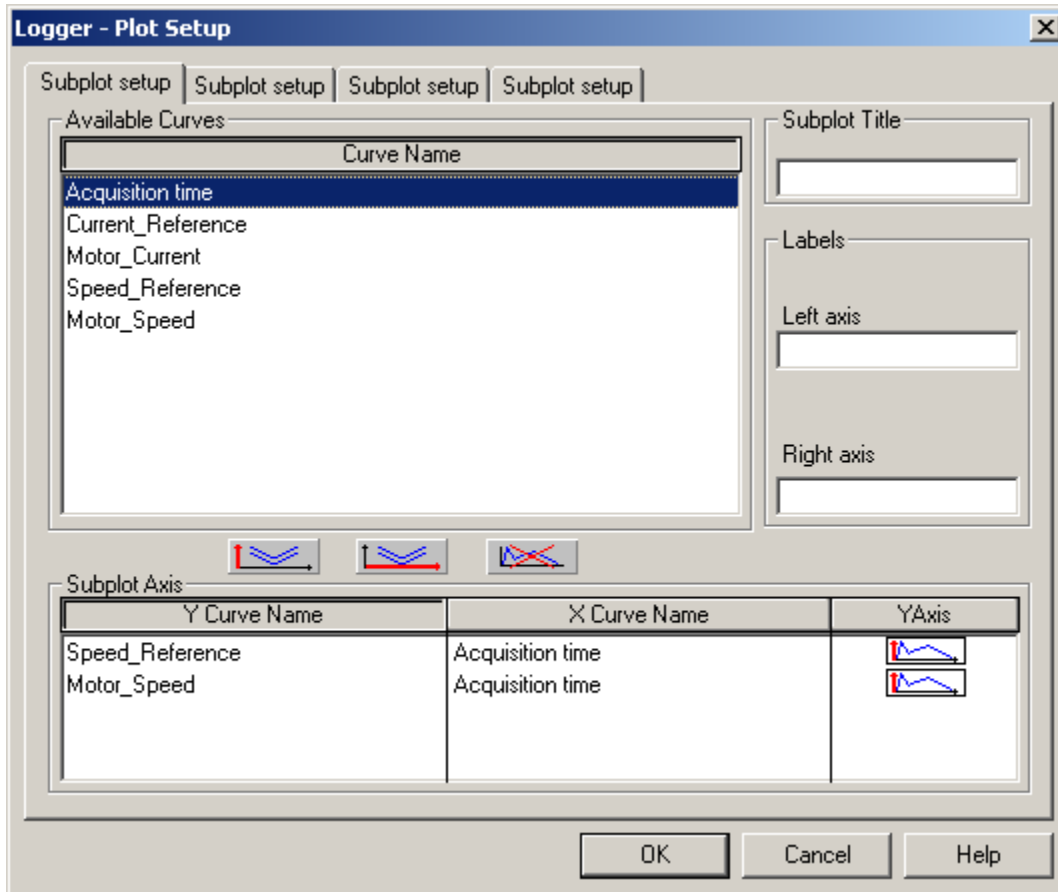
Use the **Cancel** button to cancel all the defined settings and exit back to the Logger View, by closing the **Logger - Plot Options** dialog.

See also:

[Data Logger Utility](#)

4.1.4. Data Logger - Plot Setup

The **Logger | Plot Setup...** menu command allows you to select and group on specific graphic subplots the variables which will be stored during the motion execution through the data logging procedure. Up to four subplots may be defined.





The list of variables, which will be stored during the data logging, must be defined using the [Logger | Variables...](#) menu command. Once this list defined, you may accede the **Logger - Plot Setup** dialog in order to select the corresponding variables and distribute them on the graphics subplots for visualization.

The dialog contains the complete list of the curves selected to be stored, in the top of it. For each subplot, the curves to be plotted on it may be chosen from the complete list of stored variables, grouped under the title **Available Curves**.


You may switch between the subplots using the corresponding tabs associated to each subplot. By default, each subplot tab is named as **Subplot setup**. You may freely define each of the subplot names. The **Subplot Title** edit box contains the actual (if it was defined) subplot title. You may define or modify it at any time by editing this edit control.


Each subplot has an associated list of the selected curves to be displayed on that subplot, grouped under the name of **Subplot Axis**. The list may be updated by you by adding to / removing from it curves from the **Available Curves** list.



A variable may be added to the subplot curves list by selecting it in the **Available Curves** list, with a left-button mouse click (the selected variable becomes outlined), and by pressing on one of the **Add to the**


list buttons: press  to add it as the Y axis curve, or press the  button to add it as the X axis curve.

When adding a new variable to the subplot curves list, by having at the same time a selected (outlined) curve in the **Subplot Axis** list, the added variable will automatically replace the previous one from the **Subplot Axis** list, corresponding to the Y or X axis selected to be replaced.

A curve may be removed from the subplot curves list by selecting it in the **Subplot Axis** list, with a right-button mouse click (the selected curve becomes outlined), and by pressing the **Remove from the list** button .

Always, the variable Acquisition Time exists in the **Available Curves** list. Usually, you will select some other variable to be added to the **Subplot Axis** list. When the first variable is selected and added to that list as a Y axis curve, the program automatically inserts by default, as the X-axis, the Acquisition Time variable .

The variables may be related to the left or to the right vertical axis of the subplot. Usually, the variables are introduced as related to the left vertical axis . If you want to change this setting to the right vertical axis, you need to double-click the vertical axis symbol, which will commute to the right vertical axis symbol . (A similar double-click on this symbol will reverse again the vertical axis to the left one).

If you want to use a special X-axis coordinate, different that the time variable (in order to visualize the dependence between two variables), you must select the desired X-axis variable in the **Available Curves** list, and add it to the **Subplot Axis** list using the Add to X axis button . The newly selected variable will replace the time variable as the X-axis coordinate.

By default, the vertical axes do not have a name. You may freely define for each of the subplots names for the two vertical axes. The **Labels** group contains the actual (if it was defined) subplot **Left axis** and **Right axis** names. You may define or modify them at any time by editing the corresponding edit control.

Use the **OK** button to effectively apply the defined settings and exit back to the Logger View, by closing the **Logger - Plot Setup** dialog.

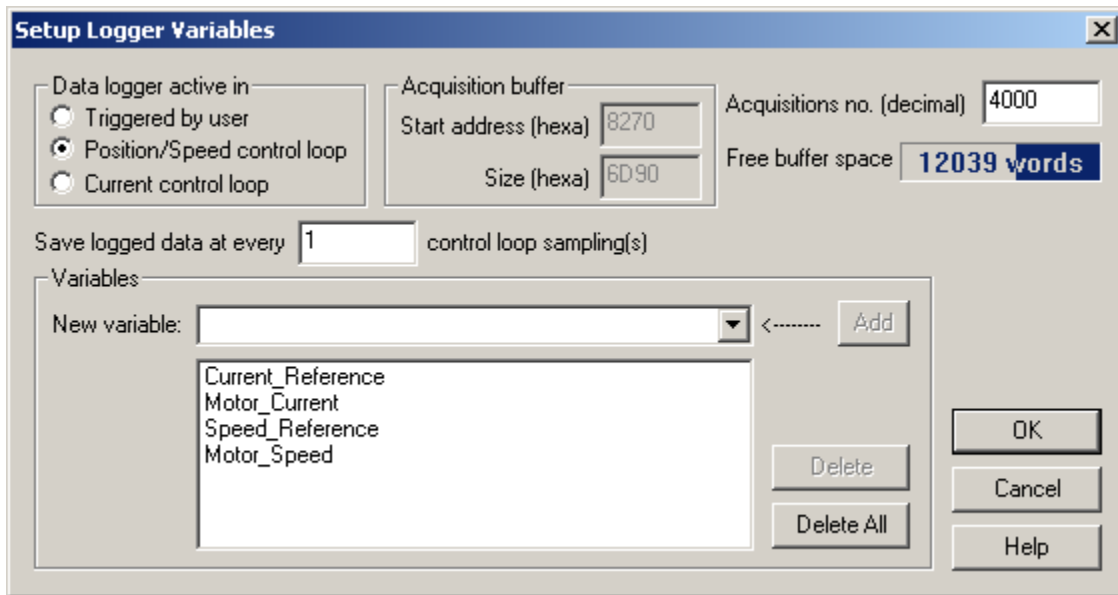
Use the **Cancel** button to cancel all the defined settings and exit back to the Logger View, by closing the **Logger - Plot Setup** dialog.

See also:

[Data Logger Utility](#)

4.1.5. Data Logger - Variables

Use this dialogue to select the variables to acquire for plotting. You can select any variable. All the variables selected will be saved into the drive/motor memory at some predefined moments. The total number of acquisitions points (“**Acquisitions No**” value) depends on the drive/motor memory available for data logging storage. Also in this dialog you can select the data logging moments. The data logging process may be: **triggered by user**, done in **position/speed control loop (default)**, or into the current loop. You can select to acquire data at each sampling loop or from n to n samplings.



From this dialogue you can:

- Select where to perform the data logging (“Data logger active in” box):
 - **Triggered by user** (available only for some products)
 - in the **Position/Speed control loop** (slow sampling loop) - default
 - in the **Current control loop** (fast sampling loop). To be used with care, since it can impose to the processor a too big overhead, and thus can affect the behavior of the motion system.
- **See the location and size of the data acquisition buffer** (“**Acquisition Buffer**” box), depending on the memory available in your system. The memory location and size can’t be changed directly, they result from the memory setting dialogue.
- **Define the number of points to store** (“**Acquisition Number**” parameter). Note that the maximum value of this parameter is related to the size of the data acquisition buffer, as well as to the number of variables in the list. The “**Free buffer space**” value can be used to estimate the remaining amount of memory available for data logging.
- **Choose the interval of data logging** (“**Save logged data at every**” box). You can select if the data logging will be performed at each **x** control loops.

-
- **Manage the list of variables to be stored.** You can:
 - add variables to the list. Select the variable from **New Variable** drop-down list and press “**Add**” button.
 - delete variables from the list. Select the variable and use the “**Delete**” button to delete a variable selected in the list, or the “**Delete All**” button to delete all the variables from the list.

See also:

[Data Logger Utility](#)

[Memory Settings](#)

4.1.6. Data Logger - Other Options

Arrange (Auto, Horizontal, vertical)

The **Arrange** menu command allows you to define the position of the subplots on the Logger View. The command is effective if more than one subplot are defined. The following options are available:

- **Auto**: use a default disposal of the subplots, depending on their number (2, 3 or 4).
- **Horizontal**: the plot window is divided in horizontal regions for sub-plotting. The subplots are displayed in a row, from left to right, on the graphic window.
- **Vertical**: the plot window is divided in vertical regions for sub-plotting. The subplots are displayed one below the other.

Zoom (In, Prev, Out)

The **Zoom** menu command allows you to select fixed zoom areas of the selected subplot on the Logger View. The following options are available:

- **In**: zoom-in the graphical image of the first subplot
- **Prev**: zoom-out one step the graphical image of the first subplot
- **Out**: zoom-out back to the initial graphical image of the first subplot

In order to freely zoom any graphical image, you may use the mouse to select a part of the current subplot, allowing the zooming of the selected region. The selection is done by **pressing the left mouse button** and **dragging** the zoom cursor on the display surface (the movement is bound to the area of the subplot). On the release of the mouse button, the selected region is expanded to the dimension of the entire subplot. Successive zooms may be applied to any of the subplots.

Note that, when moving the mouse cursor, you can see, at the bottom of the graphic window, the coordinates on the left and right axes of the current cursor position on the screen. Thus, measurements may be done on the plots. If no region is selected for zooming, the plot is unchanged.

Double-click the left mouse button, with the mouse in the graphical area of a subplot, in order to zoom-out one level back from the currently displayed image.

Import...

Use the "**Logger | Import...**" menu command in order to load a pre-defined logger configuration into a special format file. Thus, all logger settings, including selected variables, pre-defined sub-plots contents, and other preferences (colors, etc), can be loaded, replacing the actual logger settings. This feature is useful in order to easily select a pre-defined preferred logger environment. Such files can be created by saving an already defined logger context, using the "**Logger | Export...**" menu command (see next paragraph).

Note that the command also loads the plotted variables graphs, as existing when the **.igs** file was saved. Use the "**Logger | Upload Data**" menu command to load from the drive the current values for the selected variables.

Export...

Use the " **Logger | Export...**" menu command in order to save the actual logger configuration into a special format file. Thus, all logger settings, including selected variables, pre-defined sub-plots contents, and other preferences (colors, etc), can be saved on that file. This feature is useful in order to save pre-defined preferred logger environments. Such files can be latter-on loaded in order to re-create the same logger context, using the "**Logger | Import...**" menu command (see previous paragraph).

Note that the command also saves the actual plotted variables graphs. Use the "**Logger | Upload Data**" menu command to load from the drive the current values for the selected variables.

Export to ASCII

The **Export to ASCII** menu command will be used to save the actual values of **all the uploaded variables values**, on a file on the system disk, into a standard **ASCII** text format. A special dialog is opened, similar to the **Export...** one, which asks you to indicate the name of the **ASCII** file (its default extension is ".txt"). The saved file may then be examined, and also read and imported in different other programs as Excel, Word, etc. The file will contain:

- on the first line, the number **n** of saved curves, and the number **m** of saved points for each curve, separated by the **TAB** character
- on the next **m** lines, **n** values for the saved curves on each line, separated by **TAB** characters. Each line contains variables values corresponding to a data logger X-axis instant (time sampling)

Export to WMF

The **Export to WMF** menu command will be used to save the actual graphic window contents to a file on the system disk, into a standard format, the Windows Metafile Format (or WMF). A special dialog is opened, similar to the **Export...** one, which asks the user to indicate the name of the metafile file (its default extension is ".WMF"). The saved file may then be imported in other Windows applications that have adequate graphic filters and recognize the metafile format. Thus, the graphics may be included in other documents; more text may be added to the plots, colors and other features may be changed.

Print...

The **Print...** menu command opens a dialogue which allows you to print the represented graphics.

Print Preview

The **Print Preview** menu command opens a new window allowing you to see how the graphics will look after the print.

Print Setup

The **Print Setup** command opens a dialogue with settings related to the printer, paper size and orientation.

See also:

[Data Logger Utility](#)

4.2. Control Panel

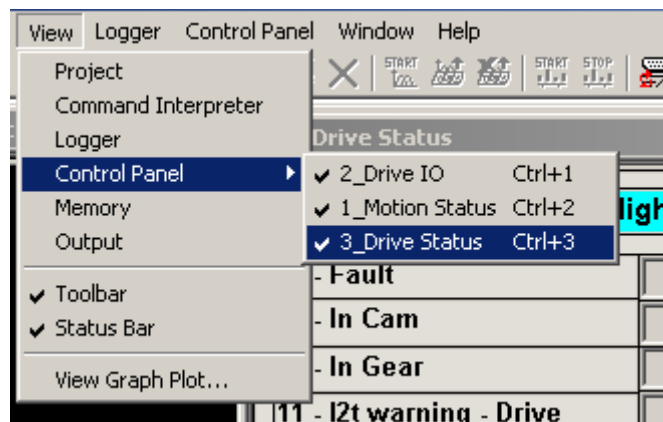
4.2.1. Control Panel

The Control Panel is a tool enabling you to define specific control panels where you can send commands and visualize status variables.

Each ElectroCraft product comes with a set of pre-defined control panels. Using the “**Customize**” option, you can define new control panels or modify the existing ones according with your application specific.

Handling the control panels

Displaying the control panels



Use the “**View | Control Panel**” command menu, in order to see the list of the currently defined control panels. Click on a list item in order to alternatively change its display status (show or hide).

The “Control Panel” menu

All of menu commands are displayed and can be used when you click the right-mouse button, while over a control panel window.

Adding pre-defined control panels to an application

Use the “**Control Panel | Add Control Panel from file ...**” menu command to add into your current application control panels defined in another application (e.g. associated with another setup file).


Adding new control panels to an application

Use the “**Control Panel | Add Control Panel**” menu command to define a new control panel (see next paragraphs how to customize a control panel).

Deleting control panels from an application

Use the “**Control Panel | Delete Control Panel**” menu command in order to delete a control panel.


Activating the control panels of an application

Use the “**Start Control Panel**” button  or the “**Control Panel | Start**” menu command to start the control panels of an application. From this moment, all the contents of all the objects contained in the visible control panels of that application will be updated and displayed on the screen.

Important notes:

1. The **update rate** depends on the communication speed between the PC and your drive/motor and on the number of different variables that must be read from the drive/motor in order to be displayed. In order to keep the update rate high, try to activate only those control panels, which are needed at one moment. Thus you'll avoid over-charging this process and slowing down too much the update rate.
2. The **display rate** of the objects is individually selectable, at their definition (see next paragraphs, the customization procedure of control panels)

Stopping the control panels of an application

Use the “**Stop Control Panel**” button  or the “**Control Panel | Stop**” menu command in order to stop the update of information on the control panels of the application.

Note that this command will delete all the information associated to that control panel. If you want to preserve the control panel, use the “**Control Panel | Export to File...**” menu command, before deleting the control panel. The delete operation acts only at the level of the application, but does not affect the control panels saved on files.

Customizing a control panel

Use the “**Control Panel | Customize**” menu command in order to be able to customize a control panel. A special toolbar will be displayed, containing all the possible objects, which can be added in a control panel. You'll be able to add, remove and parameterize all the objects of a control panel. Note that during the parameterization stage, all the control panels are stopped. See the “**Control Panel Objects**” paragraph for more details regarding the objects, which can be used in a control panel, and their parameter setting.

Renaming a control panel

A name must be given to a control panel at the moment of its loading from an external file, or at its creation. This name is displayed in the window bar of the panel. You can change this name using the “**Control Panel | Rename**” menu command.

Note that this name is valid at the level of the application, and does not affect the name of the control panel file that was eventually used to load the control panel into the application.

Saving a control panel

Use the “**Control Panel | Export to File...**” menu command in order to save a defined control panel on an external file. This will allow you to load and use this control panel in a different application.

Deleting a control panel

Use the “**Control Panel | Delete**” menu command in order to delete the currently selected control panel from the application.

Note that if you previously saved this control panel using the “**Control Panel | Export to File...**” menu command, this command will only delete the control panel from the application, while the saved file will remain unchanged. This will allow you to re-load the control panel again, using the “**Control Panel | Add Control Panel from file ...**” menu command.

In case that you didn't saved the control panel, using the “**Control Panel | Delete**” menu command will completely lose the information defined in it.


Control Panel Customization

A control panel can be freely defined and/or customized by you. Specific basic control panel templates can be saved and included in other MotionPRO Developer applications, as preferred by you. When you start creating a new control panel, using the “**Control Panel | Add Control Panel**” menu command, a new, empty control panel window is opened. At the same time, the specific control panel objects toolbar is also displayed. This toolbar is also displayed when you use the “**Control Panel | Customize**” menu command.

While in the customization mode, all control panels are stopped and can be modified. Use again the “**Control Panel | Customize**” menu command to end the customization and return in the normal operation mode of the control panels.

Several types of visualization or setting objects can be included in a control panel, and positioned / sized as preferred. Each object will be associated to one or more MPL variable(s) (for display-type objects) or MPL parameter (for setting objects). Depending on their types, specific parameters can be defined.

Selecting objects in a control panel. Click on an object from the control panel in order to select it. Press the left-mouse button and drag the mouse in order to select more objects simultaneously. Currently selected object(s) are highlighted, and specific operations can be done related to them (see below). Alternatively, press the **CTRL** key and click the left-button of the mouse in order to select one by one the objects. Note that the **LAST** selected object is **the dominant object**, and alignments and resizing are referred to it.

Editing an object in a control panel. Double-clicking an object, using the **Control Panel | Edit Active Item...** menu command, or the corresponding icon , will open its specific parameterization dialog. This dialogue is automatically opened when a new object is defined. See next paragraph for details related to the parameterization of each type of control panel object.


Deleting an object in a control panel. Use the **DEL** key in order to delete the currently selected objects from a control panel.


Duplicating objects in a control panel. Use the **Control Panel | Duplicate Selected Items** menu command in order to create a copy of all the objects which are selected in that moment. The newly created objects have the same characteristics and parameters as the original ones.


Moving objects in a control panel. Once one or more objects are selected in a control panel, drag them by pressing the mouse left-button and moving the mouse. The objects will move all together, keeping the same distance between them.


Aligning objects in a control panel. Use the “**Control Panel | Align to ...**” menu command or the corresponding icons, in order to align all the objects which are selected, at left, right, top or bottom. Note that the reference position is taken from the LAST selected object in the currently selected objects. To align objects:

- Select the objects you want to align by holding down the CTRL key and clicking the mouse’s left button on the appropriate object window
- Make sure the correct dominant object (the last selected object) is selected.
- The final position of the group of objects depends on the position of the dominant object.


Use **Align Left** button  from the toolbar to align the selected objects along their left side.


Use **Align Right** button  from the toolbar to align the selected objects along their right side.

Use **Align Top** button  from the toolbar to align the selected objects along their top edges.


Use **Align Bottom** button  from the toolbar to align the selected objects along their bottom edges.


Spacing objects in a control panel. Use the **Control Panel | Space evenly...** menu command or the corresponding icons, in order to equally space all the objects which are selected, horizontally (across) or vertically (down). Note that the reference position is taken from the selected objects placed in the extremes of the currently selected objects.


Choose **Space Evenly Across** button  from the local toolbar to space objects evenly between the leftmost and the rightmost control selected.

Choose **Space Evenly Down** button  from the local toolbar to space objects evenly between the topmost and the bottommost object selected.


Resizing objects in a control panel. You can manually resize an object by using the specific resize mouse cursors and the mouse left-button. If more objects are selected, the **Control Panel | Make Same...** menu commands or the corresponding icons, allows you to make the same width, height or size (both width and height) for all these objects. Note that there are some limits when trying to resize some of the objects. Note that the reference size is taken from the LAST selected object in the currently selected objects.


Choose **Make Same Size Width** button  from the local toolbar to size objects with the same width as the dominant object;

Choose **Make Same Size Height** button  from the local toolbar to size objects with the same height as the dominant object;

Choose **Make Same Size Both** button  from the local toolbar to size objects with both the same height and the same width as the dominant object.

Superposing objects in a control panel. In order to create some special visual appearance effects, you can totally or partially superpose objects in a control panel. In this case, it is important to control the relative position of the objects. Use the **Control Panel | Send to Back** or **Control Panel | Bring to Front** menu commands or the corresponding icons.

Choose **Send to Back** button  from the local toolbar to send to back the selected items.

Choose **Send to Front** button  from the local toolbar to send to front the selected items.

Control Panel Objects

This section contains the description of the different objects that can be defined in a control panel. In the customization mode, you can freely add / remove objects to a control panel. Simply drag and drop an object from the toolbar containing the object symbols, and place it on the control panel area. Objects are user-resizable. For each of these objects, as already mentioned, there is associated a **variable / parameter, I/O port number or data memory location** or an **expression** can be defined, to be evaluated before being displayed. They can be selected at the moment when the control panel is defined.

A data memory location must be specified with the following format: *type@address* where *type* represents the data type supported by MPL: integer, long or fixed and *address* is the memory location address expressed in hexadecimal form.

Example: *fixed@0x0903* the memory contents from addresses 0x0903 and 0x904 are interpreted as a fixed data.

The **Expression** can be built using following operators:

“+”	- addition
“-“	- subtraction
“*”	- multiply
“/”	- division
“^”	- power
“(“ and “)”	- parentheses

The operands used for editing of an **Expression** are:

variable_name [unit]
variable_name
type@address
number

Example: $100.5 + 5 * (Position_Command [rot] - 10.0)$

Note that for all the objects used to display the value of a variable, the variable can be selected from the current list of variables.

In the “**Axis ID**” edit field of each object, you can select the Axis ID of the axis from where the variable will be read/set. This will allow you to visualize in one control panel variables from different drives / motors

connected into a network. The default axis ID is **as set in Comm Setup** e.g. the axis ID set in **Communication |Setup** dialogue.

Select the measurement unit in “**Unit**” section, corresponding to the selected variable type.

Adjust the number of decimals by choosing in “**Precision**” section one of the values from the scroll list, for a convenient representation depending on values range. Note that if the selected measurement unit is “**IU**” (Internal Units), the decimals parameter is not used. Also for **IU** representations, hexadecimal format can be selected.

The title of the object window (displayed in the object window title bar) can be specified in **Title** section. By default, if you don't specify a title, the variable name will be assumed instead.

For display purposes, display time intervals can be selected in the “**Read value at every**” field.

[Value Object](#)

[Scope Object](#)

[Dual Chanel Scope Object](#)

[Y\(X\) Dual Chanel Scope Object](#)

[Gauge Object](#)

[Cursor Object](#)

[Input Port Viewer Object](#)

[Output Port Setting Tool Object](#)

[Viewer of a Bit of a Variable Object](#)

[User Defined MPL Sequence Object](#)

[Label Object](#)

4.2.2. Control Panel - Show Value



It's used to visualize the value of one MPL variable or data memory contents.

Axis ID: as set in Comm Setup

Variable: Motor_Position Unit: rot

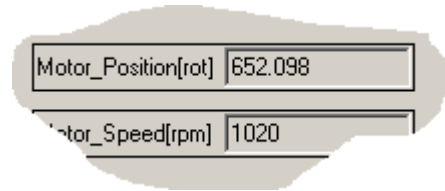
Expression:

Precision: 3 decimals Hexadecimal

Title: Motor_Position[rot]

Read value at every: 0 ms Border

Buttons: OK, Cancel, Help



In the **Variable** field select the desired one from the current list of variables or insert a memory location.

A data memory location must be specified with the following format: *type@address* where *type* represents a data type supported by MPL: integer, long or fixed and *address* is the memory location address expressed in hexadecimal form.

Example: *fixed@0x0903* the memory contents from addresses 0x0903 and 0x904 are interpreted as a fixed data.

Select **Expression** to define a formula to be evaluated before being displayed. They can be selected at the moment when the control panel is defined.

The **Expression** can be built using following operators:

- “+” - addition
- “-” - subtraction
- “*” - multiply
- “/” - division
- “^” - power
- “(“ and “)” - parentheses

The operands used for editing of an **expression** are:

variable_name [unit]

variable_name

number

Example: $100.5 + 5 * (Position_Command [rot] - 10.0)$

In the **Axis ID** edit field of each object, you can select the Axis ID of the axis from where the variable will be read/set. This will allow you to visualize in one control panel variables from different drives / motors connected into a network. The default Axis ID is “**as set in Comm Setup**” e.g. the Axis ID selected in the **Communicate with** field from **Communication | Setup** dialogue.

Select the measurement unit in “**Unit**” section, corresponding to the selected variable type.

Adjust the number of decimals by choosing in “**Precision**” section one of the values from the scroll list, for a convenient representation depending on values range. Note that if the selected measurement unit is “**IU**” (Internal Units), the decimals parameter is not used. Also for **IU** representations, hexadecimal format can be selected.

The title of the object window (displayed in the object window title bar) can be specified in **Title** section. By default, if you don't specify a title, the variable name will be assumed instead.

For display purposes, display time intervals can be selected in the “**Read value at every**” field.

See also:

[Control Panel Utility](#)

4.2.3. Control Panel - Scope



It's used to visualize one variable. Note that, because the update rate of these values is somehow limited (depending on the communication speed between the PC and the drive, and on the functionality of the Windows environment), the evolution of fast changing variables cannot be correctly visualized. You cannot visualize AC currents or voltages, for example. Use this tool for slow varying or steady state regime analysis. Otherwise, use the **Logger** utility.

Scope

Axis ID: as set in Comm Setup

Variable: Motor_Speed Unit: rpm

Expression:

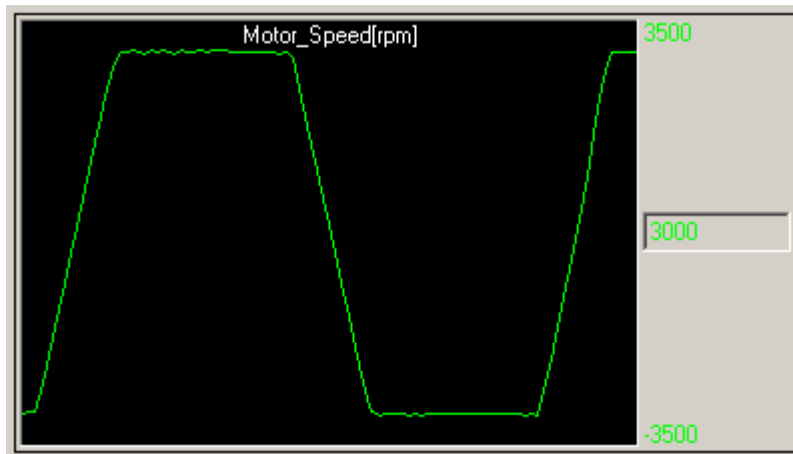
Precision: 3 decimals Hexadecimal

Min: -3500 Max: 3500

Title: Motor_Speed[rpm] Color: [Green]

Time period: 0 seconds Border

OK Cancel Help



In the **Variable** field select the desired one from the current list of variables or insert a memory location.

A data memory location must be specified with the following format: *type@address* where *type* represents the data type supported by MPL: integer, long or fixed and *address* is the memory location address expressed in hexadecimal form.

Example: *fixed@0x0903* the memory contents from addresses 0x0903 and 0x904 are interpreted as a fixed data.

Select **Expression** to define a formula to be evaluated before being displayed. They can be selected at the moment when the control panel is defined.

The **Expression** can be built using following operators:

“+”	- addition
“-“	- subtraction
“*“	- multiply
“/“	- division
“^“	- power
“(“ and “)”	- parentheses

The operands used for editing of an **expression** are:

variable_name [unit]

variable_name

number

Example: $100.5 + 5 * (Position_Command [rot] - 10.0)$

In the **Axis ID** edit field of each object, you can select the Axis ID of the axis from where the variable will be read/set. This will allow you to visualize in one control panel variables from different drives / motors connected into a network. The default Axis ID is “**as set in Comm Setup**” e.g. the Axis ID selected in the **Communicate with** field from **Communication | Setup** dialogue.

Select the measurement unit in “**Unit**” section, corresponding to the selected variable type.

Adjust the number of decimals by choosing in “**Precision**” section one of the values from the scroll list, for a convenient representation depending on values range. Note that if the selected measurement unit is “**IU**” (Internal Units), the decimals parameter is not used. Also for **IU** representations, hexadecimal format can be selected.

The title of the object window (displayed in the object window title bar) can be specified in **Title** section. By default, if you don’t specify a title, the variable name will be assumed instead.

For display purposes, display time period can be set in the “**Time period**” field.

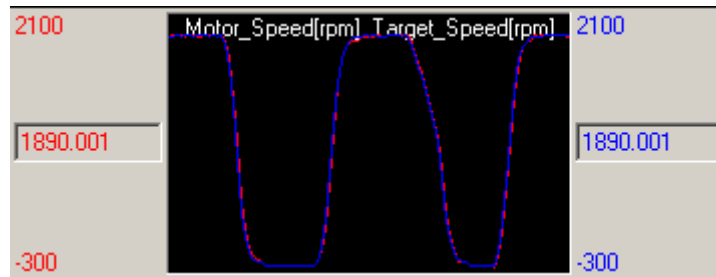
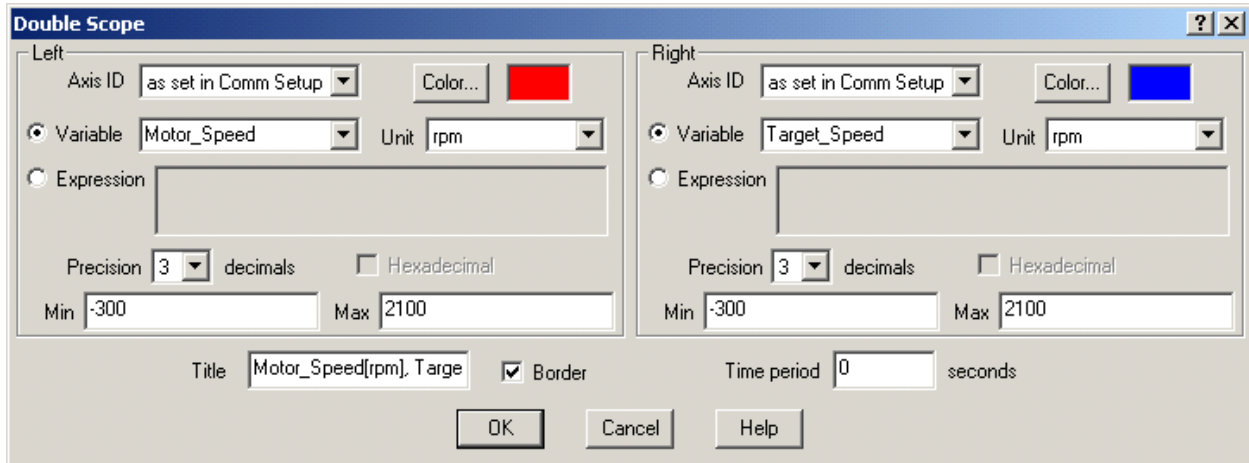
See also:

[The Control Panel Utility](#)

4.2.4. Control Panel - Double Scope



It's used to visualize two variables on the same area. Note that, because the update rate of these values is somehow limited (depending on the communication speed between the PC and the axis, and on the functionality of the Windows environment), the evolution of fast changing variables cannot be correctly visualized. You cannot visualize AC currents or voltages, for example. Use this tool for slow varying or steady state regime analysis. Otherwise, use the **Logger** utility.



In the **Variable** field select the desired one from the current list of variables or insert a memory location.

A data memory location must be specified with the following format: *type@address* where *type* represents the data type supported by MPL: integer, long or fixed and *address* is the memory location address expressed in hexadecimal form.

Example: *fixed@0x0903* the memory contents from addresses 0x0903 and 0x904 are interpreted as a fixed data.

Select **Expression** to define a formula to be evaluated before being displayed. They can be selected at the moment when the control panel is defined.

The **Expression** can be built using following operators:

- “+” - addition
- “-” - subtraction
- “*” - multiply

“/” - division
“^” - power
“(“ and “)” - parentheses

The operands used for editing of an **expression** are:

variable_name [unit]

variable_name

number

Example: $100.5 + 5 * (Position_Command [rot] - 10.0)$

In the **Axis ID** edit field of each object, you can select the Axis ID of the axis from where the variable will be read/set. This will allow you to visualize in one control panel variables from different drives / motors connected into a network. The default Axis ID is “**as set in Comm Setup**” e.g. the Axis ID selected in the **Communicate with** field from **Communication | Setup** dialogue.

Select the measurement unit in “**Unit**” section, corresponding to the selected variable type.

Adjust the number of decimals by choosing in “**Precision**” section one of the values from the scroll list, for a convenient representation depending on values range. Note that if the selected measurement unit is “**IU**” (Internal Units), the decimals parameter is not used. Also for **IU** representations, hexadecimal format can be selected.

The title of the object window (displayed in the object window title bar) can be specified in **Title** section. By default, if you don't specify a title, the variable name will be assumed instead.

For display purposes, display time period can be set in the “**Time period**” field.

See also:

[The Control Panel Utility](#)

4.2.5. Control Panel - Y(X) Scope Object



This object is similar to the Dual-channel scope (**Double Scope Object**) except that you visualize one variable as function of another variable on the same area.

In the **Variable** field select the desired one from the current list of variables or insert a memory location.

A data memory location must be specified with the following format: *type@address* where *type* represents the data type supported by MPL: integer, long or fixed and *address* is the memory location address expressed in hexadecimal form.

Example: *fixed@0x0903* the memory contents from addresses 0x0903 and 0x904 are interpreted as a fixed data.

Select **Expression** to define a formula to be evaluated before being displayed. They can be selected at the moment when the control panel is defined.

The **Expression** can be built using following operators:

- “+” - addition
- “-” - subtraction
- “*” - multiply
- “/” - division
- “^” - power
- “(“ and “)” - parentheses

The operands used for editing of an **expression** are:

variable_name [unit]
variable_name
number

Example: $100.5 + 5 * (Position_Command [rot] - 10.0)$

In the **Axis ID** edit field of each object, you can select the Axis ID of the axis from where the variable will be read/set. This will allow you to visualize in one control panel variables from different drives / motors connected into a network. The default Axis ID is “**as set in Comm Setup**” e.g. the Axis ID selected in the **Communicate with** field from **Communication | Setup** dialogue.

Select the measurement unit in “**Unit**” section, corresponding to the selected variable type.

Adjust the number of decimals by choosing in “**Precision**” section one of the values from the scroll list, for a convenient representation depending on values range. Note that if the selected measurement unit is “**IU**” (Internal Units), the decimals parameter is not used. Also for **IU** representations, hexadecimal format can be selected.

The title of the object window (displayed in the object window title bar) can be specified in **Title** section. By default, if you don't specify a title, the variable name will be assumed instead.

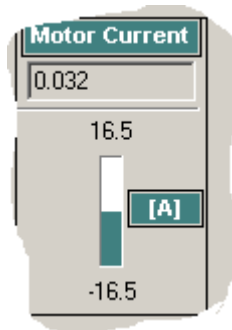
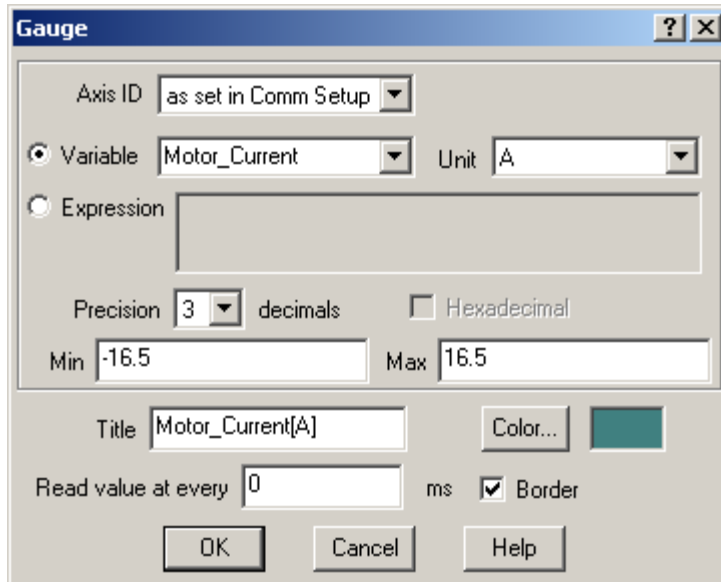
For display purposes, display time period can be set in the “**Time period**” field.

See also:

[The Control Panel Utility](#)

4.2.6. Control Panel - Gauge

Gauge is used to indicate the value of a variable and its variation in time



Horizontal gauge: used to indicate the value of a variable and its variation in time. Disposed on horizontal direction.



Vertical gauge: used to indicate the value of a variable and its variation in time. Disposed on vertical direction.

In the **Variable** field select the desired one from the current list of variables or insert a memory location.

A data memory location must be specified with the following format: *type@address* where *type* represents the data type supported by MPL: integer, long or fixed and *address* is the memory location address expressed in hexadecimal form.

Example: *fixed@0x0903* the memory contents from addresses 0x0903 and 0x904 are interpreted as a fixed data.

Select **Expression** to define a formula to be evaluated before being displayed. They can be selected at the moment when the control panel is defined.

The **Expression** can be built using following operators:

“+”	- addition
“-“	- subtraction
“*”	- multiply
“/”	- division
“^”	- power
“(“ and “)”	- parentheses

The operands used for editing of an **expression** are:

variable_name [unit]

variable_name

number

Example: $100.5 + 5 * (\text{Position_Command [rot]} - 10.0)$

In the **Axis ID** edit field of each object, you can select the Axis ID of the axis from where the variable will be read/set. This will allow you to visualize in one control panel variables from different drives / motors connected into a network. The default Axis ID is “**as set in Comm Setup**” e.g. the Axis ID selected in the **Communicate with** field from **Communication | Setup** dialogue.

Select the measurement unit in “**Unit**” section, corresponding to the selected variable type.

Adjust the number of decimals by choosing in “**Precision**” section one of the values from the scroll list, for a convenient representation depending on values range. Note that if the selected measurement unit is “**IU**” (Internal Units), the decimals parameter is not used. Also for **IU** representations, hexadecimal format can be selected.

In the **Min** and **Max** edit field you can specify the minimum and the maximum values you wish to visualize.

The title of the object window (displayed in the object window title bar) can be specified in **Title** section. By default, if you don’t specify a title, the variable name will be assumed instead.

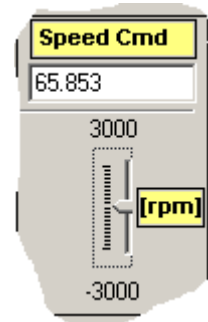
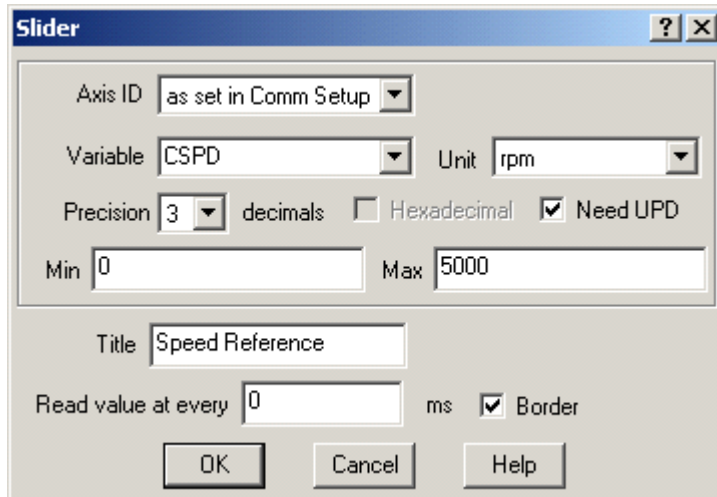
For display purposes, display time intervals can be selected in the “**Read value at every**” field.

See also:

[The Control Panel Utility](#)

4.2.7. Control Panel - Slider

This cursor object is used to change the value of a parameter.



Horizontal cursor: is disposed on horizontal direction.



Vertical cursor: is disposed on vertical direction.

In the **Variable** field select the desired one from the current list of variables or insert a memory location.

A data memory location must be specified with the following format: *type@address* where *type* represents a data type supported by MPL: int, long or fixed and *address* is the memory location address expressed in hexadecimal form. Type is optional if it's not specified data is interpreted as integer.

Example: *fixed@0x0903* the memory contents from addresses 0x0903 and 0x904 are interpreted as a fixed data.

Select **Expression** to define a formula to be evaluated before being displayed. They can be selected at the moment when the control panel is defined. The **Expression** can be built using following operators:

- “+” - addition
- “-” - subtraction
- “*” - multiply
- “/” - division
- “^” - power
- “(“ and “)” - parentheses

The operands used for editing of an **expression** are:

variable_name [unit]

variable_name

number

Example: $100.5 + 5 * (Position_Command [rot] - 10.0)$

In the **Axis ID** edit field of each object, you can select the Axis ID of the axis from where the variable will be read/set. This will allow you to visualize in one control panel variables from different drives / motors connected into a network. The default Axis ID is “**as set in Comm Setup**” e.g. the Axis ID selected in the **Communicate with** field from **Communication | Setup** dialogue.

Select the measurement unit in “**Unit**” section, corresponding to the selected variable type.

Adjust the number of decimals by choosing in “**Precision**” section one of the values from the scroll list, for a convenient representation depending on values range. Note that if the selected measurement unit is “**IU**” (Internal Units), the decimals parameter is not used. Also for **IU** representations, hexadecimal format can be selected.

The title of the object window (displayed in the object window title bar) can be specified in **Title** section. By default, if you don't specify a title, the variable name will be assumed instead.

For display purposes, display time intervals can be selected in the “**Read value at every**” field.

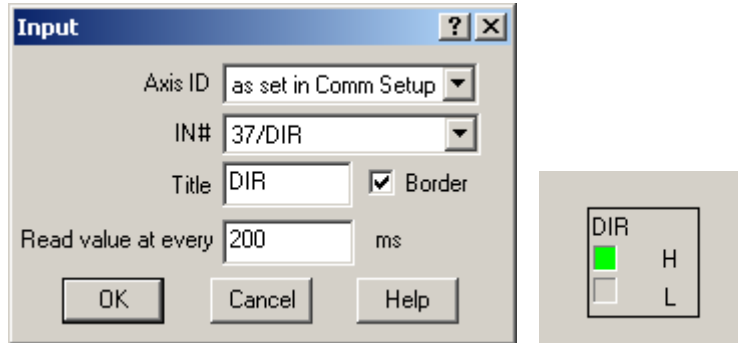
See also:

[The Control Panel Utility](#)

4.2.8. Control Panel - Input



It's used to display the status of an input port.



In the **Axis ID** edit field of each object, you can select the Axis ID of the axis from where the variable will be read/set. This will allow you to visualize in one control panel variables from different drives / motors connected into a network. The default Axis ID is “**as set in Comm Setup**” e.g. the Axis ID selected in the **Communicate with** field from **Communication | Setup** dialogue.

The input number (**IN#**) can be selected from the current list of input ports.

The title of the object window (displayed in the object window title bar) can be specified in **Title** section. By default, if you don't specify a title, the variable name will be assumed instead.

For display purposes, display time intervals can be selected in the “**Read value at every**” field.

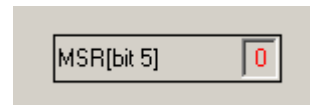
See also:

[The Control Panel Utility](#)

4.2.9. Control Panel - Bit Value



It's used to display the status of a bit of one MPL variable or data memory contents.



The **variable** can be selected from the current list of variables.

In the **Axis ID** edit field of each object, you can select the Axis ID of the axis from where the variable will be read/set. This will allow you to visualize in one control panel variables from different drives / motors connected into a network. The default Axis ID is “**as set in Comm Setup**” e.g. the Axis ID selected in the **Communicate with** field from **Communication | Setup** dialogue.

The bit number will be selected from **Bit Position** .

The title of the object window (displayed in the object window title bar) can be specified in **Title** section. By default, if you don't specify a title, the variable name will be assumed instead.

For display purposes, display time intervals can be selected in the “**Read value at every**” field.

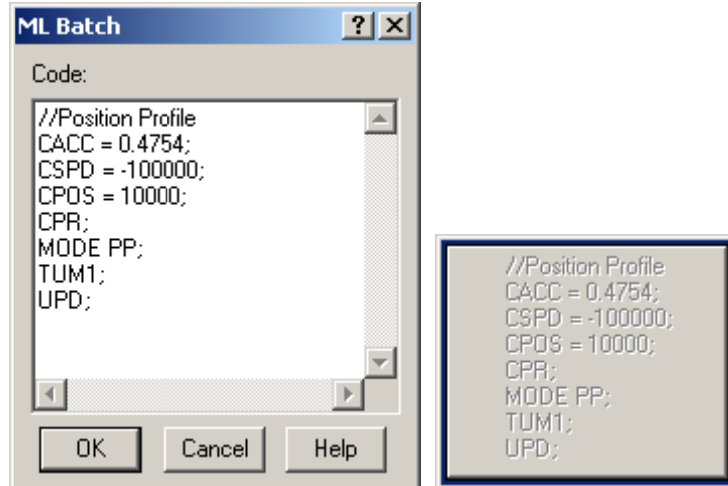
See also:

[The Control Panel Utility](#)

4.2.10. Control Panel - User Defined MPL Sequence Object



It's a button having associated a MPL instructions sequence, user-defined.



You can freely define these instructions. Pressing the button will send the associated MPL commands to the ElectroCraft drive.

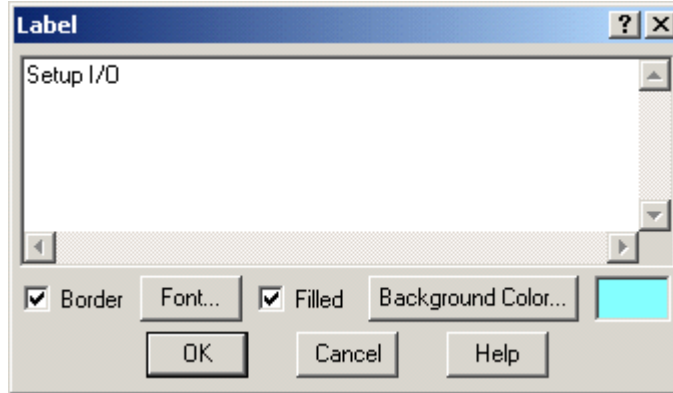
See also:

[The Control Panel Utility](#)

4.2.11. Control Panel - Label



It's an object defining a text or a color-filled rectangle.



Define a text the main edit field.

Choose the font attributes (type, color, size, etc.) by pressing the **Font...** button.

Check **Border** if you want that the text window to be bordered.

You can check **Filled** and choose the background color by pressing the **Background Color...**

Use such objects in order to create more specific control panels, with a better graphical appearance.

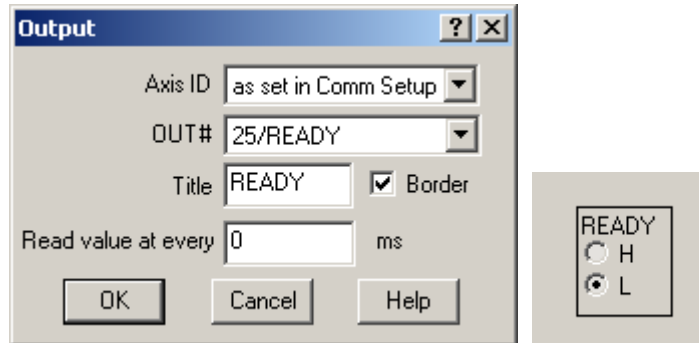
See also:

[The Control Panel Utility](#)

4.2.12. Control Panel - Output



It's used to set the status of an output port.



In the **Axis ID** edit field of each object, you can select the Axis ID of the axis from where the variable will be read/set. This will allow you to visualize in one control panel variables from different drives / motors connected into a network. The default Axis ID is “**as set in Comm Setup**” e.g. the Axis ID selected in the **Communicate with** field from **Communication | Setup** dialogue.

The output number (**OUT#**) can be selected from the current list of output ports.

The title of the object window (displayed in the object window title bar) can be specified in **Title** section. By default, if you don't specify a title, the variable name will be assumed instead.

For display purposes, display time intervals can be selected in the “**Read value at every**” field.

See also:

[The Control Panel Utility](#)

4.2.13. Control Panel Properties

Use this dialog to define / change the name of the current Control Panel.

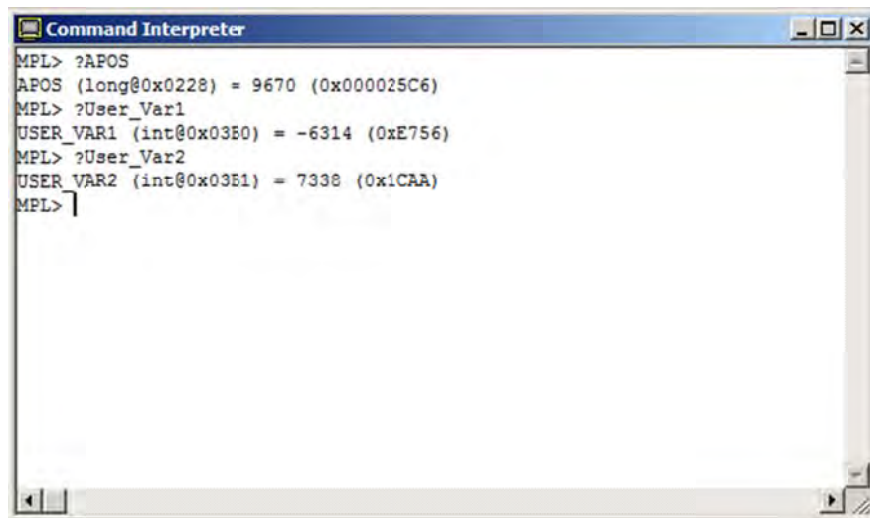
4.3. Command Interpreter

The **Command Interpreter** allows you to send on-line MPL commands to your drive/motor. You can use this tool to set/get MPL data: registers, parameters and variables or to display memory locations.

You can open the Command Interpreter using the "**View | Command Interpreter**" menu command, or by selecting the associated toolbar icon 

In **PROconfig** the MPL commands are sent to the drive/motor for which the setup is performed. This is the drive/motor selected in **Communication | Setup** to communicate with.

In **MotionPRO Developer**, the MPL commands are sent to the drive/motor with the same axis ID as the axis number of the selected application.



To find the value of a MPL data, type in the question mark character "?" followed by the MPL data name and press the [Enter] key. The command interpreter displays the MPL data type and address as *type@address* and its value in decimal and hexadecimal format.

```
MPL> ?apos<Enter>
APOS (long@0x0228) = 1345754 (0x001488DA)
MPL>
```

Remark: Through this method you can find the type and address of any MPL data, including the user-defined variables you create in an MotionPRO Developer application. Note that user-defined variables are accessible only after you compile your application.

To set the value of a MPL data, type its name followed by equal and the value, then press the [Enter] key.

```
MPL> var_i1=0<Enter>
```

With Command Interpreter you can also perform the following operations related with the drive/motor EEPROM or RAM memory:

- **Fill** with a value all the MPL program memory locations between a start address and stop address.

```
MPL>fillmemory 0x4000, 0x4010, 0xABCD<Enter>
```

```
MPL>
```

- **Fill** with a value all the MPL data memory locations between a start address and stop address.

```
MPL>filldatamemory 0x8000, 0x8010, 0x0101<Enter>
```

```
MPL>
```

- **Set** a MPL program memory location with specified value

```
MPL>setmemory 0x4000, 0x0001<Enter>
```

```
MPL>
```

- **Set** a MPL data memory location with specified value

```
MPL>setdatamemory 0x8000, 0x0001<Enter>
```

```
MPL>
```

- **Show** all the MPL program memory locations contents between a start address and stop address

```
MPL>showmemory 0x4000, 0x4010<Enter>
```

```
4000: ABCD ABCD ABCD ABCD ABCD ABCD ABCD ABCD
```

```
4008: ABCD ABCD ABCD ABCD ABCD ABCD ABCD ABCD
```

```
4010: ABCD
```

```
MPL>
```

- **Show** all the MPL data memory locations contents between a start address and stop address

```
MPL>showdatamemory 0x8000, 0x8010<Enter>
```

```
8000: 0101 0101 0101 0101 0101 0101 0101 0101
```

```
8008: 0101 0101 0101 0101 0101 0101 0101 0101
```

```
8010: 0101
```

```
MPL>
```

Remarks:

- *The Command Interpreter memory operations are intended mainly for test and debugging. Do not use them for normal operation. Note that uncontrolled change of memory locations may lead to unexpected results.*
- *For MPL program or data memory addresses ranges see [Memory Map](#).*

The Command Interpreter keeps a history with all commands sent. You can navigate between them with arrow keys UP and DOWN and select one to execute again. When the Command Interpreter window is closed the commands history is reset.

You can access the Command Interpreter menu, by clicking on the right button mouse inside its window. The menu options are:

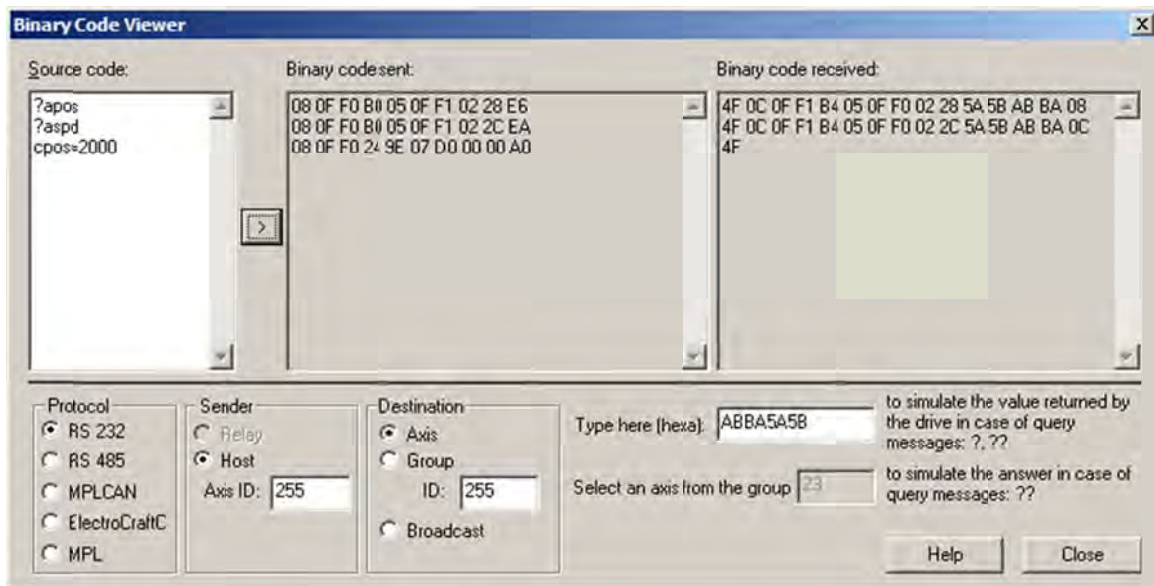
- **Undo/Redo** – reverses the last edit changes done on the current command line / reverses the Undo action
- **Cut/Copy/Paste** – cuts selected text and puts it the on clipboard/copies selected text and puts it in the clipboard/inserts text from clipboard at the insertion point in the command line
- **Toggle Bookmark** – activates/deactivates a bookmark at the insertion point. To navigate between bookmarks use key F2.

See also:

[Memory Map](#)

4.4. Binary Code Viewer

The **Binary Code Viewer** offers you a quick way to program your host for exchanging messages with a ElectroCraft drive/motor. Through this tool, you can find how to encapsulate a MPL command for all the communication types and the supported protocols. You get the both the contents of the messages you have to send and the expected answers from the drive/motor, if it is the case.



First select the communication **Protocol** between: serial **RS 232**, serial **RS 485**, CAN-bus with **MPLCAN**, CAN-bus with **ElectroCAN** or no protocol, just the binary code of the **MPL** commands and answers.

At **Sender** select the **Axis ID** of the message sender. In the case of RS-232, the sender is always your **Host**, as 2 drives/motors may not be connected between them using RS-232. If you select RS-485, MPLCAN or ElectroCAN, the sender can be an **Axis/Host** (another **Axis** or your **Host**) or **None**. Option None, means [non-requested messages](#) sent by the drive/motor, containing a specific MPL data. You can simulate these messages with a ?? query followed by the returned MPL data name and by selecting None. If you select MPL, you can find the binary code for MPL commands sent from both the Host or another axis, which in the case of RS-232 plays the role of an **Relay Axis** (see [Communication Protocols](#) for details)

Remark: Though theoretically possible, activation of non-requested messages is not recommended for the RS-485 where the host must control the communication to avoid conflicts.

At **Destination** choose either an **Axis** of a **Group** of axes. In the first case, set the axis ID of the receiver. In the second case, select a group from 1 to 8 or set group number to 0 for a broadcast message.

In the case of query messages asking the drive/motor to return a MPL data, you can introduce the returned value in hexadecimal format in the **Type here (hexa)** edit box. This helps you to quickly identify the position of the returned data in the message received.

You can simulate 2 types of query or Type B messages (see [Communication Protocols](#) for details):

- A “GiveMeData: request, by typing at **Source Code** a question mark ? followed by a MPL data name (for example **?apos** to read the actual position). In this case the answer is a “TakeData” message
- A “GiveMeData2” request, by typing at **Source Code** a double question mark ?? followed by a MPL data name (for example **??aspd** to read the actual speed). In this case the answer is a “TakeData2” message

On CAN-bus, a “GiveMeData2” request may be sent to a group of drives/motors. For the returned answer you can **Select an axis from the group**.

Remarks:

- *If a “GiveMeData2” request is sent to a group, the “TakeData2” answers are prioritized function of the respondents’ axis ID: the drive/motor with the lowest axis ID has the highest priority.*
- *The “GiveMeData” request is intended only for a single axis. If in a CAN-bus network, “GiveMeData” is sent to a group, all the returned answers have the same CAN identifier and therefore can’t be differentiated, causing an error.*
- *On RS-485, the query messages can’t be sent to group, as the answers will overlap.*

For simulating Type A messages, which do not request to return a data, simply type the MPL instruction at **Source code**. For example to set a position command CPOS of 2000 encoder counts, type **cpos=2000**.

After you have introduced one or more commands, press to arrow button “>” to generate the code.

At **Binary code sent** you’ll see the binary code (in hexadecimal format), which must be sent by your host. When RS-232 and RS-485 are selected, the code displayed represents the bytes you have to send via the serial asynchronous port of your host. When MPLCAN is used, the first 8 hexadecimal numbers represent the 29-bit identifier of the CAN message (the 3MSB of the 32-bit value are zero) and the remaining bytes represent, the CAN message data: byte 0, byte 1, etc. When ElectroCAN protocol is chosen, the first 3 hexadecimal numbers represents the 11-bit identifier of the CAN message (the MSB of the 12-bit value is zero).

At **Binary code received** you’ll see the answer sent by the drive/motor.

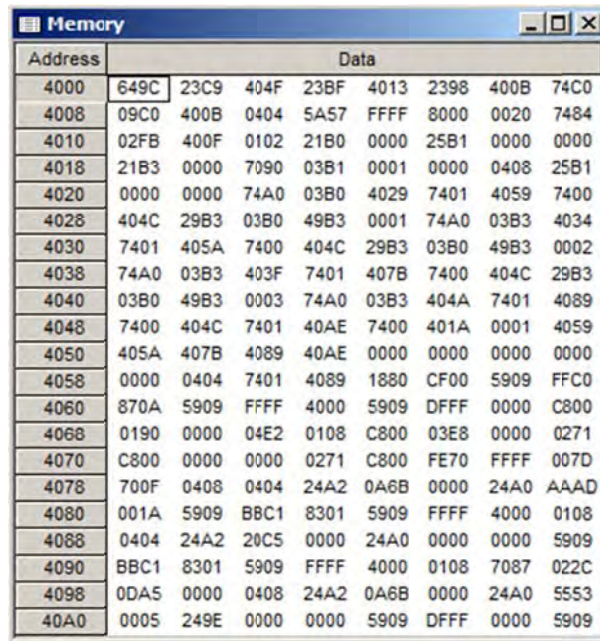
Remark: *On RS-232 and RS-485 each message sent to one axis is confirmed with an acknowledge byte 4Fh. Therefore, in a query message, you’ll see first the 4Fh byte as confirmation for the reception of the data request, followed by the contents of the answer message. On RS-485, the 4Fh acknowledge byte is not sent if the command is sent to a group.*

See also:

[Communication protocols](#)

4.5. Memory View

In **Memory** window you can view/modify the contents of the ElectroCraft drive's/motor's memory from where the MPL program runs.



Address	Data							
4000	649C	23C9	404F	23BF	4013	2398	400B	74C0
4008	09C0	400B	0404	5A57	FFFF	8000	0020	7484
4010	02FB	400F	0102	21B0	0000	25B1	0000	0000
4018	21B3	0000	7090	03B1	0001	0000	0408	25B1
4020	0000	0000	74A0	03B0	4029	7401	4059	7400
4028	404C	29B3	03B0	49B3	0001	74A0	03B3	4034
4030	7401	405A	7400	404C	29B3	03B0	49B3	0002
4038	74A0	03B3	403F	7401	407B	7400	404C	29B3
4040	03B0	49B3	0003	74A0	03B3	404A	7401	4089
4048	7400	404C	7401	40AE	7400	401A	0001	4059
4050	405A	407B	4089	40AE	0000	0000	0000	0000
4058	0000	0404	7401	4089	1880	CF00	5909	FFC0
4060	870A	5909	FFFF	4000	5909	DFFF	0000	C800
4068	0190	0000	04E2	0108	C800	03E8	0000	0271
4070	C800	0000	0000	0271	C800	FE70	FFFF	007D
4078	700F	0408	0404	24A2	0A6B	0000	24A0	AAAD
4080	001A	5909	BBC1	8301	5909	FFFF	4000	0108
4088	0404	24A2	20C5	0000	24A0	0000	0000	5909
4090	BBC1	8301	5909	FFFF	4000	0108	7087	022C
4098	0DA5	0000	0408	24A2	0A6B	0000	24A0	5553
40A0	0005	249E	0000	0000	5909	DFFF	0000	5909

The window is opened selecting the **View | Memory** menu command or the associated toolbar icon. You can refresh the displayed data by selecting the menu command **View | Refresh** button or **F12** key.

Remark: As this feature is a very low level function, it is **NOT** recommended to modify memory contents without a deep knowledge of the use made by the ElectroCraft drive/motor of each memory location you intend to modify

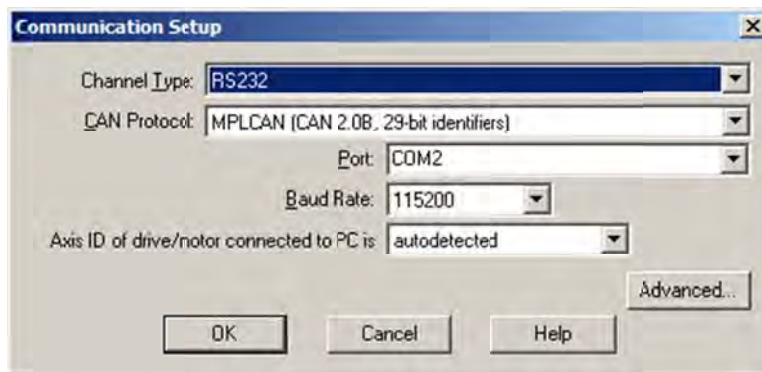
See also:

[Memory Settings](#)

5. Communication

5.1. Communication Setup

The communication settings from this dialogue define how **MotionPRO Developer** is using your PC serial port or a communication interface board. The dialogue allows you to select the communication type between your PC and your ElectroCraft drives/motors. You can choose between: serial **RS-232**, serial **RS-485**, **CAN-bus** or **Ethernet** and setup in each case the communication parameters. With the exception of the RS-232, all the other options require a specific interface. For CAN-bus, the communication settings depend on the interface used. Therefore the **Channel Type** list includes all the CAN-bus interfaces supported.



Remark: If your PC is equipped with another CAN-bus interface, contact ElectroCraft to check for compatibility with one of the interfaces supported

When several drives/motors are connected in a CAN-bus network you have to specify the CAN-bus communication protocol used. This option is also available for serial RS-232 and Ethernet, when the drive/motor connected to the host acts as retransmission relay (see [Communication Protocols](#)). At **CAN Protocol** you can choose either **MPLCAN (CAN 2.0B, 29-bit identifier)** or **CANopen or ElectroCAN (CAN2.0A, 11-bit identifier)**.

Remark: When the **CANopen or ElectroCAN (CAN2.0A, 11-bit identifier)** protocol is selected the **Axis IDs**, of the drives/motors and of the PC, are interpreted as modulo 32.

Through this dialogue you also specify the **Axis IDs** for your PC or in the case of RS-232 or Ethernet the Axis ID of the drive/motor connected with your PC. Each time you close MotionPRO Developer, the communication settings are saved. Next time when you open the MotionPRO Developer, the last settings you have set are restored.

Important Note:

Only a part of the ElectroCraft products supports all communication types. Make sure you select a communication type supported by your product!

Remark: *If you get a communication error message, select “**Communication | Refresh**” command or press the associated button from the toolbar to restore the communication.*

*Note that when using serial RS-232 or RS-485 communication, MotionPRO Developer automatically sets the drives/motors with the baud rate selected in this dialogue. If a drive/motor is reset (power supply is temporary turned off), the serial communication with your PC may no longer work. This happens if the drive/motor default baud rate after reset (9600 baud) differs from that set in MotionPRO Developer. Use “**Communication | Refresh**” command to restore the communication. This starts the automatic baud rate detection, followed by the baud rate change to the value set in MotionPRO Developer.*

See also:

[RS-232 Communication Setup](#)

[RS-232 Communication Troubleshoots](#)

[RS-485 Communication Setup](#)

[RS-485 Communication Troubleshoots](#)

[CAN-bus Communication Setup](#)

[CAN-bus Communication Troubleshoots](#)

[Ethernet Communication Setup](#)

[Ethernet Communication Troubleshoots](#)

[User Implemented Serial Driver Setup](#)

[User Implemented Serial Driver Troubleshoots](#)

[Advanced Communication Setup](#)

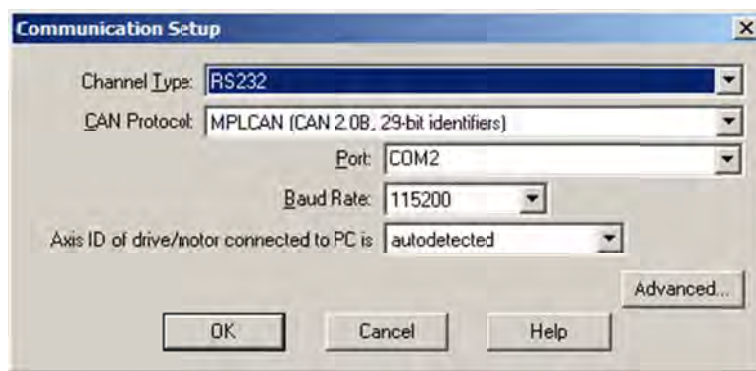
5.1.1. RS-232 Communication Setup

Steps to follow:

1. Setup the drive/motor for RS-232 communication
2. Set MotionPRO Developer for communication via RS-232 with the drive/motor

Step 1 Setup the drive/motor for RS-232 communication

1. Power-Off your drive/motor
2. In order to use the RS-232 communication, you need to connect your PC with the ElectroCraft drive/motor through an RS-232 serial cable. If the drive/motor is equipped with a standard 9-pin DB9 connector for serial communication, use a 9-wire standard serial cable: male-female, non-inverting (e.g. one-to-one), else check the drive/motor user manual for cable connections.
3. If the drive/motor supports also RS-485 communication, set the RS-232/RS-485 switch (or solder-joint) to the position RS-232.
4. Power-On the drive/motor



Step 2 Set MotionPRO Developer for communication via RS-232 with the drive/motor

1. Select menu command “**Communication | Setup**”
2. Select at “**Channel Type**” RS-232 (default).
3. Select the “**CAN Protocol**” between the drives/motors connected in the CAN-bus network, the drive/motor connected to PC acting as a retransmission relay (see [Communication Protocols](#)). You can choose either **MPLCAN (CAN2.0B, 29-bit identifier)** or **CANopen or ElectroCAN (CAN2.0A, 11bit identifier)**.
4. Select at “**Port**” the serial port of your PC, where you have connected the serial cable. By default the selected port is COM1
5. Select the desired baud rate from “**Baud Rate**” list
6. Set the “**Axis ID of the drive/motor connected to PC**”. The default option is **autodetected** enabling MotionPRO Developer to detect automatically the axis ID of the drive connected to the serial port. If your drive/motor doesn't support this feature (see remark below) select its axis ID from the list. The drives/motors axis ID is set at power on using the following algorithm:
 - a. With the value read from the EEPROM setup table containing all the setup data
 - b. If the setup table is invalid, with the last axis ID value read from a valid setup table

-
- c. If there is no axis ID set by a valid setup table, with the value read from the hardware switches/jumpers for axis ID setting
 - d. If the drive/motor has no hardware switches/jumpers for axis ID setting, with the default axis ID value which is 255.

Remark: *When the ElectroCAN communication protocol is used the Axis IDs, of the drives/motors and of the PC, are interpreted as modulo 32.*

7. Press the OK button

If the communication works properly, you'll see displayed on the status bar (the bottom line) of the MotionPRO Developer the text "**Online**", the axis ID of the drive/motor and the firmware version read from the drive/motor.

Remark: *If your drive/motor firmware number:*

- *Starts with 1 – examples: F100A, F125C, F150G, etc., or*
- *Starts with 0 or 9 and has a revision letter below H – examples: F000F, F005D, F900C*

you can't use the axis ID autodetected option.

See also:

[RS-232 Communication Troubleshoots](#)

[Advanced Communication Setup](#)

[Communication Setup](#)

5.1.2. RS-232 Communication Troubleshoots

If the serial RS232 communication does not operate properly, MotionPRO Developer will issue an error message and you'll see displayed on the status bar (the bottom line) of the MotionPRO Developer the text **Offline**.

1. If the error message is **Cannot open the selected serial port**, the serial port you have selected from **Port** does not exist or is used by another device of your PC (mouse, modem, etc.). Click **Cancel**, reopen **Communication | Setup** dialogue, select another serial port and try again.
2. If the error message is **Cannot synchronize the computer and drive/motor baud rates** click **Cancel**, then check the following:
 - Serial cable connections
 - Serial cable type, if you use a standard cable. Make sure that the cable is non-inverting (one-to-one)
 - In **Communication | Setup** dialogue, the **Axis ID of the drive/motor connected to PC is** selection. If you use MotionPRO Developer with a previously bought drive/motor, this may not support the default option **autodetected**. Select the same axis ID with that of your drive/motor. The drives/motors axis ID is set at power on using the following algorithm:
 - a. With the value read from the EEPROM setup table containing all the setup data
 - b. If the setup table is invalid, with the last axis ID value read from a valid setup table
 - c. If there is no axis ID set by a valid setup table, with the value read from the hardware switches/jumpers for axis ID setting
 - d. If the drive/motor has no hardware switches/jumpers for axis ID setting, with the default axis ID value which is 255.
 - Drive hardware settings for RS-232 communication (see [RS-232 Setup](#))
3. If the communication operates usually but gives communication errors from time to time, check the following:
 - If your PC has an earth connection.
 - If your drive/motor is linked to earth. For the drives/motors without an explicit earth point, connect the earth to the ground of the supply/supplies.
 - In **Communication | Setup** dialogue click on the [Advanced...](#) button and increase the **Read interval timeout**, **Timeout multiplier** and **Timeout constant** parameters. Note that these parameters are related to PC serial operation and usually the default values for these parameters do not need to be modified.

After you fix the problem, execute menu command **Communication | Refresh** to restore the communication.

See also:

[RS-232 Communication Setup](#)

[Advanced Communication Setup](#)

[Communication Setup](#)

5.1.3. CAN-bus Communication Setup

Steps to follow:

1. Setup the drives/motors for CAN-bus communication
2. Mount on/connect to your PC a CAN-bus interface board
3. Install on your PC an CAN-bus software driver
4. Build the CAN-bus network
5. Set MotionPRO Developer for communication via CAN-bus with the drives/motors

Step 1. Setup the drives/motors for CAN-bus communication

1. Power-Off the drive/motor
2. Choose a different axis ID for each drive/motor and also different from the axis ID of PC (which is set by default at 255). The drives/motors axis ID is set at power on using the following algorithm:
 - a. With the value read from the EEPROM setup table containing all the setup data
 - b. If the setup table is invalid, with the last axis ID value read from a valid setup table
 - c. If there is no axis ID set by a valid setup table, with the value read from the hardware switches/jumpers for axis ID setting
 - d. If the drive/motor has no hardware switches/jumpers for axis ID setting, with the default axis ID value which is 255.

Remarks:

- *If your drive/motor has no hardware switches/jumpers for axis ID setting, you must program the desired axis ID in the drive/motor setup table. You can do this operation from MotionPRO Developer – the setup part of an application, where you can select the axis ID to be saved in the drive/motor setup table from the EEPROM. Use the RS-232 communication to download the setup data.*
- *When the ElectroCAN communication protocol is used the Axis IDs, of the drives/motors and of the PC, are interpreted as modulo 32.*

Step 2. Mount on your PC a CAN-bus interface board

MotionPRO Developer offers the possibility to choose one of the following PC to CAN-bus interfaces:

- **IxxAT** PC to CAN interface
- **Sys Tec** USB to CAN interface
- **ESD** PC to CAN interface
- **LAWICEL** CANUSB interface
- **PEAK System** PCAN-PCI interface
- **PEAK System** PCAN-ISA
- **PEAK System** PC/104
- **PEAK System** PCAN-USB
- **PEAK System** Dongle interfaces

- Dongle using SPP/EPP protocol
- Dongle with SJA chipset using SPP/EPP protocol
- Dongle Pro with SJA chipset using SPP/EPP protocol

Step 3. Install on your PC a CAN-BUS software driver

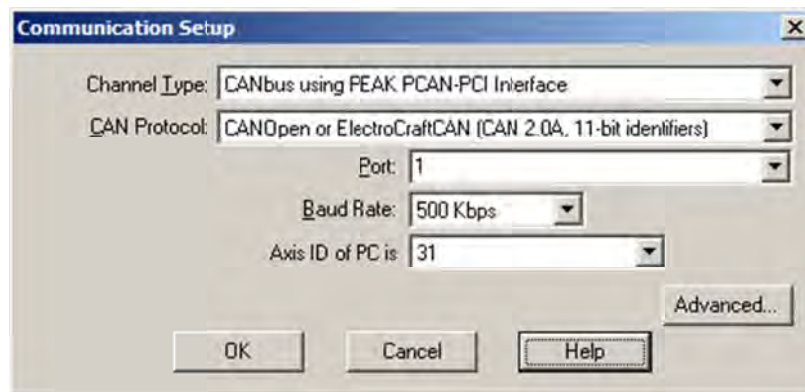
In order to use a CAN-bus interface you need to install on your PC the CAN-bus driver for the chosen interface. For each CAN-bus interface, the producer provides the driver as well as the installation guidelines. You can find detailed information regarding the above interfaces and their installation on the following web pages: www.ixxat.com, www.systemec-electronic.de, www.peak-system.com, www.esd-electronics.com and www.canusb.com (Lawicel interface).

Remarks: For the CAN-bus interfaces from PEAK System you must copy the DLL interface provided in the folder where MotionPRO Developer was installed.

Step 4. Build the CAN-bus network

Each drive/motor manual shows how to do the connections in order to build a CAN-bus network.

Step 5. Set MotionPRO Developer for communication via CAN-bus with the drives/motors



1. Select menu command “**Communication | Setup**”
2. Select at “**Channel Type**” the CAN-bus option corresponding to your interface. For IXXAT CAN-bus interface, PEAK PCAN-ISA, PCAN-PC/104 and PEAK PCAN-Dongle interfaces press “**Select Device...**” button to choose the hardware model corresponding to your device.
3. Select the “**CAN Protocol**” used by the PC to communicate with the drives/motors connected in the CAN-bus network. You can choose either **MPLCAN (CAN2.0B, 29-bit identifier)** or **CANopen or ElectroCAN (CAN2.0A, 11bit identifier)**.
4. Depending on the CAN-bus interface used, you have more or less ports available. Select from “**Port**” the device where you have connected the CAN-bus

Remark: For **Sys Tec USB to CAN interface** the port number must be the same with the device number set with the device configuration utility.

5. Select the CAN-bus interface baud rate from “**Baud Rate**” drop list

Remark: The baud rate selection refers **ONLY** to the CAN-bus interface on the PC. It doesn't change the CAN baud rate on the drives/motors. The default baud rate on CAN-bus for the ElectroCraft drives/motors is 500kbps.

6. Select at “**Axis ID of PC is**” an address for the PC. By default the value proposed is 255. **Attention!** Make sure that all the drives/motors from the network have a different address. When the **CANopen**

*or **ElectroCAN** communication protocol is used the Axis IDs of the drives/motors and of the PC are interpreted as modulo 32.*

7. Press the OK button

If the CAN interface mounted on the PC works properly, you'll see displayed on the status bar (the bottom line) of the **MotionPRO Developer** the text "**Online**" and the axis ID of the PC.

See also:

[CAN-bus Communication Troubleshoots](#)

[Advanced Communication Setup](#)

[Communication Setup](#)

5.1.4. CAN-bus Communication Troubleshoots

If the CAN-bus communication does not operate properly, MotionPRO Developer will issue an error message and you'll see displayed on the status bar (the bottom line) of the MotionPRO Developer the text **Offline**.

1. If the error message is **Cannot find board with selected Axis ID**, click **Cancel** button, then check the following:
 - CAN **Baud rate** selected in the **Communication | Setup** dialogue for the CAN-bus interface. It should be the same with the drives/motors baud rate, which is set by default at power on at 500kbps.
 - CAN-bus cable connections and the presence of the 120 ohms terminal resistors at the two ends of the network
 - If the CAN-bus supply is on
 - In MotionPRO Developer project, the **Axis Number** of the selected application. This should match with the Axis ID of one of the drives from the network. As a general rule, the axis number of each application must correspond with the axis ID of one drive from the network. Each drive must have a different axis ID. No drive can have the same axis ID value as that set as **Axis ID of PC**.
 - The setup of the CAN-bus interface on your PC
 - Drive/motor hardware settings for CAN-bus communication (see [CAN-bus Setup](#))
2. If the error message is **Cannot load interface with PEAK SYS xxxx devices (PCAN_XXXX.DLL)**, click **Cancel** button, and then copy the file "PCAN_XXXX.DLL" from the Peak System CD (or other storage media) in the folder where MotionPRO Developer was installed.
3. If the error message is **Invalid Parameter**, click **Cancel** and check the CAN-bus interface selected in the **Communication | Setup** dialogue. This message occurs when the selected interface is not installed and/or configured on your PC.
4. If the communication operates usually but gives communication errors from time to time, in **Communication | Setup** click on [Advanced...](#) button and increase the **Send message timeout** (when present) and **Receive message timeout** parameters. Note that for these parameters, usually, the default values do not need to be modified.

After you fix the problem, execute menu command **Communication | Refresh** to restore the communication.

See also:

[CAN-bus Communication Setup](#)

[Advanced Communication Setup](#)

[Communication Setup](#)

5.1.5. User implemented serial driver example

```
// Defines the entry point for the DLL application.

// Make sure that shdata is a shared section (e.g. for Microsoft linker you
should use: /SECTION:shdata,RWS)
// This way the s_nInstances variable will be globally visible to all
applications using this DLL

#pragma data_seg("shdata")
static int s_nInstances = 0;

#pragma data_seg()

HANDLE g_hSerialPort = INVALID_HANDLE_VALUE;
DWORD g_nBaudRate = 0;

BOOL APIENTRY DllMain( HANDLE /*hModule*/,
                      DWORD ul_reason_for_call,
                      LPVOID /*lpReserved*/
                      )
{
    switch (ul_reason_for_call)

        case DLL_PROCESS_ATTACH:
            if (s_nInstances == 0)

                g_hSerialPort = CreateFile( "COM1", GENERIC_READ |
GENERIC_WRITE,
                                           0, // exclusive access
                                           NULL, // no security attrs
                                           OPEN_EXISTING,
                                           FILE_ATTRIBUTE_NORMAL,
                                           NULL );
            if (g_hSerialPort == INVALID_HANDLE_VALUE)
                return false;

```

```

        //Initialize serial parameters
        DCB dcb;
        if (!GetCommState(g_hSerialPort, &dcb))
            return false;
        dcb.BaudRate = g_nBaudRate = CBR_9600;
        dcb.ByteSize = 8;
        dcb.Parity = NOPARITY;
        dcb.StopBits = TWOSTOPBITS;
        // Standard flow control
        // setup no hardware flow control
        dcb.fOutxDsrFlow = 0;
        dcb.fDtrControl = DTR_CONTROL_DISABLE;
        dcb.fOutxCtsFlow = 0;
        dcb.fRtsControl = RTS_CONTROL_DISABLE;
        dcb.fDsrSensitivity = false;

        // setup no software flow control
        dcb.fInX = dcb.fOutX = 0;
        dcb.fBinary = true ;
        if (!SetCommState(g_hSerialPort, &dcb))
            return false;

        //Set serial timeouts. ReadData and WriteData must
return
        //in a determined period of time
        COMMTIMEOUTS CommTimeOuts;
        CommTimeOuts.ReadIntervalTimeout = 1000;
        CommTimeOuts.WriteTotalTimeoutMultiplier =
CommTimeOuts.ReadTotalTimeoutMultiplier = 700 ;
        CommTimeOuts.WriteTotalTimeoutConstant =
CommTimeOuts.ReadTotalTimeoutConstant = 500 ;
        if(!SetCommTimeouts(g_hSerialPort, &CommTimeOuts))
            return false;
    }
    else

        //This library does not support connection sharing
between applications

```

```

        //If you need it, you must duplicate file handler from
one process to another
        return false;
    }
    s_nInstances++;
    break;
case DLL_THREAD_ATTACH:
case DLL_THREAD_DETACH:
    break;
case DLL_PROCESS_DETACH:
    s_nInstances--;
    if (s_nInstances == 0)

        CloseHandle(g_hSerialPort);
        g_hSerialPort = INVALID_HANDLE_VALUE;
    }
    break;
}
return true;
}

//must have a timeout
bool __stdcall ReadData(BYTE* pData, DWORD dwBufSize, DWORD* pdwBytesRead)

    return ReadFile(g_hSerialPort, pData, dwBufSize, pdwBytesRead, NULL) ?
true : false;
}

//must have a timeout
bool __stdcall WriteData(const BYTE* pData, DWORD dwBufSize, DWORD*
pdwBytesWritten)

    return WriteFile(g_hSerialPort, pData, dwBufSize, pdwBytesWritten, NULL)
? true : false;
}

int __stdcall GetBytesCountInQueue() // should be non-blocking, < 0 means
error

    COMSTAT comStat;

```

```

    DWORD dwComErrors;
    if (!ClearCommError(g_hSerialPort, &dwComErrors, &comStat))
        return -1;
    return comStat.cbInQue;
}

void __stdcall PurgeQueues()

    PurgeComm(g_hSerialPort, PURGE_TXABORT | PURGE_RXABORT | PURGE_TXCLEAR |
PURGE_RXCLEAR );
}

DWORD __stdcall GetCommBaudRate()

    return g_nBaudRate;
}

bool __stdcall SetCommBaudRate(DWORD nNewBaudRate)

    if(nNewBaudRate != g_nBaudRate)

        DCB dcb;
        if (!GetCommState(g_hSerialPort, &dcb))
            return false;
        dcb.BaudRate = g_nBaudRate = nNewBaudRate;
        if (!SetCommState(g_hSerialPort, &dcb))
            return false;
    }
    return true;
}

```

5.1.6. User Implemented Serial Driver Setup

Steps to follow:

1. Implement the serial driver accordingly with the MPLcomm.dll interface
2. Setup the drive/motor for RS-232 communication
3. Set MotionPRO Developer for communication via user implemented serial driver with the drive/motor

Step 1 Implement the serial driver

In the main function of the dll initialize the communication channel with the serial settings implemented on the ElectroCraft drives/motors: 8 data bits, 2 stop bits, no parity, no flow control and one of the following baud rates: 9600 (default after reset), 19200, 38400, 56600 and 115200.

Implement the functions for interfacing your communication driver with MPLcomm. This functions are:

```
bool __stdcall ReadData(BYTE* pData, DWORD dwBufSize, DWORD* pdwBytesRead)

bool __stdcall WriteData(const BYTE* pData, DWORD dwBufSize, DWORD*
pdwBytesWritten)

int __stdcall GetBytesCountInQueue()

void __stdcall PurgeQueues()

DWORD __stdcall GetCommBaudRate()

bool __stdcall SetCommBaudRate(DWORD nNewBaudRate)
```

where:

pData	Pointer to buffer from/to the data is read/wrote
dwBufsize	Parameter specifying the number of bytes to be read/write from/to serial port
pdwBytesRead	Pointer to the variable that contains the number of bytes read
pdwBytesWritten	Pointer to the variable that contains the number of bytes written
nNewBaudRate	Variable that contains the new value for serial baud rate

Export the functions from the communication driver using a module-definition (.DEF) file with the following content:

```
LIBRARY "virtRS232"

DESCRIPTION 'Example of a virtual serial driver for MPLcomm.dll'

EXPORTS

; Explicit exports can go here

ReadData

WriteData
```

GetBytesCountInQueue

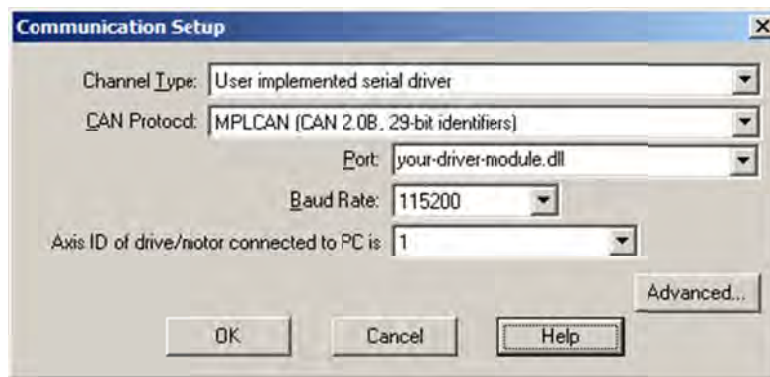
PurgeQueues

GetCommBaudRate

SetCommBaudRate

Step 2 Setup the drive/motor for RS-232 communication

1. Power-Off your drive/motor
2. In order to use the RS-232 communication, you need to connect your PC with the ElectroCraft drive/motor through an RS-232 serial cable. If the drive/motor is equipped with a standard 9-pin DB9 connector for serial communication, use a 9-wire standard serial cable: male-female, non-inverting (e.g. one-to-one), else check the drive/motor user manual for cable connections.
3. If the drive/motor supports also RS-485 communication, set the RS-232/RS-485 switch (or solder-joint) to the position RS-232.
4. Power-On the drive/motor



Step 3 Set MotionPRO Developer for communication via user implemented serial driver with the drive/motor

1. Select menu command “**Communication | Setup**”
2. Select User implemented serial driver at “**Channel Type**”.
3. Select the “**CAN Protocol**” between the drives/motors connected in the CAN-bus network, the drive/motor connected to PC acting as a retransmission relay (see [Communication Protocols](#)). You can choose either **MPLCAN (CAN2.0B, 29-bit identifier)** or **CANopen or ElectroCAN (CAN2.0A, 11bit identifier)**.
4. Specify at “**Port**” the communication dll you implemented
5. Select the desired baud rate from “**Baud Rate**” list
6. Set the “**Axis ID of the drive/motor connected to PC**”. The default option is **autodetected** enabling MotionPRO Developer to detect automatically the axis ID of the drive connected to the serial port. If your drive/motor doesn't support this feature (see remark below) select its axis ID from the list. The drives/motors axis ID is set at power on using the following algorithm:
 - a. With the value read from the EEPROM setup table containing all the setup data
 - b. If the setup table is invalid, with the last axis ID value read from a valid setup table
 - c. If there is no axis ID set by a valid setup table, with the value read from the hardware switches/jumpers for axis ID setting

-
- d. If the drive/motor has no hardware switches/jumpers for axis ID setting, with the default axis ID value which is 255.

Remark: When the ElectroCAN communication protocol is used the Axis IDs, of the drives/motors and of the PC, are interpreted as modulo 32.

7. Press the OK button

If the communication works properly, you'll see displayed on the status bar (the bottom line) of the MotionPRO Developer the text "Online", the axis ID of the drive/motor and the firmware version read from the drive/motor.

Remark: If your drive/motor firmware number:

- Starts with 1 – examples: F100A, F125C, F150G, etc., or
- Starts with 0 or 9 and has a revision letter below H – examples: F000F, F005D, F900C

you can't use the axis ID autodetected option.

See also:

[User Implemented Serial Driver Example](#)

[User Implemented Serial Driver Troubleshoots](#)

[Advanced Communication Setup](#)

[Communication Setup](#)

5.1.7. User Implemented Serial Driver Troubleshoots

If the serial communication does not operate properly, MotionPRO Developer will issue an error message and you'll see displayed on the status bar (the bottom line) of the MotionPRO Developer the text **“Offline”**.

1. If the error message is **“The specified module could not be found”**, the serial driver you have specified at **“Port”** does not exist or its path is not properly set. Click **“Cancel”**, reopen **Communication | Setup** dialogue, check your environment variables and try again.
2. If the error message is **“Cannot synchronize the computer and drive/motor baud rates”** click **“Cancel”**, then check the following:
 - Serial cable connections
 - Serial cable type, if you use a standard cable. Make sure that the cable is non-inverting (one-to-one)
 - In **“Communication | Setup”** dialogue, the **“Axis ID of the drive/motor connected to PC is”** selection. If you use MotionPRO Developer with a previously bought drive/motor, this may not support the default option **“autodetected”**. Select the same axis ID with that of your drive/motor. The drives/motors axis ID is set at power on using the following algorithm:
 - a. With the value read from the EEPROM setup table containing all the setup data
 - b. If the setup table is invalid, with the last axis ID value read from a valid setup table
 - c. If there is no axis ID set by a valid setup table, with the value read from the hardware switches/jumpers for axis ID setting
 - d. If the drive/motor has no hardware switches/jumpers for axis ID setting, with the default axis ID value which is 255.
 - Drive hardware settings for RS-232 communication (see [User Implemented Serial Driver Setup](#))
3. If the communication operates usually but gives communication errors from time to time, check the following:
 - If your PC has an earth connection.
 - If your drive/motor is linked to earth. For the drives/motors without an explicit earth point, connect the earth to the ground of the supply/supplies.
 - In **“Communication | Setup”** dialogue click on the [Advanced...](#) button and increase the **“Read interval timeout”**, **“Timeout multiplier”** and **“Timeout constant”** parameters. Note that these parameters are related to PC serial operation and usually the default values for these parameters do not need to be modified.

After you fix the problem, execute menu command **“Communication | Refresh”** to restore the communication.

See also:

[User Implemented Serial Driver Setup](#)

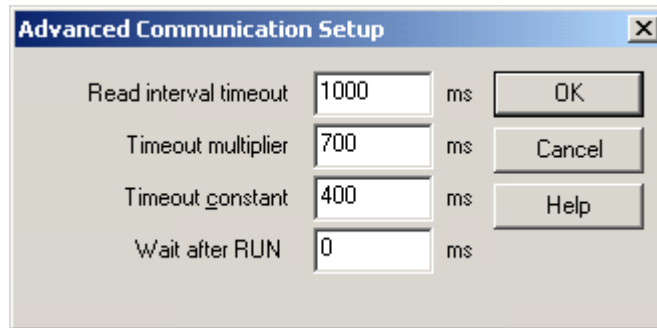
[Advanced Communication Setup](#)

[Communication Setup](#)

5.1.8. Advanced Communication Setup

The advanced communication parameters are related to the host/PC operation. Usually, the default values for these parameters do not need to be modified. You may try to increase these parameters only if the communication works but gives errors from time to time and you have already eliminated all the other possible sources of errors.

When **RS-232** or **RS-485** communication is used, the dialogue displayed is



and the parameters have the following significance:

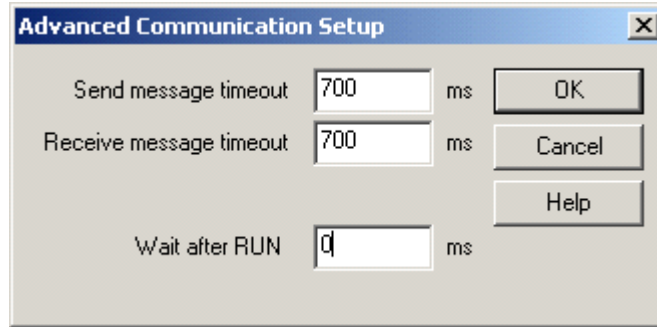
- **Read Interval Timeout** – specifies the maximum time, in milliseconds, allowed to elapse between the arrival of two characters on the communications line. During a read operation, the time period begins when the first character is received. If the interval between the arrivals of any two characters exceeds this amount, the read operation is completed and any buffered data is returned. A value of zero indicates that interval time-outs are not used.
- **Timeout Multiplier** – specifies the multiplier, in milliseconds, used to calculate the total time-out period for read operations. For each read operation, the requested number of bytes to be read multiplies this value.
- **Timeout Constant** – specifies the constant, in milliseconds, used to calculate the total time-out period for read operations. For each read operation, this value is added to the product of the Timeout Multiplier member and the requested number of bytes.

***Remark:** A value of zero for both the Timeout Multiplier and the Timeout Constant members indicates that total time-outs are not used for read operations.*

- **Wait after RUN** – specifies the time interval, in milliseconds, during which the MotionPRO Developer will not communicate with a drive/motor, after it sends it a **Run** command from MotionPRO Developer.

The default values are: **Read interval timeout** – 1000 ms, **Timeout multiplier** – 700 ms, **Timeout constant** – 400 ms, **Wait after RUN** – 0 ms.

When **CAN-bus** communication is used, the dialogue displayed is

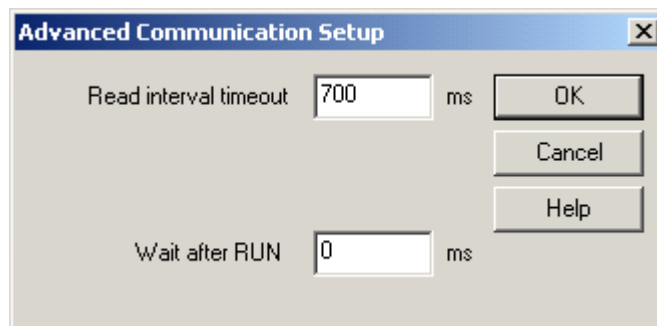


and the parameters have the same significance:

- **Send message timeout** – specifies the maximum time interval, in milliseconds allowed to send a message. If this time interval elapses without sending the message PROconfig/MotionPRO Developer will issue a communication error message. This parameter is available for IxxAT and ESD PC to CAN interfaces.
- **Receive message timeout** – specifies the maximum time, expressed in milliseconds, allowed for an expected message to be received. If this interval elapses without receiving the message PROconfig/MotionPRO Developer will issue a communication error message.
- **Wait after RUN** – same as for RS-232/RS-485

The default values are: **Send message timeout** – 700 ms, **Receive message timeout** – 700 ms, **Wait after RUN** – 0 ms.

When **Ethernet** communication is used, the dialogue displayed is



and the parameters significance is similar with the serial RS-232 case.

The default values are: **Read interval timeout** – 700 ms, **Wait after RUN** – 0 ms.

Additional communication settings can be added directly in the configuration file **kernel.cfg**, from the folder where PROconfig/MotionPRO Developer is installed. The following options can be added:

- **SYNCHRONIZATION_SLEEP_MULTIPLIER** – this parameter is multiplied with the time interval required for synchronization character to be received via RS-232/RS-485. Possible values for the parameter: between 2 and 2000. The Default value is 2.
- **NO_TRIES** – specifies how many times PROconfig/MotionPRO Developer will try to establish the communication with your drive/motor before issuing error messages. The default value is 3.
- **RS485_DTR** – determines PROconfig/MotionPRO Developer to enable/disable the Data Terminal Ready (DTR) line during communications. The Data Terminal Ready signal is sent by the PC to RS485 communication device to indicate that the PC is ready to accept incoming transmission. Possible values for the parameter: 0 (disabled) or 1 (enabled). The default value is 1 (enabled)

-
- **RS485_RTS** – determines PROconfig/MotionPRO Developer to enable/disable the Request To Send (RTS) line. The Request To Send signals that request permission to transmit data is sent from PC to RS485 communication device. Possible values for the parameter: 0 (disabled) or 1 (enabled). The default value is 1 (enabled).

In order to add this parameters open the configuration file **kernel.cfg** with any text editor and at the end of the file add a new section named **[MPLCOMM]**. Bellow the section definition, add the desired parameters in the form **parameter_name = parameter_value**. Save the file and restart PROconfig/MotionPRO Developer.

5.2. Communication Protocols

This section describes the communication protocols supported by the ElectroCraft Programmable drives / motors. It presents how the MPL instructions are packed into messages, for each type of communication channel.

This information is particularly useful for those applications where an external device like a host implements directly one of the ElectroCraft communication protocols. In this case, the host packs the binary code of each MPL command into a message which is sent, and unpacks each message received to extract from it the data provided.

Remark: An alternate way to exchange data with the ElectroCraft drives/motors is via the **MPL_LIB** libraries. A **MPL_LIB** library is a collection of high-level functions for motion programming which you can integrate in the host/master application. If the host is an industrial PC, the **MPL_LIB** library may be integrated in **C/C++**, **Delphi Pascal**, **Visual Basic** or **LabVIEW** applications. If the host is a programmable logic controller (**PLC**), a version of the **MPL_LIB**, compatible with the **PLCopen** standard for motion programming, may be integrated in the PLC IEC 61131-3 application (see ElectroCraft web page www.ElectroCraft.com for details about the **MPL_LIB** libraries)

Depending on the drive/motor, you can use two types of communication channels:

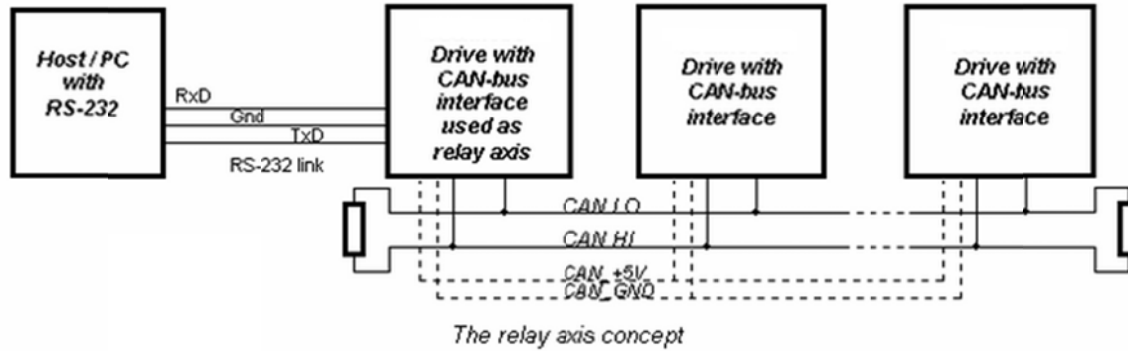
- Serial RS-232 or RS-485
- CAN-bus

The serial RS-232 communication channel can be used to connect a host with one drive/motor. The serial RS-485 and the CAN-bus communication channels can be used to connect up to 32 drives/motors with a host.

Remark: The RS-485 and CAN-bus protocols accept up to 255 nodes. The limitation to 32 nodes is determined by the hardware, using a conservative approach. If your application has more than 32 axes, contact ElectroCraft. Depending on your drive/motor and network characteristics, we can provide you the exact maximum number of axes you may use.

When CAN-bus communication is used, any drive/motor from the network may also be connected through RS-232 or Ethernet with a host. In this case, this drive/motor:

- Executes the commands received from the host via the RS-232 link
- Executes the commands received from other ElectroCraft drives via the CAN-bus link
- Acts like a retransmission relay also called *relay axis*, which:
 - Receives via RS-232, commands from host for another axis and retransmits them to the destination via CAN-bus
 - Receives via CAN-bus data requested by host from another axis and retransmits them to the host via RS-232



The relay axis concept enables a host to communicate with all the ElectroCraft drives/motors from a CAN-bus network, using a single RS-232 or Ethernet connection with one drive/motor. There is no need to have a CAN-bus interface on the host, for which the CAN-bus protocol is completely transparent.

Any drive/motor acts as a *relay axis* when it is connected both on RS-232 and CAN-bus, without any particular setup. The only requirement is to setup the address for the host equal with that of the drive connected via RS-232 (see [Message structure. Axis ID and Group ID](#) for details)

IMPORTANT! MotionPRO Developer includes a [Binary Code Viewer](#), which helps you to quickly find how to send MPL commands using one of the communication channels and protocols supported by the drives/motors. Using this tool, you can get the exact contents of the messages to send as well as of those expected to be received as answers.

See also:

[Message structure. Axis ID and Group ID](#)

[Serial communication. RS-232 and RS-485 protocols](#)

[CAN-bus communication. MPLCAN protocol](#)

[CAN-bus communication. ElectroCAN protocol](#)

5.2.1. Message Structure. Axis ID and Group ID

The data exchange on any communication bus and protocol is done using messages. Each message contains one MPL instruction to be executed by the receiver of the message. Apart from the binary code of the MPL instruction attached, any message includes information about its destination: an axis (drive/motor) or group of axes. This information is grouped in the **Axis/Group ID Code**. Depending on the communication bus and the protocol used, the Axis/Group ID Code and the binary code of the MPL instruction attached are encapsulated in different ways.

Information included in a communication message

Axis/Group ID Code
Operation Code
Data (1)
...
Data (4)

The first word **Axis/Group ID Code** identifies the destination axis or the group of axes that must receive the message. The next words represent the codification of the **MPL instruction** transmitted.

The **Axis/Group ID Code** is a 16-bit word with the following structure:

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

Where:

Bit 0 – HOST bit. 0 – *relay axis*, 1 – host. When a host is connected with a drive using RS-232, the 2 devices must have the same axis ID (bits ID7-ID0 are identical). The HOST bit makes the difference between the host and the drive connected to the other end. On RS-485, the host and the drives have different axis ID, the HOST bit has as no significance and must be set to 0.

Bits 11-4 – ID7-ID0: the 8-bit value of an axis ID or group ID

Bit 12 – GROUP bit: 0 – ID7-ID0 value is an axis ID, 1 – ID7-ID0 value is a group ID

Depending on the communication bus and protocol used, either the entire 16-bit Axis/Group ID code is included in a message or only a part of it. This part can be the 10 bits with useful information: HOST bit, ID7 – ID0 bits and the GROUP bit or a subset of those.

Remark: In the following paragraphs, the terminology **Axis ID Code** or **Group ID Code** designates the above 16-bit word. The terminology **Axis ID** and **Group ID** designates the 8-bit value of an axis or group ID i.e. value of bits ID7 – ID0.

The following example describes how the HOST bit is used: Let's suppose that we have 2 drives with the axis ID=1 and axis ID=2 (values 1 and 2 represent the value of the bits ID7-ID0) connected between them via CAN-bus. The host is connected via RS-232 to the drive with axis ID=1 which acts as a relay axis. The host axis ID (host ID) must also be 1 but with the HOST bit set. The host sends a data request message to the drive with the axis ID=2. The axis ID code of this request message is 2 e.g. the destination axis. The message includes the **sender axis ID** code e.g. where the drive with ID=2 must send the data requested. The sender axis ID code is the host address (ID=1 and the HOST bit set). The request message is sent via RS-232 to drive with axis ID=1. This drive observes that the message destination is another axis (e.g. ID=2) and resends the message via CAN-bus. The drive with the axis ID=2, will receive the request message and send the answer via CAN-bus to the sender axis (e.g. host).

As the host has the same address as the relay axis, all the messages sent via CAN-bus and having as destination the host are received by the relay axis. The relay axis looks at the HOST bit: if the bit is set, then the message received is sent back via RS-232 to the host. If the HOST bit is not set, then the message received is executed (it's destination is the relay axis).

A message can be sent to an axis or to a group of axes. In the first case, the destination is specified via an **Axis ID** code. In the second case, the destination is specified via a **Group ID** code. Each drive/motor has its own 8-bit Axis ID and Group ID stored in the AAR MPL register. If the destination of a message is specified via an Axis ID code, the message is received only by the axis with the same 8-bit Axis ID (bits 11-4 from the 16-bit Axis ID code). If the destination of a message is specified via a Group ID code, each axis compares the 8-bit group ID from the message with its own group ID. If the two group IDs have at least one group (bit set to 1) in common, the message is accepted. In the group ID, each bit corresponds to one group:

Definition of the groups

Group No.	Group ID value
1	1 (0000 0001b)
2	2 (0000 0010b)
3	4 (0000 0100b)
4	8 (0000 1000b)
5	16 (0001 0000b)
6	32 (0010 0000b)
7	64 (0100 0000b)
8	128 (1000 0000b)

A drive/motor can be programmed to be member of up to 8 groups. It will accept all the messages sent to any of the groups his is member. For example, if the drive is member of groups 1, 2 and 4, i.e. its 8-bit Group ID = 11 (00001011b), it will receive all the messages which have in their group ID value at least one of the bits 0,1 or 3 set to 1.

Remarks:

- A message with axis ID = 0 and will be accepted independently of the receiver axis ID
- A broadcast message has the group ID = 0 and will be accepted by all the axes from the network, independently of their group ID

On each drive/motor, the axis ID is initially set at power on using the following algorithm:

- With the value read from the EEPROM setup table containing all the setup data.
- If the setup table is invalid, with the last axis ID value read from a valid setup table
- If there is no axis ID set by a valid setup table, with the value read from the hardware switches/jumpers for axis ID setting
- If the drive/motor has no hardware switches/jumpers for axis ID setting, with the default axis ID value which is 255.

Remark: If the axis ID read from a valid setup table is 0 (option H/W), the axis ID is set with the value read from the hardware switches/jumpers or in their absence with the default value 255

On each drive/motor, at power on, the group ID is set to 1. i.e. all drives/motors are members of the group 1. For each drive/motor you can:

- Set/change its group ID using the MPL instruction [GROUPID](#)
- Add new groups to its group ID using the MPL instruction [ADDGRID](#)
- Remove groups from its group ID using the MPL instruction [REMGRID](#).

Remark: You can read at any moment the actual values of the **axis ID** and **group ID** of a drive/motor from the Axis Address Register [AAR](#)

The **MPL instruction** code can have 1 to 5 words. All the MPL instructions have at least one word – the **Operation Code**. Depending on the type of MPL instruction, the operation code may be followed by 0-4 **Data** words.

Remark: Use [Binary Code Viewer](#) to get the binary code of MPL instructions

See also:

[Communication protocols – Overview](#)

[Serial communication. RS-232 and RS-485 protocols](#)

[CAN-bus communication. MPLCAN protocol](#)

[CAN-bus communication. ElectroCAN protocol](#)

5.2.2. Serial communication. RS-232 and RS-485 protocols

All the ElectroCraft drives/motors can communicate via RS-232. Some of them also accept RS-485 as a substitute for RS-232. In the following paragraphs, the terminology **serial communication** refers to the features common to both RS-232 and RS-485. The terminology **RS-232 communication** or **RS-485 communication** is used to features that are specific for one or the other.

The RS-232 communication is point-to point, full duplex, and enables you to link 2 devices. A typical example is when you connect your PC with a ElectroCraft drive/motor.

Use the RS-232 communication if you want to:

- a) Setup and/or program the motion on one drive/motor using a ElectroCraft development platform like **PROconfig** or **MotionPRO Developer**, running on your PC
- b) Control a drive/motor, with commands sent via communication from your host
- c) Setup and/or program the motion on several drives/motors connected via CAN-bus, where one is also connected via RS-232 with your PC
- d) Control several drives/motors connected via CAN-bus, with commands sent from your host which is connected via RS-232 with one of them

In cases c) and d), the ElectroCraft drive/motor connected to the host acts as a *relay axis* (see [Communication protocols overview](#) for details).

The RS-485 communication is multi-point, half duplex, and enables you to link up to 32 drives/motors in a network. In an RS-485 network, at one moment only one device is allowed to send data. If two devices start by mistake to transmit in the same time, both transmissions are corrupted. Therefore for a correct operation, in an RS-485 network it is mandatory to have a master, which controls the transmission. Put in other words, only the master can initiate a transmission, while all the other devices from the network may transmit only when the master asks them to provide some data. Normally you should set as master your host.

Use the RS-485 communication if you want to:

- a) Setup and/or program the motion on several drives/motors connected via RS-485 together with your PC (requires an RS-485 interface or an RS-232/RS-485 adapter on your PC)
- b) Control several drives/motors connected via RS-485, with commands sent from your host. The host is seen as one node of the RS-485 network, and in must act as a master.

Remark: *If the absence of a host, you can use any drive as master to control the RS-485 communication. This is possible due to the powerful set of MPL commands for multiple axes (see [Motion – Data Transfer Between Axes](#))*

Serial communication settings and message encapsulation

The ElectroCraft drives communicate serially using 8 data bits, 2 stop bits, no parity at the following baud rates: 9600 (default after reset), 19200, 38400, 56600 and 115200. The messages exchanged through serial communication are encapsulated in the following format:

Serial message structure – MPL Instruction encapsulation

Byte 1: Message length
Byte 2: Axis/Group ID Code – high byte
Byte 3: Axis/Group ID Code – low byte
Byte 4: Operation Code – high byte
Byte 5: Operation Code – low byte
Byte 6: Data (1) – high byte
Byte 7: Data (1) – low byte
Byte 8: Data (2) – high byte
...
Byte13: Data (4) – low byte
Last byte: Checksum

The message length byte contains the total number of bytes of the message minus 2. Put in other words, the length byte value is the number of bytes of the: **Axis/group ID Code** (2bytes), the **Operation Code** (2 bytes) and the **Data** words (variable from 0 to 8 bytes). The **Checksum** byte is the sum modulo 256 of all the bytes of the message except the checksum byte itself.

Message types on serial communication

The serial communication protocol is based on 3 types of messages imposed by the nature of the MPL commands encapsulated:

- Type A: Messages that don't require an answer (a return message). These messages can be sent either by a host or by another drive/motor and contain MPL instructions performing parameter settings, motion programming, motor commands, etc.
- Type B: Messages that require an answer. These messages are sent by a host and contain one of the [on line MPL commands](#). These commands ask to return data, for example the value of a MPL parameter, register, or variable.
- Type C: Messages sent by a drive/motor to a host without being requested by the host. These messages may be sent either when a specific condition occurs or following the execution of the MPL command [SEND](#) (see [Messages sent to the host](#) for details)

The next paragraphs present an example of each message type.

Example 1 – Type A Message: A host is connected to a drive/motor via RS-232 and sends the MPL instruction “**KPP = 5**” (set proportional part of the position controller with value 5). The axis ID of host and of the drive/motor are 255 = 0FFh. The Axis ID code and the MPL instruction binary code are:

*Axis ID code + Binary code of MPL Instruction **KPP = 5** sent to axis 255*

Axis ID Code (IDC) = 0FF0h (Destination axis)
Operation Code = 205Eh
Data word (1) = 0005h (Data to set in KPP)

Remark: Use [Binary Code Viewer](#) to get the binary code of MPL instructions

The host must send a serial message with the following contents:

*Serial message: MPL Instruction **KPP = 5** sent to axis 255*

Byte 1: 06h – length: IDC=2,Operation Code=2,Data=2
Byte 2: 0Fh – high byte of Axis ID code = 0FF0h (Destination axis)
Byte 3: F0h – low byte of Axis ID code = 0FF0h (Destination axis)
Byte 4: 20h – high byte of Operation Code = 205Eh
Byte 5: 5Eh – low byte of Operation Code = 205Eh
Byte 6: 00h – high byte of Data (1) = 0005h (Data to set in KPP)
Byte 7: 05h – low byte of Data (1) = 0005h (Data to set in KPP)
Byte 8: 88h – checksum

The drive/motor will return a byte 0x4F as confirmation that the message was received OK. (See below the RS-232 and RS-485 protocols description for details)

Remarks:

- a) *If another drive with axis ID=1 is connected via CAN-bus with the drive having axis ID=255 and the host wants to sent the same MPL instruction “**KPP = 5**” to axis 1, the **Axis ID Code** becomes 0010h instead of 0FF0h.*
- b) *If the host is connected via RS-485 with a drive, the 2 devices must have different axis ID values. For example if the host axis ID = 255 and the drive ID = 1, the message is the same as in remark a)*

Example 2 – Type B Message: A host is connected to a drive via RS-232 and wants to get the value of the **KPP** (proportional term of the position controller) parameter from the drive. **KPP** address in MPL data memory is 025Eh. The ID of the host and the drive/motor are 255 = 0FFh. The host sends a **“GiveMeData”** request and the drive/motor answers with a **“TakeData”** message. Let’s suppose that the **KPP** value returned by the drive/motor is 288 (120h).

Remark: Use [Command Interpreter](#) to get MPL data addresses.

A **“GiveMeData”** request message for a MPL data includes the following information:

“GiveMeData” request for a MPL data – Message description

Axis ID Code (IDC): Destination axis
Operation Code: B004h for 16-bit MPL data B005h for 32-bit MPL data
Data (1): Sender Axis ID Code
Data (2): Requested Data Address

The **“TakeData”** answer message includes the following information:

“TakeData” answer - Message description

Axis ID Code (IDC): Destination axis
Operation Code: B404h for 16-bit data B405h for 32-bit data
Data (1): Sender Axis ID Code
Data (2): Requested Data Address
Data (3): Requested Data Value 16LSB
Data (4): Requested Data Value 16MSB (for 32-bit data)

In the particular case of this example, the axis ID code and the binary code of **“GiveMeData”** are:

*Axis ID code + Binary code of “GiveMeData” request for **KPP** value sent to axis 255*

Axis ID Code (IDC) = 0FF0h (Destination axis)
Operation Code = B004h (for 16-bit MPL data)
Data (1) = 0FF1h (Sender Axis ID Code = IDC + HOST bit set)
Data (2) = 025Eh (KPP address)

The axis ID code and the binary code of **“TakeData”** are:

*Axis ID Code + Binary code of “TakeData” with **KPP** value from axis 255*

Axis ID Code (IDC) = 0FF1h (Destination axis)
Operation Code = B404h (for 16-bit data)
Data(1) = 0FF0h (Sender Axis ID Code)
Data(2) = 025Eh (KPP address)
Data(3) = 0120h (KPP value)

The host must send a “GiveMeData” request message with the following contents:

Serial message: “GiveMeData” request for **KPP** value sent to axis 255

Byte 1: 08h – length: IDC=2,Operation Code=2,Data=4
Byte 2: 0Fh – high byte of Axis ID Code = 0FF0h (Destination axis)
Byte 3: F0h – low byte of Axis ID Code = 0FF0h (Destination axis)
Byte 4: B0h – high byte of Operation Code = B004h
Byte 5: 04h – low byte of Operation Code = B004h
Byte 6: 0Fh – high byte of Data (1) = 0FF1h (Sender Axis ID Code)
Byte 7: F1h – low byte of Data (1) = 0FF1h (Sender Axis ID Code)
Byte 8: 02h – high byte of Data (2) = 025Eh (KPP address)
Byte 9: 5Eh – low byte of Data (2) = 025Eh (KPP address)
Byte 8: 1Bh – checksum

The drive/motor will return a byte 0x4F as confirmation that the message was received OK (See below the RS-232 and RS-485 protocols description for details), then the “TakeData” answer message with the following contents:

Serial message: “TakeData” with **KPP** value from axis 255

Byte 1: 0Ah – length: IDC=2,Operation Code=2,Data=6
Byte 2: 0Fh – high byte of Axis ID code = 0FF1h (Destination axis)
Byte 3: F1h – low byte of Axis ID code = 0FF1h (Destination axis)
Byte 4: B4h – high byte of Operation Code = B404h
Byte 5: 04h – low byte of Operation Code = B404h
Byte 6: 0Fh – high byte of Data (1) = 0FF0h (Sender Axis ID Code)
Byte 7: F0h – low byte of Data (1) = 0FF0h (Sender Axis ID Code)
Byte 8: 02h – high byte of Data (2) = 025Eh (KPP address)
Byte 9: 5Eh – low byte of Data (2) = 025Eh (KPP address)
Byte 10: 01h – high byte of Data (3) = 0120h (KPP value)
Byte 11: 20h – low byte of Data (3) = 0120h (KPP value)
Byte 12: 42h – checksum

Remarks:

- a) If another drive with axis ID=1 is connected via CAN-bus with the drive having axis ID=255 and the host wants to get **KPP** value from axis 1, the **Axis ID Code** becomes 0010h instead of 0FF0h in the “GiveMeData” message. The “Take Data” message also will have 0010h in instead of 0FF0h as **Sender Axis ID Code**.
- b) If the host is connected via RS-485 with a drive, the 2 devices must have different axis ID values. For example if the host has axis ID = 255 and the drive has axis ID = 1, the modifications compared with the above examples are:
 - “GiveMeData”: **Axis ID Code** – 0010h instead of 0FF0h and **Sender Axis ID Code** – 0FF0 instead of 0FF1h (Host bit = 0);
 - “TakeData”: **Axis ID Code** – 0FF0h instead of 0FF1h (Host bit = 0) and **Sender Axis ID Code** – 0010h instead of 0FF0h;

Example 3 – Type C Message: A host is connected to a drive via RS-232 and wants to be informed when the programmed motion is completed. The axis ID of the host and the drive/motor are 255 = 0FFh. A Type C message is a “TakeData2” message sent without a “GiveMeData2” request. It includes the following information:

“TakeData2” - Message description

Axis ID Code (IDC) = MASTERID (Destination axis)
Operation Code: D400h for 16-bit data + 8-bit Axis ID of sender D500h for 32-bit data + 8-bit Axis ID of sender
Data (1): Sent Data Address
Data (2): Sent Data Value 16LSB
Data (3): Sent Data Value 16MSB (for 32-bit data)

The destination axis is provided by the MPL variable **MASTERID**, according with formula: **MASTERID = host axis ID * 16 + 1**. In this example, the 8-bit host axis ID = 255, hence MASTERID = 16 * 255 + 1 = 4081 (0xFF1). In the case of a Type C message, the “TakeData2” can return:

- The 32-bit value of the 2 status registers **SRL** (bits 15-0) and **SRH** (bits 31-16), if one of their selected bits changes (the requested data address is the **SRL** address)
- The 16-bit value of the error register **MER**, if one of its selected bits changes
- The 16-bit value of the PVT/PT status **PVTSTS**, if PVT/PT buffer status changes
- The 16-bit or 32-bit MPL data requested to be sent with the MPL command **SEND**.

Remark: Use [Command Interpreter](#) to get the addresses for the above MPL data. Note that the **SRL** and **SRH** status registers may also be accessed as a single 32-bit variable named **SR32**.

The bit selection is done via 3 masks, one for each register, set in MPL parameters: **SRL_MASK**, **SRH_MASK**, **MER_MASK**. A bit set in a mask, enables a message transmission when the same bit from the corresponding register changes. In this example, the motion complete condition is signaled by setting **SRL.10 = 1**. To activate automatic sending of a “TakeData2” whenever **SRL.10** changes, set **SRL_MASK = 0x0400**.

If **SRH = 0x201** and **SRL = 0x8400**, after **SRL.10** goes from 0 to 1, the host gets a “TakeData2” message with the following contents:

Serial message: **“TakeData2”** with status registers **SRL** and **SRH** from axis 255

Byte 1: 0Ah – length: IDC=2, Operation Code=2, Data=6
Byte 2: 0Fh – high byte of MASTERID = 0FF1h (Destination axis)
Byte 3: F1h – low byte of MASTERID = 0FF1h (Destination axis)
Byte 4: D5h – high byte of Operation Code = D4FFh
Byte 5: FFh – low byte of Operation Code = D4FFh
Byte 6: 09h – high byte of Data (1) = 090Eh (SRL address)
Byte 7: 0Eh – low byte of Data (1) = 090Eh (SRL address)
Byte 8: 84h – high byte of Data (2) = 8400h (SRL value)
Byte 9: 00h – low byte of Data (2) = 8400h (SRL value)
Byte 10: 02h – high byte of Data (3) = 0201h (SRH value)
Byte 11: 01h – low byte of Data (3) = 0201h (SRH value)
Byte 12: 7Ch - checksum

Remark: A **“TakeData2”** message with **SRL.10=1** signals that the last programmed motion is completed. A **“TakeData2”** message with **SRL.10=0** signals that a new motion has started and may be used as a confirmation for the last motion command.

RS-232 communication protocol

The RS-232 protocol is full duplex, allowing simultaneous transmission in both directions. After each command (Type A or B) sent by the host, the drive will confirm the reception by sending one acknowledge-**Ok** byte. This byte is: ‘O’ (ASCII code of capital letter “o”, 0x4F). If the host receives the ‘O’ byte, this means that the drive has received correctly (checksum verification was passed) the last message sent, and now is ready to receive the next message.

Remark: If the destination axis for the message is not the axis connected with the host via RS-232 (e.g. the relay axis), but another axis connected with the relay axis via CAN-bus, the reception of the acknowledge-**Ok** byte from the relay axis doesn’t mean that the message was received by the destination axis, but just by the relay axis. Depending on the CAN-bus baud rate and the amount of traffic on this bus, the host may need to consider introducing a delay before sending the next message to an axis connected on the CAN-bus. This delay must provide the relay axis the time necessary to retransmit the message via CAN-bus.

If any error occurs during the message reception, for example the checksum computed by the drive axis doesn’t match with the one sent by the host, the drive will not send the acknowledge-**Ok** byte. If the host doesn’t receive any acknowledge byte for at least 2ms after the end of the checksum byte transmission, this means that at some point during the last message transmission, one byte was lost and the synchronization between the host and the relay axis is gone. In order to restore the synchronization the host should do the following:

- 1) Send a SYNC byte having value 0x0d (higher values are also accepted)
- 2) Wait a programmed timeout (typically 2ms) period for an answer;
- 3) If the drive sends back a SYNC byte having value 0x0d, the synchronization is restored and the host can send again the last message, else go to step 1
- 4) Repeat steps 1 to 3 until the drive answers with a SYNC byte or until 15 SYNC bytes are sent. If after 15 SYNC bytes the drive/motor still doesn’t answer, then there is a serious communication problem and the serial link must be checked

When a host sends a type A message through RS-232 it has to:

- a) Send the message (as in Example 1);
- b) Wait the acknowledge-OK byte 'O' from the drive;

When a host sends a type B message through RS-232 it has to:

- a) Send the request message (as in Example 2 in case of a "Give Me Data" command)
- b) Wait the acknowledge-OK byte 'O' from the drive connected via RS-232 (relay axis);
- c) Wait the answer message from the drive/motor (as in Example 2, in case of a "Take Data" answer)

When the relay axis returns an answer message it doesn't expect to receive an acknowledge byte from the host. It is the host task to monitor the communication. If the host gets the response message with a wrong checksum, it is the host duty to send again the data request.

RS-485 communication protocol

The RS-485 protocol is half duplex. If two devices start by mistake to transmit in the same time, both transmissions are corrupted. Therefore for a correct operation, in an RS-485 network it is mandatory to have a master, which controls the transmission. This means that only the master can initiate a transmission, while all the other devices from the network may transmit only when the master asks them to provide some data. Usually you should set as master your host.

After each command (Type A or B) sent by the host to one drive, the drive will confirm the reception by sending one acknowledge-Ok byte. This byte is: 'O' (ASCII code of capital letter "o", 0x4F). If the host receives the 'O' byte, this means that the drive has received correctly (checksum verification was passed) the last message sent, and now is ready to receive the next message.

The acknowledge-Ok byte is not sent when the host broadcasts a message to a group of drives.

If any error occurs during the message reception, for example if the checksum computed by the drive axis doesn't match with the one sent by the host, the drive will not send the acknowledge-Ok byte. If the host doesn't receive any acknowledge byte for at least 2ms after the end of the checksum byte transmission, this means that at some point during the last message transmission, one byte was lost and the synchronization between the host and the relay axis is gone. In order to restore the synchronization the host should do the following:

- 1) Send 15 SYNC bytes having value 0x0d or any other bigger value up to 0xFF
- 2) Wait a programmed timeout (typically 2ms);
- 3) Send again the last command and wait for the drive answer
- 4) If the drive still doesn't answer, then there is a serious communication problem and the serial link must be checked

When a host sends a type A message through RS-485 it has to:

- a) Send the message (as in Example 1);
- b) Wait the acknowledge-OK byte 'O' from the drive, only if the message destination was a single drive;

When a host sends a type B message through RS-485 it has to:

- a) Send the request message (as in Example 2 in case of a "Give Me Data" command)

-
- b) Wait the acknowledge-OK byte 'O' from the drive;
 - c) Wait the answer message from the drive/motor (as in Example 2, in case of a "Take Data" answer)

Remarks:

- *When using the RS-485 protocol, do not send Type B request messages to a group of axes, because the answer messages will overlap*
- *When using the RS-485 protocol, the Type C messages must be suppressed. Only the host/master is allowed to initiate a transmission*

When a drive returns an answer message it doesn't expect to receive an acknowledge byte from the host. It is the host task to monitor the communication. If the host gets the response message with a wrong checksum, it is the host duty to send again the data request.

See also:

[Communication protocols – Overview](#)

[CAN-bus communication. MPLCAN protocol](#)

[CAN-bus communication. ElectroCAN protocol](#)

[Message structure. Axis ID and Group ID](#)

5.2.3. CAN-bus communication. ElectroCAN protocol

ElectroCAN is an alternate protocol to MPLCAN – the default CAN-bus protocol for the ElectroCraft drives/motors without CANopen. ElectroCAN was specifically designed to permit connection of the ElectroCraft drives/motors without CANopen on a CANopen network where messages are exchanged using CANopen protocol. ElectroCAN and CANopen do not disturb each other and therefore can co-exist on the same physical bus.

On request, the ElectroCraft drives/motors without CANopen may be delivered with ElectroCAN protocol. The difference between the drives/motors with MPLCAN protocol and those with ElectroCAN protocol is done only through the firmware: all the ElectroCraft products equipped with ElectroCAN have a firmware number starting with 2 i.e. a firmware code is F2xxY, where 2xx is the firmware number and Y is the firmware revision.

ElectroCAN is based on CAN2.0A using 11 bits for the identifier. It accepts the following baud rates: 125kb, 250kb, 500kb (default after reset), 800kb and 1Mb. Like MPLCAN, ElectroCAN offers the possibility to connect a PC via a serial RS-232 link to any drive/motor from the CANopen network and through it to access all the ElectroCraft drives/motors. In this case, this drive/motor connected both to CAN-bus and RS-232 becomes a *relay axis* (see [Communication protocols – Overview](#) for details)

In ElectroCAN the MPL instructions are split into 8 categories:

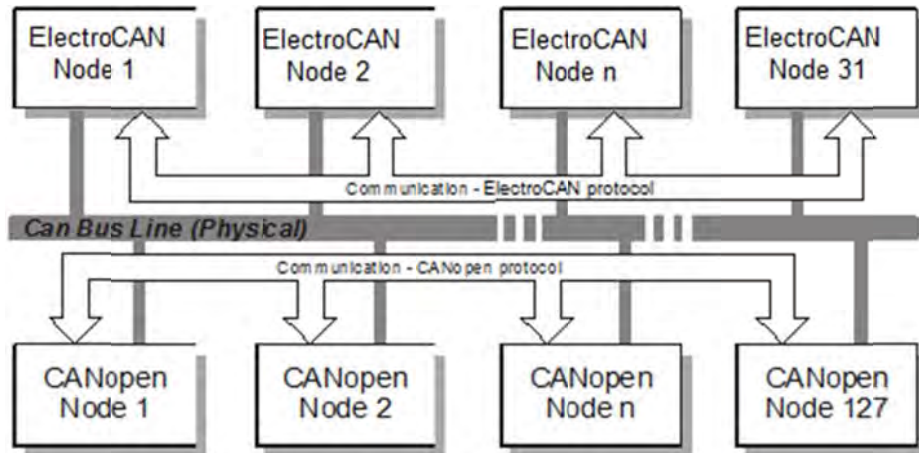
- a) **Normal** – includes all the MPL instructions addressed to a single drive/motor (axis)
- b) **TakeData** – includes the answer “TakeData” to the request “GiveMeData”
- c) **Group** – includes all the MPL instructions multicast to a group of drives/motors
- d) **Host** – includes the answers to all the other [on line MPL commands](#) except “TakeData”
- e) **PVT** – includes the instruction PVTP (the instruction is to long to be sent as a normal message)
- f) **Synchronization** – includes the synchronization message for the group 0
- g) **Broadcast** – includes all the MPL instructions addressed to the group 0 (to the all drives in the system) except the request GiveMeData.
- h) **TakeData2** – includes the answers “TakeData2” to the request “GiveMeData2”

Each category is mapped in the following range of COB-ID (Communication Object Identifier – CANopen terminology for a CAN message identifier):

COB-IDs mapping for ElectroCAN messages

Description	Range COB-ID	
	Dec	Hex
Group message	1-31	1-1F
Synchronization message	32	20
PVT message	65-95	41-5F
TakeData2 message	257-287	101-11F
Normal message	289-319	121-13F
Host message	321-351	141-15F
TakeData message	353-383	161-17F
Broadcast message	512	200

ElectroCAN uses only COB-IDs outside of the range used by CANopen. Thus, ElectroCAN protocol and CANopen protocol can co-exist and communicate simultaneously on the same physical CAN bus, without disturbing each other.



The next table shows how ElectroCAN COB-IDs are assigned in relation with the CANopen COB-IDs.

CANOpen and ElectroCAN COB-IDs

COB-ID value	COB-ID in CANopen	COB-ID in ElectroCAN
0h	USED – NMT	NOT USED
1 ÷ 1Fh	NOT USED	USED – Group messages
20	NOT USED	USED – Synchronization messages
21 ÷ 40h	NOT USED	NOT USED
40-5Fh	NOT USED	USED – PVT messages
5F ÷ 7Fh	NOT USED	NOT USED
80h	USED – SYNC	NOT USED
81 ÷ FFh	USED – EMERGENCY	NOT USED
100h	USED – TIME STAMP	NOT USED
101 ÷ 11Fh	NOT USED	USED – Take Data2 messages
120h	NOT USED	NOT USED
121 ÷ 13Fh	NOT USED	USED – Normal messages
140h	NOT USED	NOT USED
141 ÷ 15Fh	NOT USED	USED – Host messages
160h	NOT USED	NOT USED
161 ÷ 17Fh	NOT USED	USED – Take Data messages
180h	NOT USED	NOT USED
181 ÷ 19Fh	USED – PDO1 (tx)	NOT USED
1A0 ÷ 1FFh	USED – PDO1 (tx)	NOT USED
200h	NOT USED	USED – Broadcast messages
201 ÷ 21Fh	USED – PDO1 (rx)	NOT USED
220 ÷ 27Fh	USED – PDO1 (rx)	NOT USED
280h	NOT USED	NOT USED
281 ÷ 29Fh	USED – PDO2 (tx)	NOT USED
2A0 ÷ 2FFh	USED – PDO2 (tx)	NOT USED
300h	NOT USED	NOT USED
301 ÷ 31Fh	USED – PDO2 (rx)	NOT USED
320 ÷ 37Fh	USED – PDO2 (rx)	NOT USED
380h	NOT USED	NOT USED
381 ÷ 3FFh	USED – PDO3 (tx)	NOT USED
400h	NOT USED	NOT USED
401 ÷ 47Fh	USED – PDO3 (rx)	NOT USED
480h	NOT USED	NOT USED
481 ÷ 4FFh	USED – PDO4 (tx)	NOT USED
500h	NOT USED	NOT USED
501 ÷ 57Fh	USED – PDO4 (rx)	NOT USED
580h	NOT USED	NOT USED
581 ÷ 5FFh	USED – SDO (tx)	NOT USED
600h	NOT USED	NOT USED
601 ÷ 67Fh	USED – SDO (rx)	NOT USED
680 ÷ 6FFh	NOT USED	NOT USED
700h	NOT USED	NOT USED
701 ÷ 77Fh	USED – NMT Error Control	NOT USED

Remarks: *In comparison with MPLCAN, TechoCAN has the following restrictions:*

- *The maximum number of axes is 31: possible Axis ID values: 1 to 31*
- *The maximum number of groups is 5: possible Group ID values: 1 to 5*

Normal messages encapsulation: COB-ID: 121h – 13Fh

	10	9	8	7	6	5	4	3	2	1	0	
COB-ID	0	0	1	0	0	1	Axis ID					
Byte 0							Op. Code [7...0]					
Byte 1							Op. Code [15...8]					
Byte 2							Data (1) [7...0]					
Byte 3							Data (1) [15...8]					
Byte 4							Data (2) [7...0]					
Byte 5							Data (2) [15...8]					
Byte 6							Data (3) [7...0]					
Byte 7							Data (3) [15...8]					

Host messages encapsulation: COB-ID: 141h – 15Fh

	10	9	8	7	6	5	4	3	2	1	0	
COB-ID	0	0	1	0	1	0	Axis ID					
Byte 0							Op. Code [7...0]					
Byte 1							Op. Code [15...8]					
Byte 2							Data (1) [7...0]					
Byte 3							Data (1) [15...8]					
Byte 4							Data (2) [7...0]					
Byte 5							Data (2) [15...8]					
Byte 6							Data (3) [7...0]					
Byte 7							Data (3) [15...8]					

Remark: Host messages occur only when a drive/master answers to a data request (other than “GiveMeData”) where the Sender Axis ID has the HOST bit set to 1. This happens for example when the host is a PC connected to one of the drives/motors via RS-232 and asks a data from another drive/motor. The answer will be sent to the relay axis as a Host message. The Host messages do not occur when the request is sent by a drive or by a host/master connected directly on the CAN bus.

Take Data messages encapsulation: COB-ID: 161h – 17Fh

	10	9	8	7	6	5	4	3	2	1	0	
COB-ID	0	0	1	0	1	1	Axis ID					
Byte 0	Op. Code [7...0]											
Byte 1	Data (1) [8...4]						H	Op. Code [9...8]				
Byte 2	Data (2) [7...0]											
Byte 3	Data (2) [15...8]											
Byte 4	Data (3) [7...0]											
Byte 5	Data (3) [15...8]											
Byte 6	Data (4) [7...0]											
Byte 7	Data (4) [15...8]											

Remarks: In the Take Data messages, the 10-byte code of the Take Data MPL instruction is compacted to 8-bytes. This is done in the following way:

- From the 16-bit Operation Code, only the first 10LSB are transmitted. The 6MSB are always constant: 0x2D (101101b) and are not transmitted. The receiver of a Take Data message must add 0x2D on the 6MSB of the Operation Code received in order to restore the full 16-bit code for TakeData instruction.
- The HOST bit is transmitted in bit 2 of byte 1. There is no need to send the GROUP bit because the GiveMeData request can't be sent to a group of drives/motors.
- The first data word of the TakeData MPL instruction is the Sender Axis ID. As the maximum number of drives is limited to 31, only bits 8-4 are useful and are transmitted.

Group messages encapsulation: COB-ID: 001h – 01Fh

	10	9	8	7	6	5	4	3	2	1	0	
COB-ID	0	0	0	0	0	0	Group ID					
Byte 0							Op. Code [7...0]					
Byte 1							Op. Code [15...8]					
Byte 2							Data (1) [7...0]					
Byte 3							Data (1) [15...8]					
Byte 4							Data (2) [7...0]					
Byte 5							Data (2) [15...8]					
Byte 6							Data (3) [7...0]					
Byte 7							Data (3) [15...8]					

PVT messages encapsulation: COB-ID: 041h – 05Fh

	10	9	8	7	6	5	4	3	2	1	0	
COB-ID	0	0	0	0	1	0	Axis ID					
Byte 0							Position [7...0]					
Byte 1							Position [15...8]					
Byte 2							Speed [15...8]					
Byte 3							Position [23...16]					
Byte 4							Speed [23...16]					
Byte 5							Speed [31...24]					
Byte 6							Time [7...0]					
Byte 7							Counter[6...0]					T[8]

Remarks: In the PVT messages, the 10-byte code of the PVT MPL instruction is compacted to 8-bytes. This is done in the following way:

- The Operation Code is not transmitted. The receiver of a PVT message adds 0x6 on the 9MSB of the Operation Code received and the Counter value on the 7LSB in order to restore the full 16-bit code for PVT instruction.
- The first data word of the PVT instruction contains the 15LSB of the 24 bits Position
- The second data word of the PVT instruction contains the 8LSB of the 24 bits Speed value and the 8 MSB of the 24 bits Position value.
- The third data word of the PVT instruction contains the 16MSB of the 24 bits Speed value.
- The fourth data word of the PVT instruction contains the 9bits Time value.

Synchronization messages encapsulation: COB-ID: 020h

	10	9	8	7	6	5	4	3	2	1	0
COB-ID	0	0	0	0	0	1	0	0	0	0	0

Remarks:

- *The message has zero data bytes*
- *The Operation Code is 0x1000*
- *The synchronization messages are broadcast messages; they are received by every drive connected to the network*

Broadcast messages encapsulation: COB-ID: 200h

	10	9	8	7	6	5	4	3	2	1	0	
COB-ID	0	1	0	0	0	0	0	0	0	0	0	
Byte 0							Op. Code [7...0]					
Byte 1							Op. Code [15...8]					
Byte 2							Data (1) [7...0]					
Byte 3							Data (1) [15...8]					
Byte 4							Data (2) [7...0]					
Byte 5							Data (2) [15...8]					
Byte 6							Data (3) [7...0]					
Byte 7							Data (3) [15...8]					

Take Data 2 messages encapsulation: COB-ID: 101h – 11Fh

	10	9	8	7	6	5	4	3	2	1	0
COB-ID	0	0	1	0	0	0	Expeditor Axis ID				
Byte 0				VT	P	0	0	0	0	0	0
Byte 1				Data (1) [7...0]							
Byte 2				Data (1) [15...8]							
Byte 3				Data (2) [7...0]							
Byte 4				Data (2) [15...8]							
Byte 5				Data (3) [7...0]							
Byte 6				Data (3) [15...8]							

Remarks:

- The message will be never receive by one of the ElectroCraft drive, the message is dedicate for other drives.
- The COB-ID contains the Expeditor Axis ID for the host to get the answers one by one, prioritized in the ascending order of the expeditors' axis ID.
- The VT bit specifies the data length (VT = 0 for 16bits or VT = 1 for 32 bits) and is transmitted in the first byte sent.
- The P bit specifies if the message is TakeData2, in reply to a GiveMeData2 message, or a PONG, in reply to a PING message. The PING message is a broadcast message that requests the Axis ID and the firmware version of the drives in the network. For P=0 the message is Take Data2 and for P = 1 the message is a PONG (the VT bit is automatically reset and it has no meaning).

Example 1: A host connected on a CANopen network sends to drive/motor with axis ID = 5 the MPL instruction “**KPP = 0x1234**” (set proportional part of the position controller with value 0x1234). The Axis ID Code and the MPL instruction binary code are:

*Binary code of MPL instruction **KPP =0x1234***

Operation Code = 205Eh
Data word (1) = 1234h (Data to set in KPP)

Remark: Use [Binary Code Viewer](#) to get the binary code of MPL instructions

The host must send a ElectroCAN message with the following contents:

*ElectroCAN message: MPL instruction **KPP = 0x1234** sent to axis 5*

	10	9	8	7	6	5	4	3	2	1	0
COB-ID (Axis ID = 5)	0	0	1	0	0	1	0	0	1	0	1
Byte 0 (5Eh)							0	1	0	1	1
Byte 1 (20h)							0	0	1	0	0
Byte 2 (34h)							0	0	1	1	0
Byte 3 (12h)							0	0	0	1	0
Byte 4							0	0	0	0	0
Byte 5							0	0	0	0	0
Byte 6							0	0	0	0	0
Byte 7							0	0	0	0	0

Remark: *The last 4 bytes are not used and are not transmitted*

Example 2: A host connected on a CANopen network wants to get the value of the position error from the drive/motor with the axis ID=5. The host axis ID is 3. The position error is the 16-bit MPL variable named **POSERR** and its address in the MPL data memory is 0x022A. The host sends to axis 5 a “GiveMeData” request for the MPL variable **POSERR** and waits for the “TakeData” answer.

The Axis ID Code and the binary code of “GiveMeData” request for POSERR are:

*Binary code of **GiveMeData** request for **POSERR** value sent to axis 5*

Operation Code = B004h
Data word (1) = 0030h
Data word (2) = 022Ah

The host must send a ElectroCAN message with the following contents:

ElectroCAN message: GiveMeData request for POSERR value sent to axis 5

	10	9	8	7	6	5	4	3	2	1	0
COB-ID (Axis ID = 5)	0	0	1	0	0	1	0	0	1	0	1
Byte 0 (04h)							0	0	0	0	0
Byte 1 (B0h)							1	0	1	1	0
Byte 2 (30h)							0	0	1	1	0
Byte 3 (00h)							0	0	0	0	0
Byte 4 (2A)							0	0	1	0	1
Byte 5 (02)							0	0	0	0	1
Byte 6							0	0	0	0	0
Byte 7							0	0	0	0	0

Remark: The last 2 bytes are not used and are not transmitted.

Supposing that the drive/motor with Axis ID = 5 returns a position error **POSERR** = 2, the Axis ID Code and the binary code of the **“TakeData”** answer is:

*Binary code of **TakeData** with **POSERR** value from axis 5*

Operation Code = D404h
Data word (1) = 022Ah
Data word (2) = 0002h

The host gets a ElectroCAN message with the following contents:

*ElectroCAN message: **TakeData** with **POSERR** value from axis 5*

	10	9	8	7	6	5	4	3	2	1	0	
COB-ID (Axis ID = 3)	0	0	1	0	1	1	0	0	0	1	1	
Byte 0 (04h)							0	0	0	0	1	0
Byte 1 (28h)							0	0	1	0	1	0
Byte 2 (2Ah)							0	0	1	0	1	0
Byte 3 (02h)							0	0	0	0	0	1
Byte 4 (02h)							0	0	0	0	0	1
Byte 5 (00h)							0	0	0	0	0	0
Byte 6							0	0	0	0	0	0
Byte 7							0	0	0	0	0	0

Remark: The last 2 bytes are not used and are not transmitted.

Example 3: A PVT command is sent to the drive with the axis ID 5 like following: pvtp -1000L, -10, 500U, 0 (set the coordinates for the next point the position at -1000 IU = 0,5 rot = FFFC18h, the speed at -10IU = 300 rpm = FFF600 and the time 500IU = 0,5s = 01F4).

*Binary code of **PVT** command sent to axis 5*

Operation Code + Counter Value = 1800h + 0h = 1800h
Data (1) = Position Value[15..0] = FC18h
Data (2) = Position Value[23..16] Speed Value [15..8] = FF 00 h
Data (3) = Speed Value [31..16] = FFF6h
Data (4) = Time Value [8..0] = F401h

The ElectroCAN message sent has the following contents:

ElectroCAN message: PVT command for axis 5

	10	9	8	7	6	5	4	3	2	1	0
COB-ID (Axis ID = 5)	0	0	0	0	1	0	0	0	1	0	1
Byte 0 (18h)					0	0	0	1	1	0	0
Byte 1 (FCh)					1	1	1	1	1	1	0
Byte 2 (00h)					0	0	0	0	0	0	0
Byte 3 (FFh)					1	1	1	1	1	1	1
Byte 4 (F6h)					1	1	1	1	0	1	1
Byte 5 (FFh)					1	1	1	1	1	1	1
Byte 6 (F4h)					1	1	1	1	1	1	1
Byte 7 (01h)					0	0	0	0	0	0	1

Example 4: If a ElectroCraft drive/motor receives the MPL instruction SETSYNC 20, it becomes the synchronization master and starts sending every 20ms a synchronization message and its time to the all drives connected in the CAN bus network.

At a moment the master time has the value 0x246C46F and the code of MPL instruction is the following:

*Binary code of **Set Master Time** command sent to all axes*

Operation Code = 1401h
Data (1) = Time value [15...0] = C46Fh
Data (2) = Time value [31...16] = 0246h

The ElectroCAN messages are:

- The synchronization message that when it is received by everybody specifics time variables are saved.

*ElectroCAN message: **Synchronization** command for all axes*

	10	9	8	7	6	5	4	3	2	1	0
COB-ID (Group ID = 0)	0	1	0	0	0	0	0	0	0	0	0
Byte 0 (00h)					0	0	0	0	0	0	0
Byte 1 (00h)					0	0	0	0	0	0	0

Remark: The last 8 bytes are not used and are not transmitted.

- The master broadcast messages with the command to the slaves to set the master time

ElectroCAN message: **Set Master Time** command to all axes

	10	9	8	7	6	5	4	3	2	1	0
COB-ID (Group ID = 0)	0	1	0	0	0	0	0	0	0	0	0
Byte 0 (01h)							0	0	0	0	1
Byte 1 (14h)							0	0	1	0	0
Byte 2 (6Fh)							0	1	1	1	1
Byte 3 (C4h)							1	1	0	0	0
Byte 4 (46h)							0	1	0	0	1
Byte 5 (02h)							0	0	0	0	1

Example 5: If for example the axis 2 encounters a control error, the drive sends a message with the value of the error register MER (0x0008) with a TakeData2 instruction which has the following content:

Binary code of **TakeData 2** with MER register value from axis 2

Operation Code + Expeditor Axis ID = D400h + 2h = D402h
Data (1) = Memory address of data requested [15...0] = 08FCh
Data (2) = Data requested [15...0] = 0008h

Remark: The VT bit is set to zero

The ElectroCAN message sent has the following contents:

ElectroCAN message: **TakeData2** command from axis 2

	10	9	8	7	6	5	4	3	2	1	0
COB-ID (Expeditor Axis ID = 2)	0	1	0	0	0	0	0	0	0	1	0
Byte 0 (00h)							0	0	0	0	0
Byte 1 (FCh)							1	1	1	1	0
Byte 2 (08h)							0	0	0	1	0
Byte 3 (40h)							0	1	0	0	0
Byte 4 (00h)							0	0	0	0	0

Remark: The last 3 bytes are not used and are not transmitted.

See also:

[Communication protocols – Overview](#)

[Message structure. Axis ID and Group ID](#)

[Serial communication. RS-232 and RS-485 protocols](#)

[CAN-bus communication. MPLCAN protocol](#)

5.2.4. CAN-bus communication. MPLCAN protocol

Most of the ElectroCraft drives/motors can communicate via CAN-bus. The CAN-bus communication is multi-point, half duplex, and enables you to link up to 32 drives/motors in a network.

The major advantage of the CAN-bus is its capability to solve automatically the conflicts. On a CAN-bus network, if two devices start to transmit in the same time, one of them (having the higher priority) always wins the network access and completes the transmission. The other device, after losing the network access, commutes from transmission to reception, receives the message with the higher priority, then tries again to transmit its own message. All this procedure is done automatically by the hardware (CAN-bus controller) and it is transparent at higher levels. Put in other words, one can work with a CAN-bus network like being full duplex, knowing that if transmission conflicts occur, these are automatically solved.

ElectroCraft drives/motors have been specifically designed to exploit the CAN-bus benefits. For example, in multi-axis applications you can really distribute the intelligence between the master and the drives/motors. Instead of trying to command each step of an axis movement, you can program the drives/motors using MPL to execute complex tasks and inform the master when these are done. Thus for each axis the master task may be reduced at: calling MPL functions (with possibility to abort their execution if needed) and waiting for a message, which confirms the execution. If needed, the drives/motors may also be programmed to send periodically information messages to the master so it can monitor a task progress.

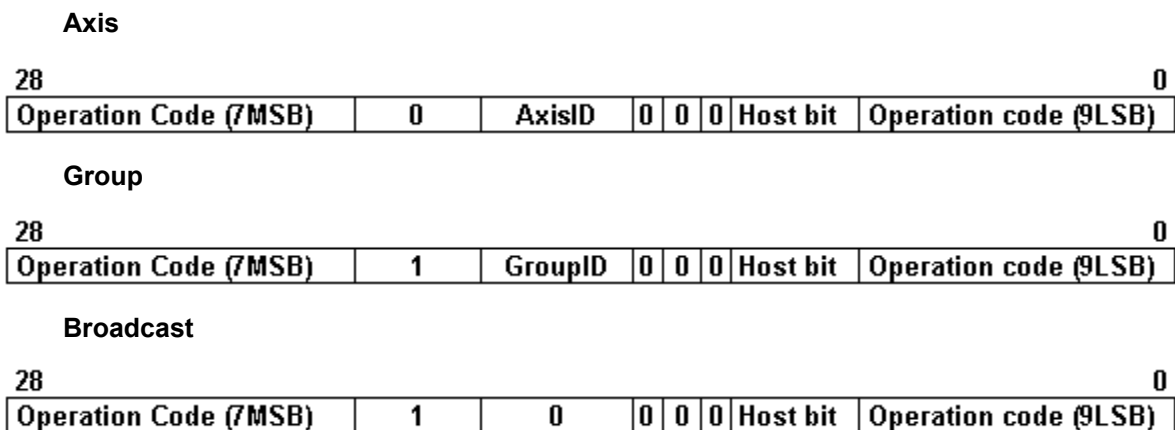
Depending on product, ElectroCraft drives/motors are delivered either with MPLCAN protocol or with CANopen. On request, the MPLCAN protocol, which is based on CAN2.0B, may be replaced with [ElectroCAN](#) protocol which is based on CAN2.0A. ElectroCAN was specifically designed to permit connection of the ElectroCraft drives/motors without CANopen on a CANopen network where messages are exchanged using CANopen protocol. ElectroCAN and CANopen do not disturb each other and therefore can co-exist on the same physical bus.

Message encapsulation in MPLCAN protocol

MPLCAN is based on CAN2.0B using 29 bits for the identifier. It accepts the following baud rates: 125kb, 250kb, 500kb (default after reset), 800kb and 1Mb.

The message destination (an axis or a group of axes) and the MPL instruction binary code are encapsulated as follows:

CAN message identifier of a message sent to:



CAN message data bytes:

CAN Message Data Byte No.	MPL Data Word
0	Data word (1) – low byte
1	Data word (1) – high byte
2	Data word (2) – low byte
3	Data word (2) – high byte
4	Data word (3) – low byte
5	Data word (3) – high byte
6	Data word (4) – low byte
7	Data word (4) – high byte

CAN message structure

Message types on CAN-bus communication

The CAN-bus communication is based on 3 types of messages imposed by the nature of the MPL commands encapsulated:

- Type A: Messages that don't require an answer (a return message). These messages can be sent either by a host or by another drive/motor and contain MPL instructions performing parameter settings, motion programming, motor commands, etc.
- Type B: Messages that require an answer. These messages are sent by a host and contain one of the [on line MPL commands](#). These commands ask to return data, for example the value of a MPL parameter, register, or variable.
- Type C: Messages sent by a drive/motor to a host without being requested by the host. These messages may be sent either when a specific condition occurs or following the execution of the MPL command [SEND](#) (see [Messages sent to the host](#) for details)

The next paragraphs present an example of each message type.

Example 1 – Type A Message: A host connected on CAN-bus sends to drive/motor with axis ID = 5 the MPL instruction “**KPP = 0x1234**” (set proportional part of the position controller with value 0x1234). The MPL instruction binary code are:

*Binary code of MPL instruction **KPP = 0x1234***

Operation Code = 205Eh
Data word (1) = 1234h (Data to set in KPP)

Remark: Use [Binary Code Viewer](#) to get the binary code of MPL instructions

The CAN message identifier is:

*CAN message identifier: MPL instruction **KPP = 0x1234** sent to axis 5*

28	0							
Operation Code (7MSB of 205Eh)	Group bit	AAR [7..0]	0	0	0	Host bit	Operation code (9LSB of 205Eh)	
0010000	0	00000101	0	0	0	0	001011110	0400A05Eh

The host must send a CAN message with the following contents:

*CAN message: MPL instruction **KPP = 0x1234** sent to axis 5*

	Value	Description
Identifier	0400A05E	CAN Message Identifier
Byte 0	34	Low byte of Data word (1) = 1234h
Byte 1	12	high byte of Data word (1) = 1234h

Example 2 – Type B Message: A host wants to get the position error of 2 drives/motors, which are members of group 1. The host axis ID is 3 and the drives/motors axis ID are 5 and 7. The position error is the 16-bit MPL variable named **POSERR** and its address in the MPL data memory is 0x022A. The host sends to group 1 a “**GiveMeData2**” request for the MPL variable **POSERR** and waits for the “**TakeData2**” answers.

The Group ID Code and the binary code of “**GiveMeData2**” request for **POSERR** are:

*Binary code of **GiveMeData2** request for **POSERR** value sent to group 1*

Operation Code = B204h
Data word (1) = 0030h
Data word (2) = 022Ah

The CAN message identifier is:

*CAN message identifier: **GiveMeData2** request for **POSERR** value sent to group 1*

28	0							
Operation Code (7MSB of B204h)	Group bit	AAR [15..8]	0	0	0	Host bit	Operation code (9LSB of B004h)	
1011001	1	00000001	0	0	0	0	000000100	16602004h

The host must send a CAN message with the following contents:

CAN message: **GiveMeData2** request for **POSERR** value sent to group 1

	Value	Description
Identifier	16602004	CAN Message Identifier
Byte 0	30	low byte of Data word (1) = 0031h
Byte 1	00	High byte of Data word (1) = 0031h
Byte 2	2A	low byte of Data word (2) = 022Ah
Byte 3	02	High byte of Data word (2) = 022Ah

Supposing that the drive/motor with Axis ID = 5 returns a position error **POSERR** = 2, the binary code of the “**TakeData2**” answer is:

Binary code of **TakeData2** with **POSERR** value from axis 5

Operation Code = D405h
Data word (1) = 022Ah
Data word (2) = 0002h

The CAN message identifier is:

CAN message identifier: **TakeData2** with **POSERR** value from axis 5

Operation Code (7MSB of D405h)	Group bit	AAR [7..0]	0	0	0	Host bit	Operation code (9LSB of B004h)
1101010	0	00000011	0	0	0	0	000000101

1A806005h

The host gets a CAN message with the following contents:

CAN message: **TakeData2** with **POSERR** value from axis 5

	Value	Description
Identifier	1A806005	CAN Message Identifier
Byte 0	2A	low byte of Data word (1) = 022Ah
Byte 1	02	high byte of Data word (1) = 022Ah
Byte 2	02	low byte of Data word (2) = 0002h
Byte 3	00	high byte of Data word (2) = 0002h

Supposing that the drive/motor with Axis ID = 7 returns a position error **POSERR** = 1, the binary code of the “**TakeData2**” answer is:

Binary code of **TakeData2** with **POSERR** value from axis 7

Operation Code = D407h
Data word (1) = 022Ah
Data word (2) = 0001h

The CAN message identifier is:

CAN message identifier: **TakeData2** with **POSERR** value from axis 7

28							0
Operation Code (7MSB of D407 h)	Group bit	AAR [7..0]	0	0	0	Host bit	Operation code (9LSB of B004 h)
1 1 0 1 0 1 0	0	0 0 0 0 0 0 1 1	0	0	0	0	0 0 0 0 0 0 1 1 1

1A806007h

The host gets a CAN message with the following contents:

CAN message: **TakeData2** with **POSERR** value from axis 7

	Value	Description
Identifier	1A806007	CAN Message Identifier
Byte 0	2A	low byte of Data word (1) = 022Ah
Byte 1	02	high byte of Data word (1) = 022Ah
Byte 2	01	low byte of Data word (2) = 0002h
Byte 3	00	high byte of Data word (2) = 0002h

Example 3 – Type C Message: A host is connected to a drive via CAN-bus and wants to be informed when the programmed motion is completed. The host axis ID = 255 and the drive/motor axis ID = 1. A Type C message is a “**TakeData2**” message sent without a “**GiveMeData2**” request. It includes the following information:

“**TakeData2**” – Message description

Operation Code: D400h for 16-bit data + 8-bit Axis ID of sender D500h for 32-bit data + 8-bit Axis ID of sender
Data (1): Sent Data Address
Data (2): Sent Data Value 16LSB
Data (3): Sent Data Value 16MSB (for 32-bit data)

The destination axis is provided by the MPL variable **MASTERID**, according with formula: **MASTERID = host axis ID * 16 + 1**. In this example, the 8-bit host axis ID = 255, hence **MASTERID = 16 * 255 + 1 = 4081 (0xFF1)**. In the case of a Type C message, the “**TakeData2**” can return:

- The 32-bit value of the 2 status registers **SRL** (bits 15-0) and **SRH** (bits 31-16), if one of their selected bits changes (the requested data address is the SRL address)
- The 16-bit value of the error register **MER**, if one of its selected bits changes
- The 16-bit value of the PVT/PT status **PVTSTS**, if PVT/PT buffer status changes
- The 16-bit or 32-bit MPL data requested to be sent with the MPL command **SEND**.

Remark: Use **Command Interpreter** to get the addresses for the above MPL data. Note that the **SRL** and **SRH** status registers may also be accessed as a single 32-bit variable named **SR32**.

The bit selection is done via 3 masks, one for each register, set in MPL parameters: **SRL_MASK**, **SRH_MASK**, **MER_MASK**. A bit set in a mask, enables a message transmission when the same bit from the corresponding register changes. In this example, the motion complete condition is signaled by setting

SRL.10 = 1. To activate automatic sending of a “TakeData2” whenever **SRL.10** changes, set **SRL_MASK = 0x0400.**

Supposing that the drive/motor with Axis ID = 1 returns **SRH = 0x201** and **SRL = 0x8400**, after **SRL.10** goes from 0 to 1, the Axis ID Code and the binary code of the “TakeData2” message is:

Axis ID Code + Binary code of TakeData2 with status registers SRL and SRH from axis 1

Operation Code = D501h
Data word (1) = 090Eh
Data word (2) = 8400h
Data word (3) = 0201h

The CAN message identifier is:

CAN message identifier: TakeData2 with status registers SRL and SRH from axis 1

28							0
Operation Code (7MSB of D501h)	Group bit	AAR [7..0]	0	0	0	Host bit	Operation code (9LSB of B004h)
1101010	0	11111111	0	0	0	1	100000001
							1A9FE301h

The host gets a CAN message with the following contents:

CAN message: TakeData2 with status registers SRL and SRH from axis 1

	Value	Description
Identifier	1A9FE301	CAN Message Identifier
Byte 0	0E	low byte of Data word (1) = 090Eh
Byte 1	09	high byte of Data word (1) = 090Eh
Byte 2	00	low byte of Data word (2) = 8400h
Byte 3	04	high byte of Data word (2) = 8400h
Byte 4	01	low byte of Data word (2) = 0201h
Byte 5	02	high byte of Data word (2) = 0201h

Remark: A “TakeData2” message with **SRL.10=1** signals that the last programmed motion is completed. A “TakeData2” message with **SRL.10=0** signals that a new motion has started and may be used as a confirmation for the last motion command.

See also:

[Communication protocols – Overview](#)

[CAN-bus communication. ElectroCAN protocol](#)

[Message structure. Axis ID and Group ID](#)

[Serial communication. RS-232 and RS-485 protocols](#)

6. Application Programming

6.1. Motion Programming – drives with built-in Motion Controller

One of the key advantages of the ElectroCraft drives/motors is their capability to execute complex motions without requiring an external motion controller. This is possible because ElectroCraft drives offer in a single compact package both a state of art digital drive and a powerful motion controller.

Programming motion on a ElectroCraft drive/motor means to create and download a MPL (ElectroCraft Motion Program Language) program into the drive/motor memory. The MPL allows you to:

- Set various motion modes (profiles, PVT, PT, electronic gearing or camming, etc.)
- Change the motion modes and/or the motion parameters
- Execute homing sequences
- Control the program flow through:
 - Conditional jumps and calls of MPL functions
 - MPL interrupts generated on pre-defined or programmable conditions (protections triggered, transitions on limit switch or capture inputs, etc.)
 - Waits for programmed events to occur
- Handle digital I/O and analogue input signals
- Execute arithmetic and logic operations
- Perform data transfers between axes
- Control motion of an axis from another one via motion commands sent between axes
- Send commands to a group of axes (multicast). This includes the possibility to start simultaneously motion sequences on all the axes from the group
- Synchronize all the axes from a network

With MPL, you can really distribute the intelligence between the master and the drives/motors in complex multi-axis applications. Thus, instead of trying to command each step of an axis movement, you can program the drives/motors using MPL to execute complex tasks and inform the master when these are done. Thus for each axis the master task may be reduced at: calling MPL functions (with possibility to abort their execution if needed) and waiting for a message, which confirms the execution. If needed, the drives/motors may also be programmed to send periodically information messages to the master so it can monitor a task progress.

A MPL program includes a **main** section, followed by the subroutines used: **functions**, **interrupt** service routines and **homing** procedures. The MPL program may also include **cam tables** used for electronic camming applications.

When you select the "[Motion](#)" part of an application, you access the main section of your application MPL program.

You can select the other components of a MPL program too. Each has 2 types of access views:

- **Definition and/or selection view**, with the following purposes:
 - **Homing modes:** select the homing procedure(s) to use from a list of already defined procedures.
 - **Functions:** create new MPL functions (initially void) and manipulate those defined: delete, rename, change their order in the program
 - **Interrupts:** choose the MPL interrupt service routines you want to view/change their default implementation
 - **Cam Tables:** create new cam tables loaded from other applications or imported from text files and manipulate those defined: select those to be downloaded and their order, delete or rename.
- **Edit view** – for editing the contents. There is one edit view for each homing procedure and cam table selected, for each function defined and each interrupt chosen for view/edit.

In order to help you create a MPL program, MotionPRO Developer includes a **Motion Editor** which is automatically activated when you select “**M Motion**” – the main section view or an edit view for a homing procedure, function or interrupt service routine. The Motion Editor adds a set of toolbar buttons in the project window just below the title bar. Each button opens a programming dialogue. When a programming dialogue is closed, the associated MPL instructions are automatically generated. Note that, the MPL instructions generated are not a simple text included in a file, but a motion object. Therefore with Motion Editor you define your motion program as a collection of motion objects.

The major advantage of encapsulating programming instructions in motion objects is that you can very easily manipulate them. For example, you can:

- Save and reuse a complete motion program or parts of it in other applications
- Add, delete, move, copy, insert, enable or disable one or more motion objects
- Group several motion objects and work with bigger objects that perform more complex functions

The Motion Editor includes the following programming dialogues:

Motion Programming and control

[Trapezoidal Profiles](#)

[S-curve Profiles](#)

[PT](#)

[PVT](#)

[External](#)

[Electronic Gearing](#)

[Electronic Camming](#)

[Motor Commands](#)

[Position Triggers](#)

[Homing](#)

[Contouring](#)

[Test](#)

Events Programming

[Event Types](#)

[When the actual motion is complete](#)

[Function of motor or load position](#)

[Function of motor or load speed](#)

[After a wait time](#)

[Function of reference](#)

[Function of inputs status](#)

[Function of a variable value](#)

[Jumps and Function Calls](#)

[I/O Handling](#)

Assignment & Data Transfer

[16-bit Integer Data](#)

[32-bit Integer Data](#)

[Arithmetic Operations](#)

[Multiple Axis Data Transfer](#)

[Send to Host](#)

[Miscellaneous commands](#)

[Interrupt Settings](#)

[Free Text Editor](#)

See also:

[Motion View](#)

[Homing Procedures View](#)

[Functions View](#)

[Interrupts View](#)

[Cam Tables View](#)

6.1.1. Motion Programming Toolbars

The top toolbar contains the buttons associated to motion programming dialogues.



The “[Motion – Trapezoidal Profiles](#)” allows you to program a positioning path described through a series of points. Each point specifies the desired Position and Time, i.e. contains a PT data. Between the points the built-in reference generator performs a linear interpolation.



The “[Motion – S-curve Profiles](#)” allows you to program a position profile with an S-curve shape of the speed. This shape is due to the jerk limitation, leading to a trapezoidal or triangular profile for the acceleration and an S-curve profile for the speed.



The “[Motion - PT](#)” allows you to program an arbitrary profile whose contour is described by a succession of linear segments.



The “[Motion – PVT](#)” allows you to program a positioning path described through a series of points. Each point specifies the desired Position, Velocity and Time, i.e. contains a PVT data. Between the points the built-in reference generator performs a 3rd order interpolation



The “[Motion – External](#)” allows you to program the drives/motors to work with external reference provided by another device.



The “[Motion – Electronic Gearing](#)” dialogue allows you to set the drives as master or a slave for *electronic gearing* mode.



The “[Motion – Electronic Camming](#)” dialogue allows you to set the drives as master or a slave for *electronic camming* mode.



The “[Motor Commands](#)” allows you to apply one of following commands to the motor: activate/deactivate the control loops and the power stage PWM output commands (AXISON / AXISOFF), stop the motor with acceleration/deceleration set, change the value of the motor position and position reference.



The “[Motion – Position Triggers](#)” dialogue allows you to define 4 position trigger points.



The “[Motion – Homing](#)” dialogue allows you choose a homing procedure and set its parameters.



The “[Motion - Contouring](#)” allows you to program an arbitrary contour via a series of points. Between the points, linear interpolation is performed, leading to a contour described by a succession of linear segments.



The “[Test](#)” dialogue allows you to set the drives/motors in a special test configuration.



The “[Events](#)” allows you to define an event to be monitored and to perform several actions.



The “[Jumps and Function Calls](#)” allows you to control the MPL program flow through unconditional or conditional jumps and unconditional, conditional or cancelable calls of MPL functions.



The “[I/O](#)” allows you program operations with the digital inputs and outputs of the drives/motors.



The “[16-bit Integer Data](#)” helps you to program an assignment operation through which you can set the value of a 16-bit variable or set a memory location with a 16-bit immediate value or the value of a 16-bit variable.



The “[32-bit Long or Fixed Data](#)” helps you to program an assignment operation through which you can set the value of a 32-bit variable, set the low part (16LSB) or the high part (16MSB) of a 32-bit variable with a 16-bit value / variable value, set a memory location with a 32-bit immediate value or the value of a 32-bit variable.



The “[Arithmetic Operations](#)” helps you to program one of the arithmetic operations accepted by the MPL (ElectroCraft Motion Program Language): addition, subtraction, product or shifting.



The “[Data Transfer Between Axis](#)” helps you to program the data transfer operations between drives that are connected in a network.



The “[Send Data to Host](#)” dialogue allows you to choose what information is sent by the drive automatically. You can send information about status register, error register or variables.



The “[Miscellaneous](#)” dialogue allows you to declare new variables, reset FAULT status, insert a END instruction, insert an NOP instruction, set the baud rates for the Serial Communication Interface (SCI) used for RS-232 and RS-485, set the baud rates for the CAN communication.



The “[MPL Interrupt Settings](#)” allows you to activate and/or deactivate the MPL (ElectroCraft Motion Program Language) interrupts



The “[Free text](#)” opens a dialogue where you can freely insert comments or MPL instructions in the current position.

Once the parameters have been entered, a “*motion sequence*” is created. Such a sequence represents a macro-instruction to which one or more specific MPL instructions correspond. The MotionPRO Developer automatically generates the MPL code for these motion sequences.

The right toolbar contains buttons used for the motion sequences management.



Insert. Allows you choose a new motion sequence to be inserted.

- **Motion.**

- **Trapezoidal Profiles.** This command allows you to program a position or speed profile with a trapezoidal shape of the speed, due to a limited acceleration.
- **S-Curve Profiles.** This command allows you to program a positioning with a limited jerk. In an S-curve mode, the acceleration profile is trapezoidal and the speed profile is like an S-curve.
- **PT** The command allows to program a positioning with path described through a set of points, for each point you specify
- **PVT.** This command allows you to program a positioning described through a series of points, each point includes the desired position, the speed and the time at which the position is to be reached. The user points are interpolated using third order polynomials.
- **External.** This command allows you to set the drives working with an external reference provided by another device.
- **Electronic Gearing.** This command dialogue allows you to set the drives as master or a slave for electronic gearing mode.
- **Electronic Camming.** This command dialogue allows you to set the drives as master or a slave for electronic camming mode.
- **Motor Commands.** This command allows you to apply one of following commands to the motor: activate/deactivate the control loops and the power stage PWM output commands (AXISON / AXISOFF), stop the motor with acceleration/deceleration set, change the value of the motor position and position reference
- **Position Triggers.** This command opens the dialogue where you define the triggering values for each trip point.
- **Homing**
- **Contouring.** This command allows you to program an arbitrary profile whose contour is described by a succession of linear segments

-
- **Test.** This command dialogue allows you to set up the drives in a special test configuration.
 - **Events.** This command allows you to define an event (a condition) to be monitored and to perform several actions.
 - **Jumps and Function Calls.** This command allows you program the operations related with the control of the program flow.
 - **I/O.** This command allows you program operations with the digital inputs and outputs of the drives
 - **Assignment & Data Transfer**
 - **16-bit Integer Data.** This command helps you to program an assignment operation through which you can set the value of a 16-bit variable or set a memory location with a 16-bit immediate value or the value of a 16-bit variable.
 - **32-bit Long or Fixed Data.** This command helps you to program an assignment operation through which you can set the value of a 32-bit variable, set the low part (16LSB) or the high part (16MSB) of a 32-bit variable with a 16-bit value / variable value, set a memory location with a 32-bit immediate value or the value of a 32-bit variable.
 - **Arithmetic Operations.** This command helps you to program one of the arithmetic operations accepted by the MPL (ElectroCraft Motion Program Language): addition, subtraction, product or shifting.
 - **Data Transfer Between Axes.** This command helps you to program the data transfer operations between drives that are connected in a network.
 - **Send Data to Host.** This command allows you to choose what information is sent by the drive automatically. You can send the status register (low part - SRL and high part - SRH), error register (MER) or the value of a variable.
 - **Miscellaneous.** This command opens the dialogue from where you can declare new variables and insert FAULTR, END, NOP, SPI and SCI instructions.
 - **Interrupt Settings.** This command allows you to activate and/or deactivate the MPL interrupts.
 - **Free text.** This command opens a dialogue where you can freely insert a sequence of MPL instructions in the current position in the Motion Editor window.




Edit. Pressing this button, the dialogue associated with the selected motion sequence opens, allowing changing the motion parameters.





Duplicate. Duplicate the selected motion sequence.




Move Down. Moves down the selected motion sequence.


 **Move Up.** Moves up the selected motion sequence.


 **Delete.** Delete the selected motion sequence.


 **Group.** The button allows you to group the selected motion sequences in a new object containing all the selected motion objects. You can give a name or title to the grouped object. This embedding process can be performed in consecutive steps. Any grouped object is displayed with a leading [+] symbol. Click on the [+] symbol to expand the grouped object content to the next embedding level. The leading [+] symbol transforms into a leading [-] symbol. Click the [-] symbol to group back the expended object. Successive embedded levels are accepted.

 **Ungroup.** Use the “Ungroup” command to restore the motion objects list instead of the group object.

 **Enable.** For debugging, you have the possibility to remove motion sequences (one or more motion objects) from the motion program like commenting lines in a text program. Use the the “Enable” button to uncomment / enable motion sequences.

 **Disable.** For debugging, you have the possibility to remove motion sequences (one or more motion objects) from the motion program like commenting lines in a text program. Use the “Disable” button to comment / disable motion sequences.

 **Import.** Use the “Import” button to load/insert motion objects previously saved in *.msq files. These are appended below the current position e.g. the immediately after the selected motion object.

 **Export.** You can select a part of your program (one or more motion objects) and save it in a separate motion file, using the "Export" button. The operation saves the selected motion objects in a file with extension *.msq.

See also:

[Motion programming Toolbars for Multi-axis motion controller](#)

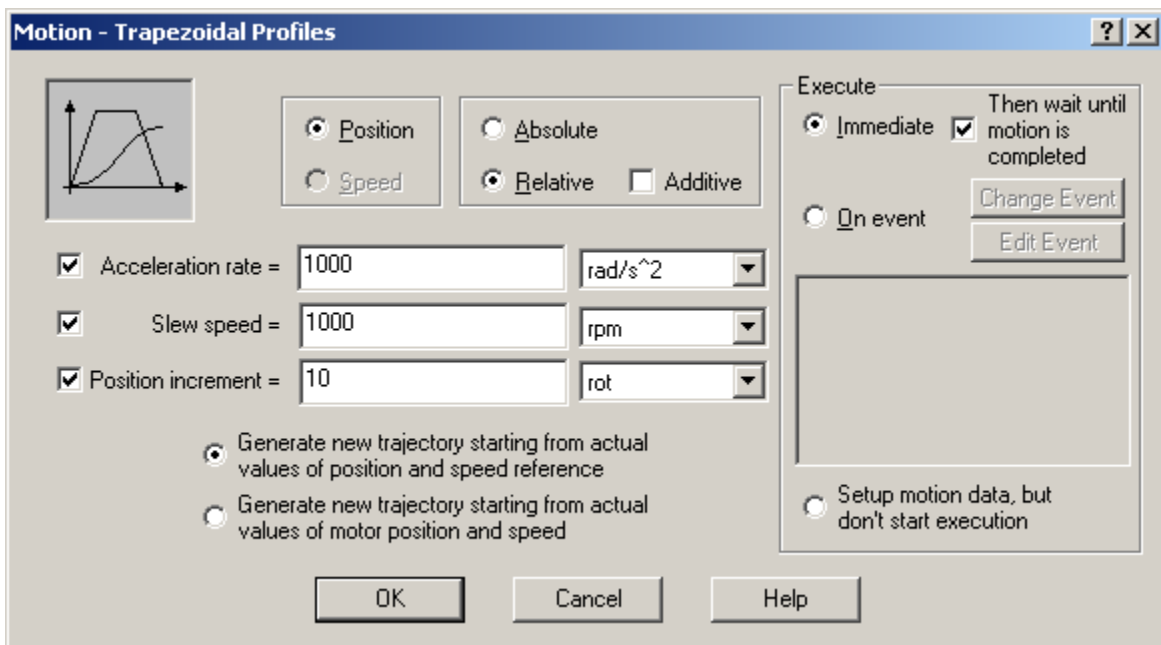
6.1.2. Motion Trapezoidal Profile

The “Motion – Trapezoidal Profiles” dialogue allows you to program a *position* or *speed profile* with a *trapezoidal* shape of the speed, due to a limited acceleration.

In the *position profile*, the load/motor is controlled in position. You specify either a position to reach in absolute mode or a position increment in relative mode, plus the slew (maximum travel) speed and the acceleration/deceleration rate. In relative mode, the position to reach can be computed in 2 ways: standard (default) or additive. In standard relative mode, the position to reach is computed by adding the position increment to the instantaneous position in the moment when the command is executed. In the additive relative mode, the position to reach is computed by adding the position increment to the previous position to reach, independently of the moment when the command was issued. During motion, you can change on the fly the position command, the slew speed and the acceleration/deceleration rate.

In the *speed profile*, the load/motor is controlled in speed. You specify the jog speed (speed sign specifies the direction) and the acceleration/deceleration rate. The load/motor accelerates until the jog speed is reached. During motion, you can change on the fly the slew speed and the acceleration/deceleration rate.

You can switch at any moment between *position* and *speed profiles* or to any of these from another motion mode.



IMPORTANT: Some setup configurations foresee a transmission ratio between the motor and the load. In these cases, the load position and speed are different from the motor position and speed. The motion parameters refer always to the load trajectory.

Choose **Position** to program a *position profile*. Select positioning mode **Relative** or **Absolute**. For relative positioning, check **Additive** to add the position increment to the position to reach set by the previous motion command. Set the values of the **Acceleration rate** and the **Slew speed**. Select the measuring units from the lists on the right. In the absolute positioning mode, set the value of the **Position to reach**. In the relative positioning, set the value of the **Position increment**.

Remark: The *position profile* option is available only if the drive/motor is setup to perform position control.

Choose **Speed** to program a *speed profile*. Set the values of the **Acceleration rate** and the **Jog speed**. Select the measuring units from the lists on the right.

Remark: *Speed profile option is active if the drive/motor is setup to perform speed control or position control with speed loop closed.*

Once set, the trapezoidal profile parameters are memorized. If you intend to use the same values as previously defined for the acceleration rate, the slew or jog speed, the position increment or position to reach you don't need to set their values again in the following trapezoidal profiles. Use the checkboxes on the left to uncheck those parameters that remain unchanged. When a parameter is unchecked, you don't need to give it a value.

Remark: *The additive mode for relative positioning is not memorized and must be set each time a new additive relative move is set.*

Select **Generate new trajectory starting from actual values of position and speed reference** if you want the reference generator to compute the motion profile starting from the actual values of the position and speed reference. Use this option for example if successive standard relative moves must be executed and the final target position should represent exactly the sum of the individual commands. Select **Generate new trajectory starting from actual values of load/motor position and speed** if you want the reference generator to compute the motion profile starting from the actual values of the load/motor position and speed. When this option is used, at the beginning of each new motion profile, the position and speed reference are updated with the actual values of the load/motor position and speed. Use this option for example if during a motion an external input triggers a stop on a precise position relative to the trigger point. Another situation to use this option is at recovery from an error or any other condition that disables the motor control while the motor is moving. Updating the reference values leads to a "glitch" free recovery because it eliminates the differences that may occur between the actual load/motor position/speed and the last computed position/speed reference (before disabling the motor control).

Remark: *In open loop control of steppers, this option is ignored because there is no position and/or speed feedback.*

Choose **Execute Immediate** to start the programmed motion immediately when the motion sequence is executed. Check **Then wait until motion is completed** if you want to postpone the start of the following motion until this programmed motion is completed.

Remark: *Verify the motion complete condition parameters. If these are incorrectly set, you may never reach the motion complete condition:*

- **[POSOKLIM](#)** – the settle band tolerance, expressed in internal [position units](#)
- **[TONPOSOK](#)** – the stabilize time, expressed in internal [time units](#)
- **[UPGRADE.11](#)**:
 - 1 = uses the above parameters,
 - 0 = sets motion complete when the reference generator has completed the trajectory and has arrived to the commanded position

If these parameters have not been set previously, check their default value. Reset the drive/motor and using the [command interpreter](#) get their value.

Choose **Execute On event** to start this new motion when a programmable event occurs. Click **Change Event** to select the event type or **Edit Event** to modify the parameters of the selected event (see [Events](#) for details). Select **Setup motion data, but don't start execution** if you want only to set the motion parameters without starting the execution.

OK: Close this dialogue and save the motion sequence in your motion sequence list.

Cancel: Close this dialogue without saving the motion sequence in your motion sequence list.

Help: Open this help page.

See also:

[Trapezoidal Position Profiles – MPL Programming Details](#)

[Trapezoidal Speed Profiles – MPL Programming Details](#)

[Trapezoidal Position Profiles – Related MPL Instructions and Data](#)

[Trapezoidal Speed Profiles – Related MPL Instructions and Data](#)

[Motion Programming](#)

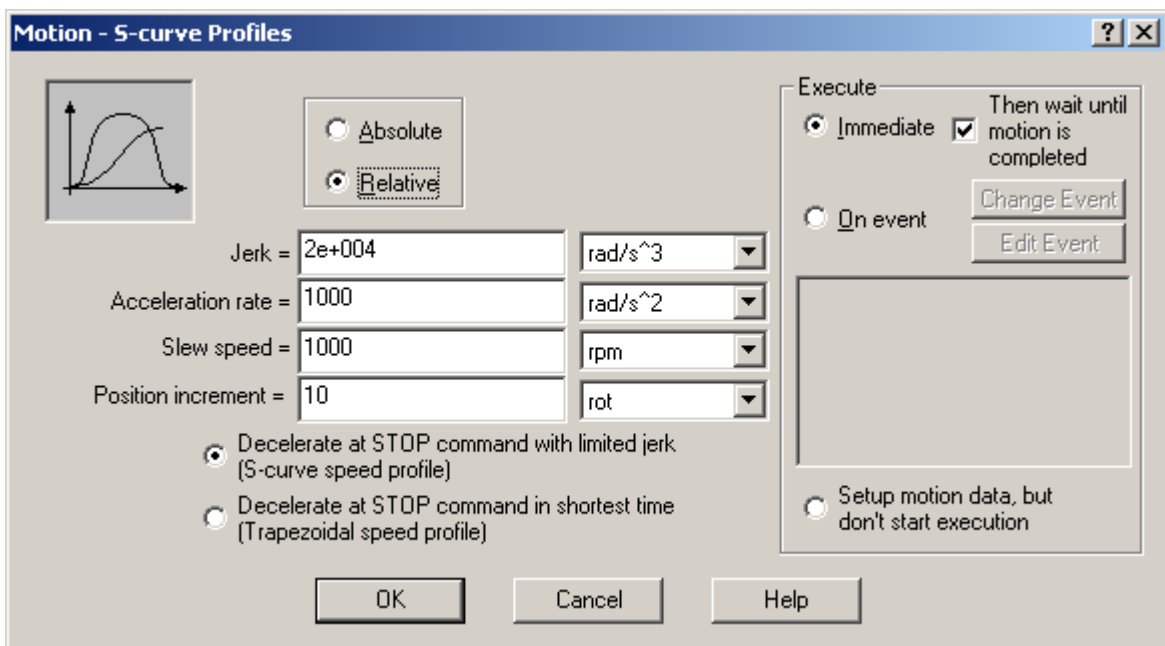
[Internal Units and Scaling Factors](#)

6.1.3. Motion S-Curve Profile

The “Motion – S-curve Profiles” dialogue allows you to program a *position profile* with an *S-curve* shape of the speed. This shape is due to the jerk limitation, leading to a trapezoidal or triangular profile for the acceleration and an S-curve profile for the speed.

In the *S-curve profile*, the load/motor is controlled in position. You specify either a position to reach in absolute mode or a position increment in relative mode, plus the slew (maximum travel) speed, the maximum acceleration/deceleration rate and the jerk rate.

An *S-curve profile* must begin when load/motor is not moving. During motion the parameters should not be changed. Therefore when executing successive S-curve commands, you should wait for the previous motion to end before setting the new motion parameters and starting next motion. During an S-curve execution, you can switch at any moment to another motion mode (except PVT and PT interpolated modes) or stop the motion with a STOP command.



IMPORTANT: Some setup configurations foresee a transmission ratio between the motor and the load. In these cases, the load position and speed are different from the motor position and speed. The motion parameters refer always to the load trajectory.

Choose the option **Relative** to program a relative positioning or **Absolute** for an absolute positioning. Set the values of the **Jerk**, **Acceleration rate** and the **Slew speed**. Select the measuring units from the lists on the right. In the absolute positioning mode, set the value of the **Position to reach**. In the relative positioning, set the value of the **Position increment**.

Remarks:

- The reference generator actually uses the **jerk time** to compute the profile. This is computed as the ratio between the **acceleration rate** and the **jerk rate** you provided and must be a positive integer number, in internal time units. If the **jerk value** is too low, the **jerk time** may be zero. In this case you'll get the error message "**Jerk parameter must be greater than zero!**"
- The S-curve requires the drive/motor to be setup for position control. Otherwise, in the Motion view, the button opening this dialogue will not occur.

Select **Decelerate at STOP command with a limited jerk** if you want a smooth deceleration, using an S-curve speed profile in case of a STOP command. Select **Decelerate at STOP command in shortest time** if you want a faster deceleration, using a trapezoidal speed profile in case of a STOP command.

Choose **Execute Immediate** to start the programmed motion immediately when the motion sequence is executed. Check **Then wait until motion is completed** if you want to postpone the start of the following motion until this programmed motion is completed. If the next motion is an S-curve too, checking this option is mandatory.

Remark: *Verify the motion complete condition parameters. If these are incorrectly set, you may never reach the motion complete condition:*

- [**POSOKLIM**](#) – the settle band tolerance, expressed in internal [**position units**](#)
- [**TONPOSOK**](#) – the stabilize time, expressed in internal [**time units**](#)
- [**UPGRADE.11:**](#)
 - 1 = uses the above parameters,
 - 0 = sets motion complete when the reference generator has completed the trajectory and has arrived to the commanded position

If these parameters have not been set previously, check their default value. Reset the drive/motor and using the [command interpreter](#) get their value.

Choose **Execute On event** to start this new motion when a programmable event occurs. Click **Change Event** to select the event type or **Edit Event** to modify the parameters of the selected event (see [Events](#) for details). Select **Setup motion data, but don't start execution** if you want only to set the motion parameters without starting the execution.

OK: Close this dialogue and save the motion sequence in your motion sequence list.

Cancel: Close this dialogue without saving the motion sequence in your motion sequence list.

Help: Open this help page.

See also:

[S-Curve Profiles – MPL Programming Details](#)

[S-Curve Profiles – Related MPL Instructions and Data](#)

[Motion Programming](#)

[Internal Units and Scaling Factors](#)

6.1.4. Motion PT

The “Motion – PT” dialogue allows you to program a positioning path described through a series of points. Each point specifies the desired **P**osition and **T**ime, i.e. contains a **PT** data. Between the points the built-in reference generator performs a linear interpolation.

In the PT mode the load/motor is controlled in position. A PT sequence must begin when load/motor is not moving.

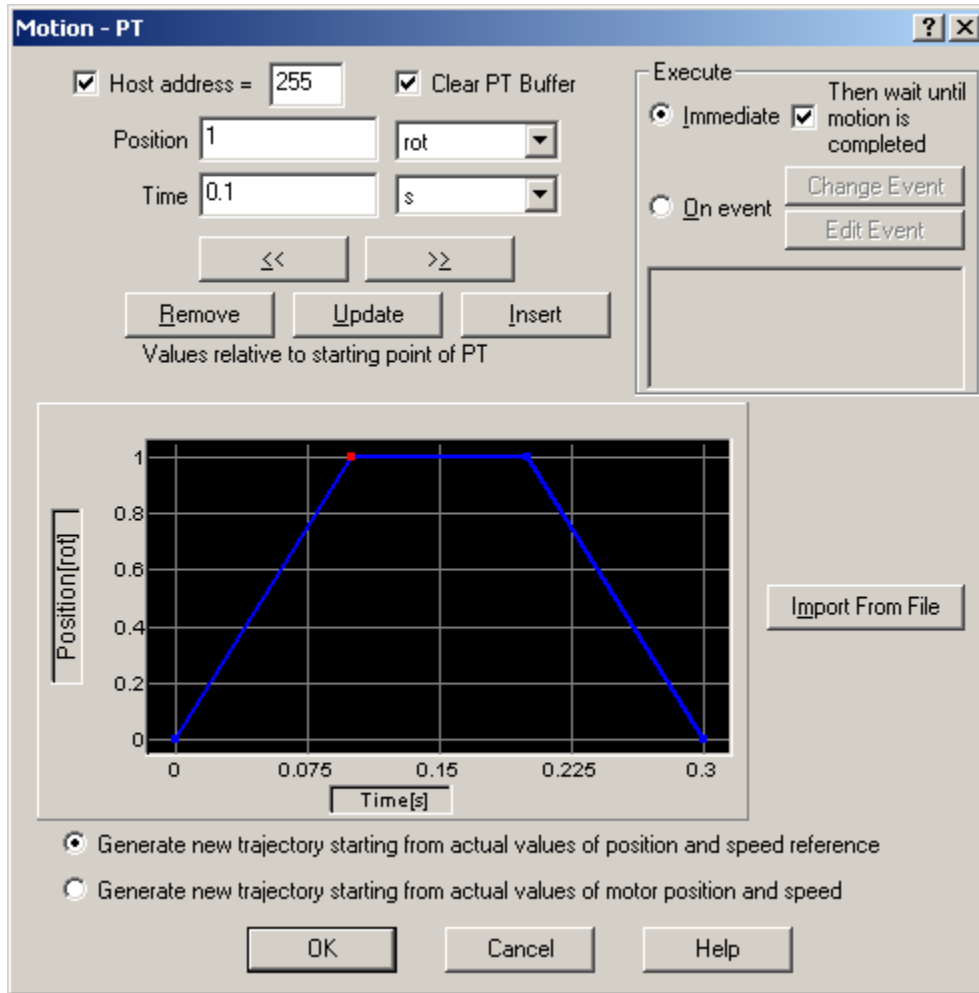
The PT mode is typically used together with a host, which sends PT points via a communication channel. Due to the interpolation, the PT mode offers the possibility to describe arbitrary position contours using a reduced number of points. It is particularly useful when the motion reference is computed on the fly by the host, like for example, in vision systems. By reducing the number of points, both the computation power and the communication bandwidth needed are substantially reduced optimizing the costs. When the PT motion mode is used simultaneously with several drives/motors having the time synchronization mechanism activated, the result is a very powerful multi-axis system that can execute complex synchronized moves.

Upon reception, each PT point is stored in a reception buffer. The reference generator empties the buffer as the PT points are executed. The drive/motor automatically sends warning messages when the buffer is full, low or empty. The buffer full condition occurs when the number of PT points in the buffer is equal with the buffer size. The buffer low condition occurs when the number of PT points in the buffer is less or equal with a programmable value. The buffer empty condition occurs when the buffer is empty and the execution of the last PT point is over.

Remarks:

- *The PT buffer size is programmable and if needed can be substantially increased. By default it is set to 7 PT points.*
- *The buffer low condition is set by default when the last PT point from the buffer is read and starts to be executed*
- *After the execution of the last PT point from a sequence the drive/motor keeps the last reference position, waiting for the next PT commands.*
- *The PT mode requires the drive/motor to be setup for position control. Otherwise, in the Motion view, the button opening this dialogue will not occur.*

The “Motion – PT” dialogue was specifically created to help you quickly evaluate, in advance, a PT sequence of points. The included graphical plot shows you the interpolated trajectory allowing you to check the results. Moreover, you can execute the whole sequence of PT points and check your application behavior before implementing the PT handshake on your host.



You can introduce the PT points in 2 ways:

- One by one, by setting for each point its **Position** and **Time** values. Both are relative to the beginning of the PT motion. Select the measuring units from the list on the right. The graphical tool included, will automatically update the evolution of the position after each point change. A red spot, indicates the *active point*. Use buttons: **Remove**, **Update**, **Insert**, **<<** and **>>** to navigate between the PT points and modify them.
- With **Import From File** to insert a set of PT points previously defined. The file format is a simple text with 2 columns separated by space or tabs representing from left to right: position and time values. The number of rows gives the number of PT points

Check **Host address** and set your PC/host address if the drive/motor is connected via CANbus with your host. The host address is where the PT messages regarding buffer status are sent.

Remark: By default, the host address is initialized with the same value as the drive/motor address, plus the host bit set. This causes to send the PT messages via RS-232 link.

Check **Clear PT Buffer** to erase all the previously stored points from the PT buffer. Use this option each time when you initiate a new PT motion. Uncheck this option if the execution of the PT points was interrupted and you want to resume the execution of the remaining points.

Select **Generate new trajectory starting from actual values of position and speed reference** if you want the reference generator to compute the PT motion path starting from the actual value of the position

reference (the speed reference is always considered zero). Select **Generate new trajectory starting from actual values of load/motor position and speed** if you want the reference generator to compute the PVT motion starting from the actual value of the load/motor position. When this option is used, the position and speed reference are updated with the actual values of the load/motor position and speed. Use this option for example at recovery from an error or any other condition that disables the motor control while the motor is moving. Updating the reference values leads to a “glitch” free recovery because it eliminates the differences that may occur between the actual load/motor position/speed and the last computed position/speed reference (before disabling the motor control).

Remark: *In open loop control of steppers, this option is ignored because there is no position and/or speed feedback.*

Choose **Execute Immediate** to start the programmed motion immediately when the motion sequence is executed. Check **Then wait until motion is completed** if you want to postpone the start of the following motion until this programmed motion is completed.

Remark: *Verify the motion complete condition parameters. If these are incorrectly set, you may never reach the motion complete condition:*

- **POSOKLIM** – the settle band tolerance, expressed in internal **position units**
- **TONPOSOK** – the stabilize time, expressed in internal **time units**
- **UPGRADE.11:**
 - 1 = uses the above parameters,
 - 0 = sets motion complete when the reference generator has completed the trajectory and has arrived to the commanded position

If these parameters have not been set previously, check their default value. Reset the drive/motor and using the [command interpreter](#) get their value.

Choose **Execute On event** to start this new motion when a programmable event occurs. Click **Change Event** to select the event type or **Edit Event** to modify the parameters of the selected event (see [Events](#) for details).

OK: Close this dialogue and save the motion sequence in your motion sequence list.

Cancel: Close this dialogue without saving the motion sequence in your motion sequence list.

Help: Open this help page.

See also:

[PT – MPL Programming Details](#)

[PT – Related MPL Instructions and Data](#)

[Motion Programming](#)

[Internal Units and Scaling Factors](#)

6.1.5. Motion PVT

The “Motion – PVT” dialogue allows you to program a positioning path described through a series of points. Each point specifies the desired **P**osition, **V**elocity and **T**ime, i.e. contains a **PVT** data. Between the points the built-in reference generator performs a 3rd order interpolation.

In the PVT mode the load/motor is controlled in position. A PVT sequence must begin when load/motor is not moving and must end with a last PVT point having velocity zero.

The PVT mode is typically used together with a host, which sends PVT points via a communication channel. Due to the 3rd order interpolation, the PVT mode offers the possibility to describe complex position contours using a reduced number of points. It is particularly useful when the motion reference is computed on the fly by the host, like for example, in vision systems. By reducing the number of points, both the computation power and the communication bandwidth needed are substantially reduced optimizing the costs. When the PVT motion mode is used simultaneously with several drives/motors having the time synchronization mechanism activated, the result is a very powerful multi-axis system that can execute complex synchronized moves.

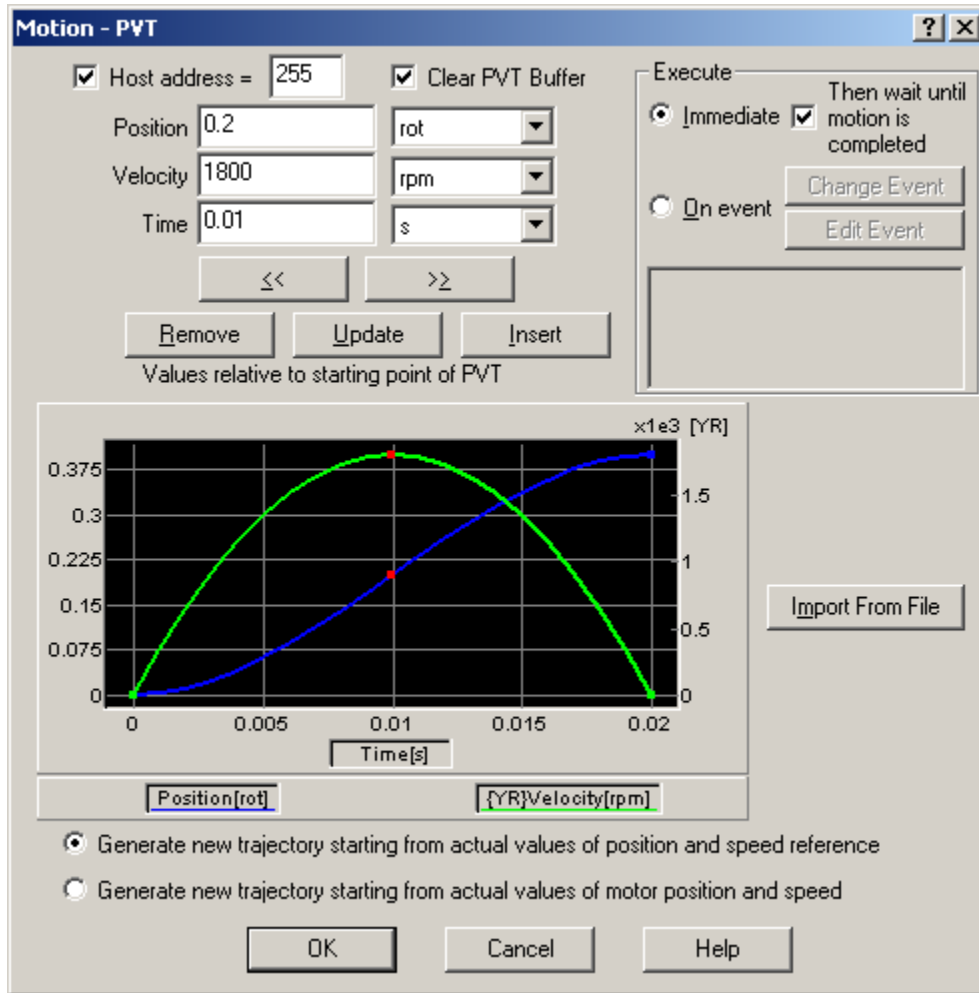
Upon reception, each PVT point is stored in a reception buffer. The reference generator empties the buffer as the PVT points are executed. The drive/motor automatically sends warning messages when the buffer is full, low or empty. The buffer full condition occurs when the number of PVT points in the buffer is equal with the buffer size. The buffer low condition occurs when the number of PVT points in the buffer is less or equal with a programmable value. The buffer empty condition occurs when the buffer is empty and the execution of the last PVT point is over.

Remarks:

- *The PVT buffer size is programmable and if needed can be substantially increased. By default it is set to 7 PVT points.*
- *The buffer low condition is set by default when the last PVT point from the buffer is read and starts to be executed*
- *The normal end of a PVT sequence means: buffer empty condition and velocity zero of the last PVT point executed. If the velocity is not zero, the drive/motor enters in quick stop mode and stops using the quick stop deceleration rate.*
- *The PVT mode requires the drive/motor to be setup for position control. Otherwise, in the Motion view, the button opening this dialogue will not occur.*

When PVT mode is used, a key factor for getting a correct positioning path is to set correctly the distance in time between the points. Typically this is 10-20ms, the shorter the better. If the distance in time between the PVT points is too big, the 3rd order interpolation may lead to important variations compared with the desired path.

The “Motion – PVT” dialogue was specifically created to help you quickly evaluate, in advance, the results of the 3rd order interpolation applied to your data. The included graphical plot shows you the interpolation results for both position and speed reference allowing to check if with the data provided the results are correct. Moreover, you can execute the whole sequence of PVT points and check your application behavior before implementing the PVT handshake on your host.



You can introduce the PVT points in 2 ways:

- One by one, by setting for each point its **Position**, **Velocity** and **Time** values. Both Position and Time values are relative to the beginning of the PVT motion. Select the measuring units from the list on the right. The graphical tool included, will automatically update the evolution of the position and speed after each point change. A red spot, indicates the *active point*. Use buttons: **Remove**, **Update**, **Insert**, **<<** and **>>** to navigate between the PVT points and modify them.
- With **Import From File** to insert a set of PVT points previously defined. The file format is a simple text with 3 columns separated by space or tabs representing from left to right: position, velocity and time values. The number of rows gives the number of PVT points

Check **Host address** and set your PC/host address if the drive/motor is connected via CANbus with your host. The host address is where the PVT messages regarding buffer status are sent.

Remark: By default, the host address is initialized with the same value as the drive/motor address, plus the host bit set. This causes to send the PVT messages via RS-232 link.

Check **Clear PVT Buffer** to erase all the previously stored points from the PVT buffer. Use this option each time when you initiate a new PVT motion. Uncheck this option if the execution of the PVT points was interrupted and you want to resume the execution of the remaining points.

Select **Generate new trajectory starting from actual values of position and speed reference** if you want the reference generator to compute the PVT motion path starting from the actual value of the

position reference (the speed reference is always considered zero). Select **Generate new trajectory starting from actual values of load/motor position and speed** if you want the reference generator to compute the PVT motion starting from the actual value of the load/motor position. When this option is used, the position and speed reference are updated with the actual values of the load/motor position and speed. Use this option for example at recovery from an error or any other condition that disables the motor control while the motor is moving. Updating the reference values leads to a “glitch” free recovery because it eliminates the differences that may occur between the actual load/motor position/speed and the last computed position/speed reference (before disabling the motor control).

Remark: *In open loop control of steppers, this option is ignored because there is no position and/or speed feedback.*

Choose **Execute Immediate** to start the programmed motion immediately when the motion sequence is executed. Check **Then wait until motion is completed** if you want to postpone the start of the following motion until this programmed motion is completed.

Remark: *Verify the motion complete condition parameters. If these are incorrectly set, you may never reach the motion complete condition:*

- **POSOKLIM** – the settle band tolerance, expressed in internal **position units**
- **TONPOSOK** – the stabilize time, expressed in internal **time units**
- **UPGRADE.11**
 - 1 = uses the above parameters,
 - 0 = sets motion complete when the reference generator has completed the trajectory and has arrived to the commanded position

If these parameters have not been set previously, check their default value. Reset the drive/motor and using the [command interpreter](#) get their value.

Choose **Execute On event** to start this new motion when a programmable event occurs. Click **Change Event** to select the event type or **Edit Event** to modify the parameters of the selected event (see [Events](#) for details). Select **Setup motion data, but don't start execution** if you want only to set the motion parameters without starting the execution.

OK: Close this dialogue and save the motion sequence in your motion sequence list.

Cancel: Close this dialogue without saving the motion sequence in your motion sequence list.

Help: Open this help page.

See also:

[PVT – MPL Programming Details](#)

[PVT – Related MPL Instructions and Data](#)

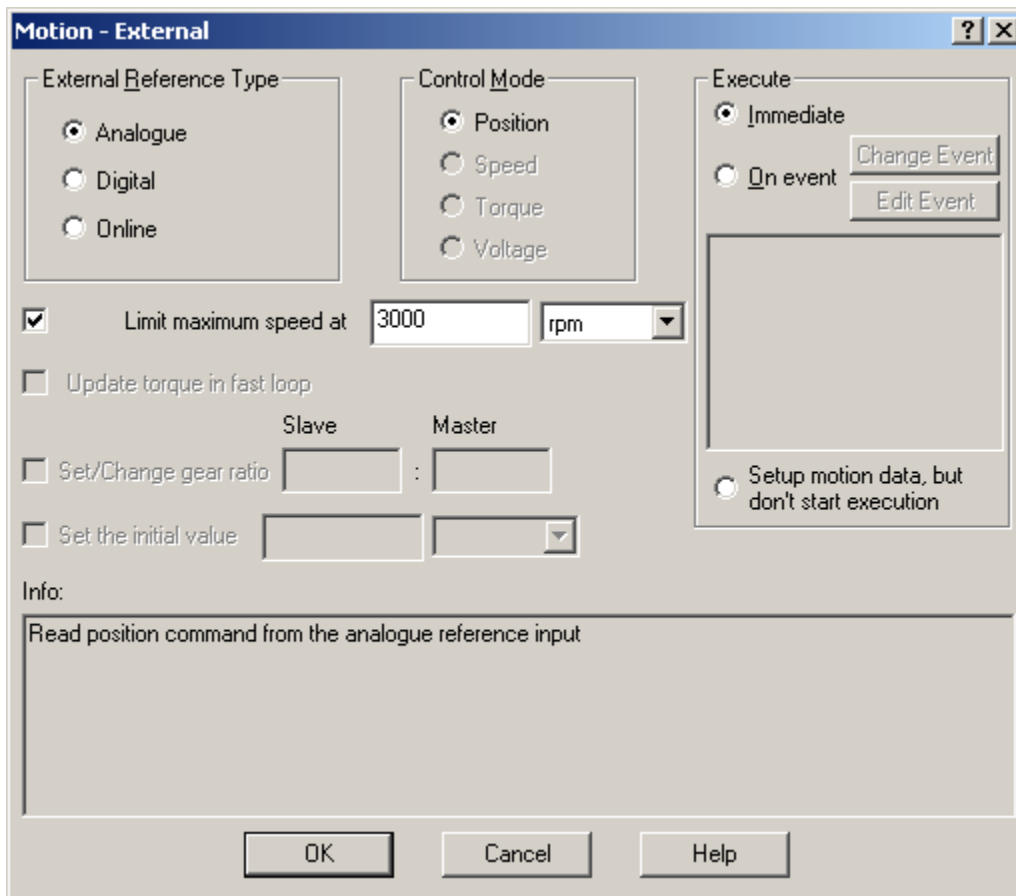
[Motion Programming](#)

[Internal Units and Scaling Factors](#)

6.1.6. Motion External

The “Motion - External” dialogue allows you to program the drives/motors to work with an external reference provided by another device. There are 3 types of external references:

- Analogue – read by the drive/motor via a dedicated analogue input (10-bit resolution)
- Digital – computed by the drive/motor from:
 - Pulse & direction signals
 - Quadrature signals like A, B signals of an incremental encoder
- Online – received online via a communication channel from a host and saved in a dedicated MPL variable



Select **Analogue** if the external reference is an analogue signal. This signal is interpreted as a:

- Position reference, if the drive/motor was setup for position control
- Speed reference, if the drive/motor was setup for speed control
- Current/torque reference, if the drive/motor was setup for torque control

Remark: Check the drive/motor setup for the correspondence between the analogue input voltage and the reference values.

In position control, check **Limit maximum speed at** and set a desired value, if you want to avoid mechanical shocks by limiting the maximum speed at sudden changes of the position reference. In speed

control, check **Limit maximum acceleration at** and set a desired value, if you want a smoother transition at sudden changes of the speed reference. In torque control, check **Update torque in fast loop** if you want to read the analogue input at each fast loop sampling period. When unchecked, the analogue input is read at each slow loop sampling period.

Select **Digital** if the external reference is provided as pulse & direction or quadrature encoder signals. In either case, the drive/motor performs a position control with the reference computed from the external signals. Check **Set/Change gear ratio** if you want to follow the external position reference with a different ratio than 1:1. Set the desired **Slave / Master** ratio.

Remarks:

- *A 1:3 ratio means that the actual position reference TPOS is 1/3 of the external reference.*
- *Due to an automatic compensation procedure, the actual position reference is computed correctly without cumulating errors, even if the ratio is an irrational number like 1: 3*

Select **Online** if an external device sends the reference via a communication channel. Depending on the **Control Mode** chosen, the external reference is saved in one of the MPL variables:

- EREFP, which becomes the position reference if the **Control Mode** selected is **Position**
- EREFS, which becomes the speed reference if the **Control Mode** selected is **Speed**
- EREFT, which becomes the torque reference if the **Control Mode** selected is **Torque**
- EREFV, which becomes voltage reference if the **Control Mode** selected is **Voltage**

If the external device starts sending the reference AFTER the external online mode is activated, it may be necessary to initialize EREFP, EREFS, EREFT or EREFV. Check **Set the initial value** to set the desired starting value.

Remarks:

- *The external online mode may also be used as a test mode in which you assign in EREFP, EREFS, EREFT or EREFV the desired reference*
- *Use external online voltage mode with caution. If the motor is moving, an abrupt reduction of the voltage reference may lead to a high peak of regenerated energy injected into the DC supply. Without proper surging capacity, this may cause high over-voltages*

Choose **Execute Immediate** to activate the external reference mode immediately when the motion sequence is encountered. Choose **Execute On Event** to activate the external reference when a programmable event occurs. Click **Change Event** to select the event type or **Edit Event** to modify the parameters of the selected event (see [Events](#) for details). Select **Setup motion data, but don't start execution** if you want to set the external reference mode parameters for a later use.

OK: Close this dialogue and save the motion sequence in your motion sequence list.

Cancel: Close this dialogue without saving the motion sequence in your motion sequence list.

Help: Open this help page.

See also:

[External –MPL Programming Details](#)

[External –MPL Instructions and Data](#)

[Motion Programming](#)

[Internal Units and Scaling Factors](#)

6.1.7. Motion Electronic Gearing

The “Motion – Electronic Gearing” dialogue allows you to set a drive/motor as **master** or a **slave** for *electronic gearing* mode.

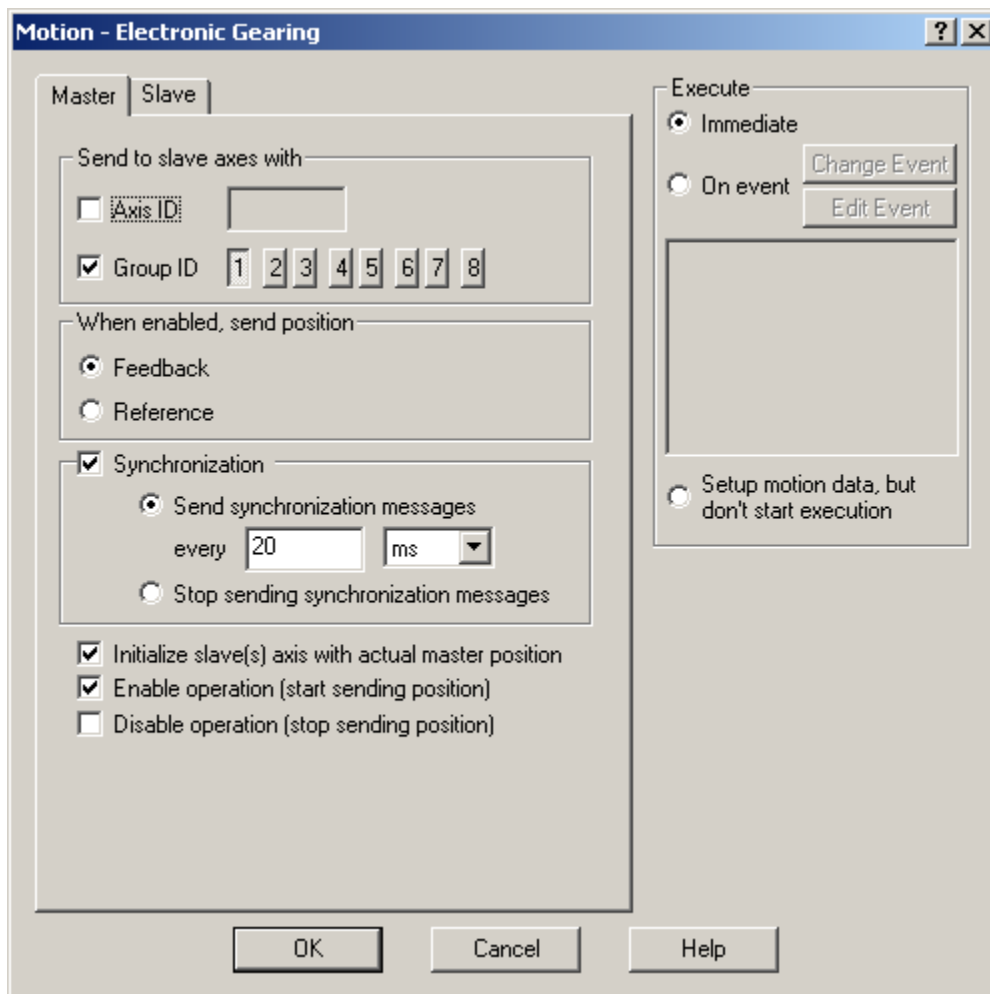
When set as **master**, a drive/motor sends its position via a multi-axis communication channel, like the CANbus. The master sends either the load position or the position reference once at each slow loop sampling time interval.

When set as **slave**, a drive/motor follows the master position with a programmable ratio. The slaves can get the master position in two ways:

1. Via a communication channel, from a drive/motor set as master
2. Via an external digital reference of type pulse & direction or quadrature encoder. Both options have dedicated inputs. The pulse & direction signals are usually provided by an indexer and must be connected to the pulse & direction inputs of the drive/motor. The quadrature encoder signals are usually provided by an encoder on the master and must be connected to the 2nd encoder inputs.

Remark: In case 2, you don't need to program a drive/motor as master in *electronic gearing*

Select **Master** tab to set a drive/motor as master in *electronic gearing*.



If the master sends its position to a single drive/motor, check the **Axis ID** and fill the associated field with the axis ID of the slave. If the master sends its position to more drives, indicate the **Group ID** of the slaves. Select one groups of drives (1 to 8) to which the master should send its position.

Remark: *You need to specify the Axis ID or the Group ID where master sends its position only the first time (after power on) when a drive is set as master. If the master mode is later on disabled, then enabled again, there is no need to set again the Axis ID or the Group ID, as long as they remain unchanged. In this case, just uncheck both the Axis ID and the Group ID.*

Select **Feedback**, to set the master sending its load position, or **Reference**, for sending its position reference.

Remark: *The feedback option is disabled if the master operates in open loop. It is meaningless if the master drive has no position sensor.*

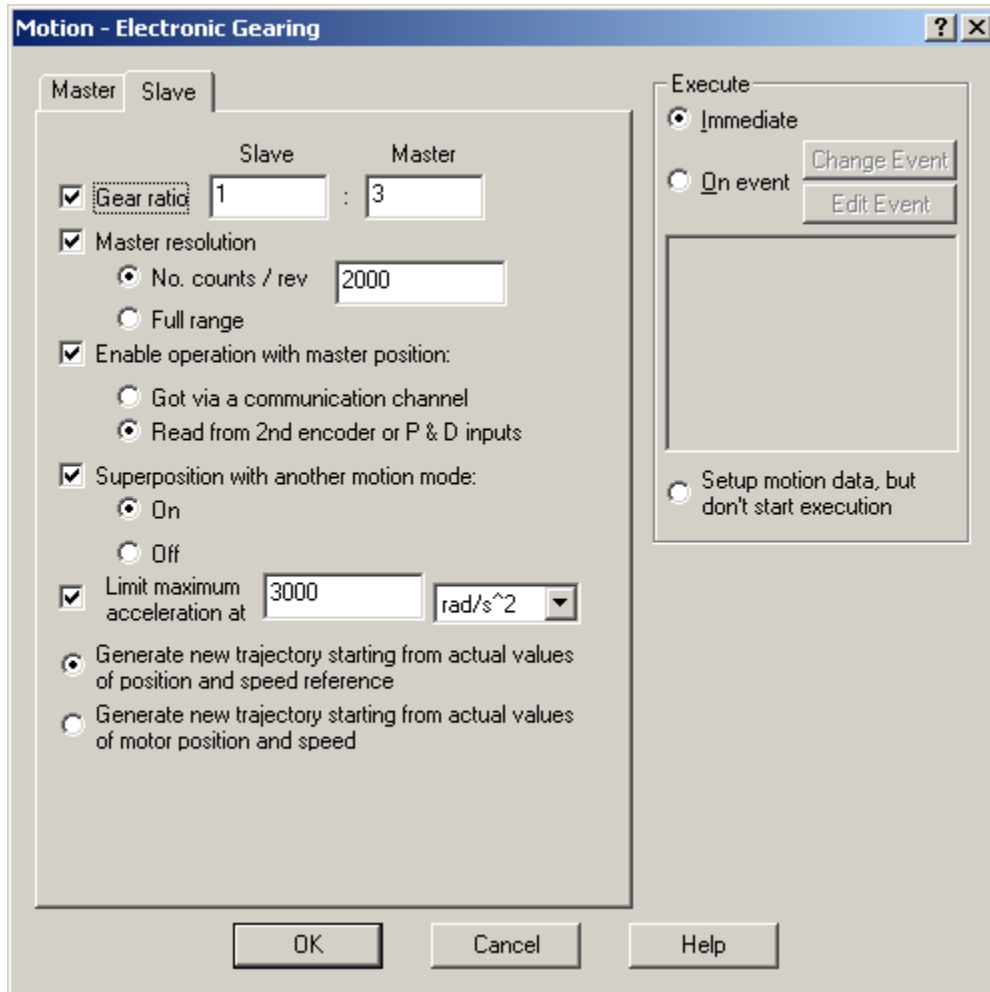
Check **Synchronization** to activate the synchronization procedure between the master and the slave axes. Select **Send synchronization messages** and set the time interval between synchronization messages. Recommended starting value is 20ms. When synchronization procedure is active, the execution of the control loops on the slaves is synchronized with those of the master within a 10 μ s time interval. Due to this powerful feature, drifts between master and slave axes are eliminated. Select **Don't send synchronization** to disable the synchronization procedure.

If the master activation is done AFTER the slaves are set in electronic gearing mode, check **Initialize slave(s) axis with master position**. This determines the master to send an initialization message to the slaves.

Check **Enable operation** to activate the master mode and start the sending of master position to the slaves. Check **Disable operation** to deactivate the master mode and stop sending of master position to the slaves. Note that enabling or disabling master operation has no effect on the motion executed by the master.

Choose **Execute Immediate** to enable the slave operation mode immediately when the motion sequence is encountered. Choose **Execute On Event** to start the slave operation mode when a programmable event occurs. Click **Change Event** to select the event type or **Edit Event** to modify the parameters of the selected event (see [Events](#) for details). Select **Setup motion data, but don't start execution** if you want to prepare the slave operation mode for a later execution.

Select **Slave** tab to set a drive/motor as slave in *electronic gearing*.



Check **Gear Ratio** to set/change the gear ratio with which the slave follows the master position. The gear ratio is specified as a ratio of 2 integer values: **Slave / Master**. The slave value is signed, while the master one is unsigned. The sign indicates the direction of movement: positive – same as the master, negative – reversed to the master.

Remarks:

- *Slave=1 and Master=3, means that slave does 1/3 of master displacement and its speed is 1/3 of the master speed*
- *Due to an automatic compensation procedure, the slave reference is computed correctly without cumulating errors, even if the ratio is an irrational number like 1: 3*

Check **Master Resolution** to specify the number of encoder counts per one revolution of the master motor. The slaves need the master resolution to compute correctly the master position and speed (i.e. position increment). Select **Full range** if master position is not cyclic (e.g. the resolution is equal with the whole 32-bit range of position). In this case the master resolution is set to value 0x80000001.

Check **Enable operation with master position** and select how to get the master position: via communication or via an external reference. Leave unchecked if you want to set the slave parameters without enabling slave operation mode.

Check **Superposition with other motions** and select **On** or **Off** to enable or disable the superposition of the electronic gearing mode with a second motion mode. When this *superposed mode* is activated, the position reference is computed as the sum of the position references for each of the 2 superposed motions.

You may enable the *superposed mode* at any moment, independently of the activation/deactivation of the electronic gearing slave. If the *superposed mode* is activated during an electronic gearing motion, any subsequent motion mode change is treated as a second move to be superposed over the basic electronic gearing move, instead of replacing it. If the *superposed mode* is activated during another motion mode, a second electronic gearing mode will start using the motion parameters previously set. This move is superposed over the first one. After the first move ends, any other subsequent motion will be added to the electronic gearing.

When you disable the *superposed mode*, the electronic gearing slave move is stopped and the drive/motor executes only the other motion. If you want to remain in the electronic gearing slave mode, set first the electronic gearing slave move and then disable the *superposed mode*.

Check **Limit maximum acceleration at**, to smooth slave coupling with the master, when this operation is done with master running at high speed. This option limits the slave acceleration during coupling to the programmed value.

***Remark:** Bit 12 from the Status Register High is set (SRH.12 = 1), when slave coupling with the master is complete. The same bit is reset to zero if the slave is decoupled from the master. The bit has no significance in other motion modes.*

Select **Generate new trajectory starting from actual values of position and speed reference** if you want the reference generator to compute the slave position starting from the actual values of the position and speed reference. Select **Generate new trajectory starting from actual values of load/motor position and speed** if you want the reference generator to compute the slave position starting from the actual values of the load/motor position and speed.

Choose **Execute Immediate** to enable the slave operation mode immediately when the motion sequence is encountered. Choose **Execute On Event** to start the slave operation mode when a programmable event occurs. Click **Change Event** to select the event type or **Edit Event** to modify the parameters of the selected event (see [Events](#) for details). Select **Setup motion data, but don't start execution** if you want to prepare the slave operation mode for a later execution.

OK: Close this dialogue and save the motion sequence in your motion sequence list.

Cancel: Close this dialogue without saving the motion sequence in your motion sequence list.

Help: Open this help page

See also:

[Electronic Gearing – MPL Programming Details](#)

[Electronic Gearing – MPL Instruction and Data](#)

[Motion Programming](#)

[Internal Units and Scaling Factors](#)

6.1.8. Motion Electronic Camming

The “Motion – Electronic Camming” dialogue allows you to set a drive/motor as **master** or **slave** for *electronic camming* mode.

When set as **master**, a drive/motor sends its position via a multi-axis communication channel, like the CAN bus. The master sends either the load position or the position reference once at each slow loop sampling time interval.

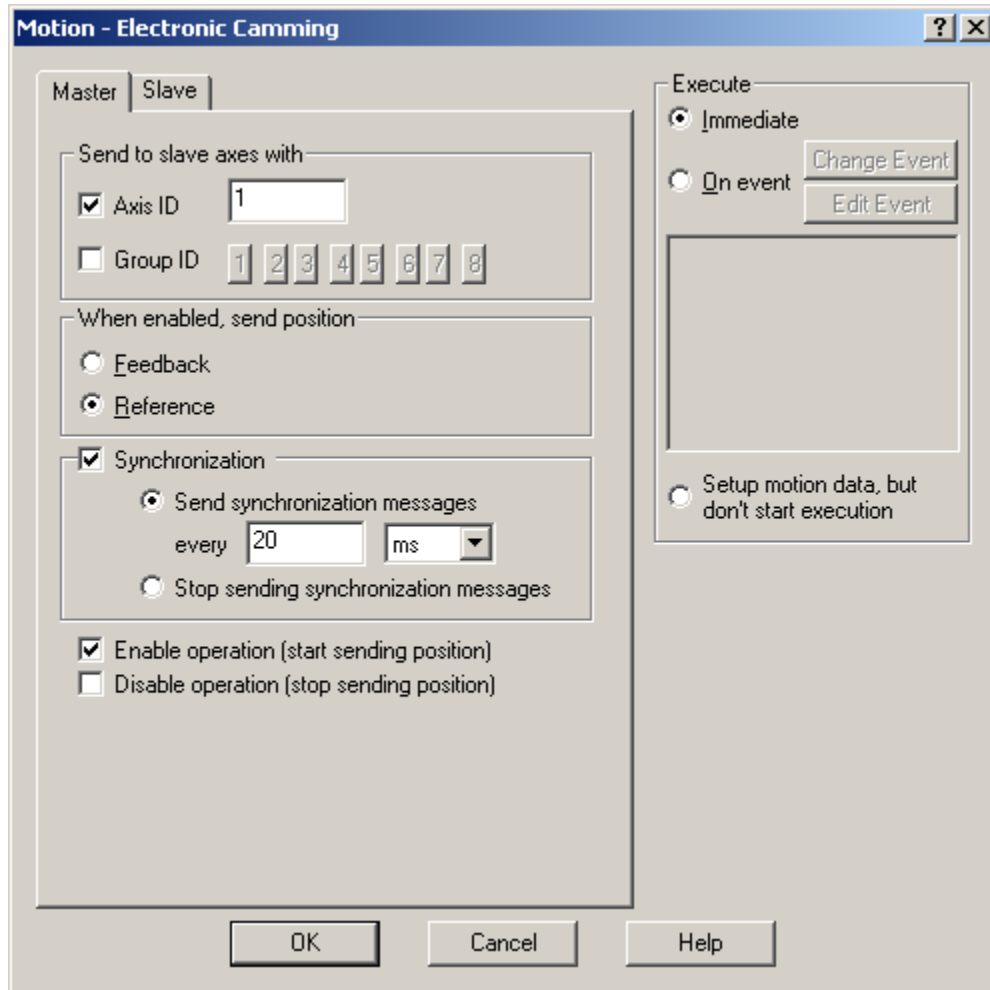
When set as **slave**, a drive/motor executes a cam profile function of the master position. The cam profile is defined by a cam table – a set of (X, Y) points, where X is cam table input i.e. the master position and Y is the cam table output i.e. the corresponding slave position. Between the points the drive/motor performs a linear interpolation. Using [Cam Tables Selection](#) selection you can associate cam tables to your application. These may be visualized and modified using the [Cam Tables Edit](#). You may also import cam tables. The required format is: text file with 2 columns, one for X, and the other for Y, separated by space or tab. Data must be in internal units.

The slaves can get the master position in two ways:

1. Via a communication channel, from a drive/motor set as master
2. Via an external digital reference of type pulse & direction or quadrature encoder. Both options have dedicated inputs. The pulse & direction signals are usually provided by an indexer and must be connected to the pulse & direction inputs of the drive/motor. The quadrature encoder signals are usually provided by an encoder on the master and must be connected to the 2nd encoder inputs.

Remark: For 2nd option you don't need to program a drive/motor as master in *electronic camming*

Select **Master** tab to set a drive/motor as master in *electronic camming*.



If the master sends its position to a single drive/motor, check the **Axis ID** and fill the associated field with the axis ID of the slave. If the master sends its position to more drives, indicate the **Group ID** of the slaves. Select one group of drives (1 to 8) to which the master should send its position.

Remark: You need to specify the Axis ID or the Group ID where master sends its position only the first time (after power on) when a drive is set as master. If the master mode is later on disabled, then enabled again, there is no need to set again the Axis ID or the Group ID, as long as they remain unchanged. In this case, just uncheck both the Axis ID and the Group ID.

Select **Feedback**, to set the master sending its load position, or **Reference**, for sending its position reference.

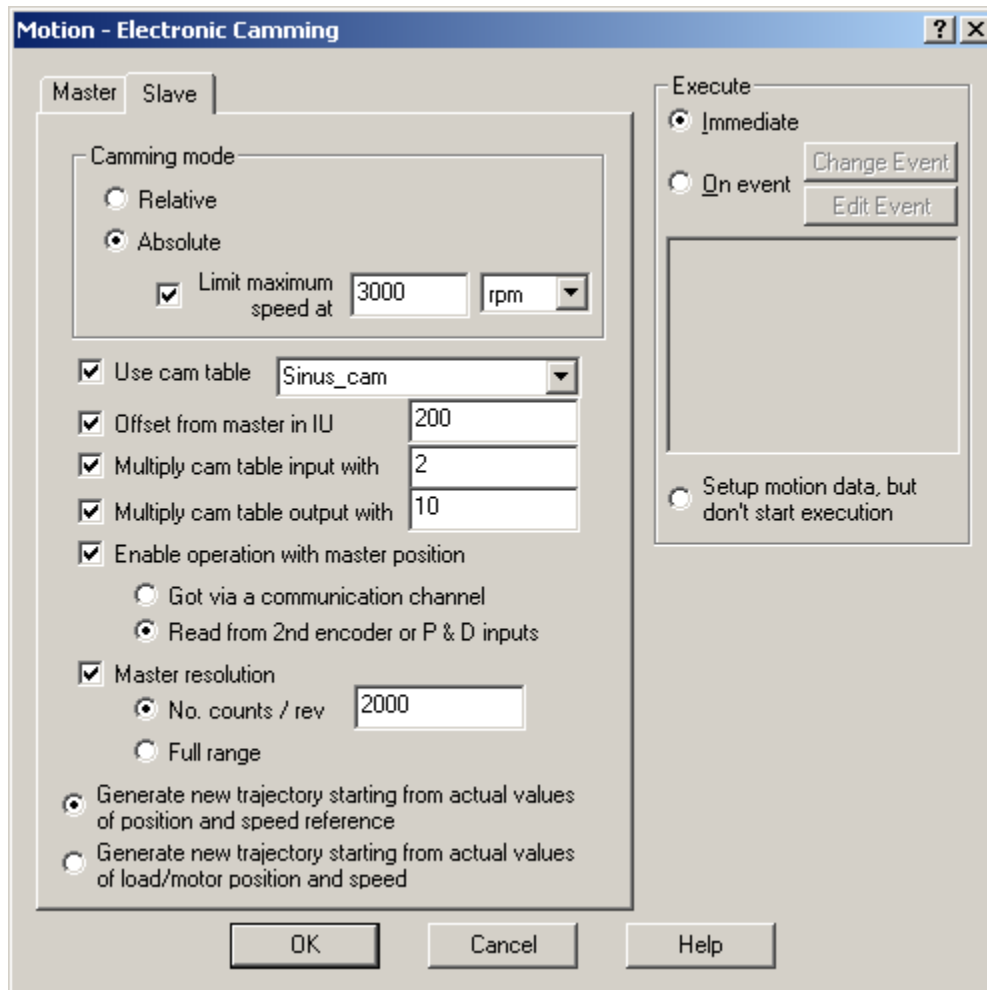
Remark: The feedback option is disabled if the master operates in open loop. It is meaningless if the master drive has no position sensor.

Check **Synchronization** to activate the synchronization procedure between the master and the slave axes. Select **Send synchronization messages** and set the time interval between synchronization messages. Recommended starting value is 20ms. When synchronization procedure is active, the execution of the control loops on the slaves is synchronized with those of the master within a 10 μ s time interval. Due to this powerful feature, drifts between master and slave axes are eliminated. Select **Don't send synchronization** to disable the synchronization procedure.

Check **Enable operation** to activate the master mode and start the sending of master position to the slaves. Check **Disable operation** to deactivate the master mode and stop sending of master position to the slaves. Note that enabling or disabling master operation has no effect on the motion executed by the master.

Choose **Execute Immediate** to enable the slave operation mode immediately when the motion sequence is encountered. Choose **Execute On Event** to start the slave operation mode when a programmable event occurs. Click **Change Event** to select the event type or **Edit Event** to modify the parameters of the selected event (see [Events](#) for details). Select **Setup motion data, but don't start execution** if you want to prepare the slave operation mode for a later execution.

Select **Slave** tab to set a drive/motor as slave in *electronic camming*.



Select the camming mode:

- In **Relative** mode, the output of the cam table represents for the slave a position increment, which is added to its actual position
- In **Absolute** mode, the output of the cam table represents for the slave the position to reach.

Remark: The absolute mode may generate abrupt variations on the slave position reference, mainly at entry in the camming mode. Check **Limit maximum speed at** to limit the speed of the slave during travel towards the position to reach.

Check **Use CAM table** and choose between the selected cam tables which one to use.

Remark: Note that at runtime, all the selected cam tables are loaded into the drive memory. If needed, you may switch between the cam tables loaded. This operation means just to change the value of the **CAMSTART** parameter which points towards the active cam table.

Check **Offset from master in IU** to shift the cam profile versus the master position, by setting a cam offset for each slave. The cam table input is computed as the master position minus the cam offset. For example, if a cam table is defined between angles 100 to 250 degrees, a cam offset of 50 degrees will make the cam table to execute between master angles 150 and 300 degrees.

Check **Multiply table input with** to compress/extend a cam table input. Specify the input correction factor by which the cam table input is multiplied. For example, an input correction factor of 2, combined with a cam offset of 180 degrees, will make possible to execute a cam table defined for 360 degrees of the master in the last 180 degrees.

Check **Multiply table output with** in order to compress/extend a cam table output. Specify the output correction factor by which the cam table output is multiplied. This feature addresses the applications where the slaves must execute different position commands at each master cycle, all having the same profile defined through a cam table. In this case, the drive/motor is programmed with a unique normalized cam profile and the cam table output is multiplied with the relative position command updated at each master cycle.

Check **Enable operation with master position** and select how to get the master position: via communication or via an external reference. Leave unchecked if you want to set the slave parameters without enabling slave operation mode.

Check **Master Resolution** to specify the number of encoder counts per one revolution of the master motor. The slaves need the master resolution to compute correctly the master position and speed (i.e. position increment). Select **Full range** if master position is not cyclic (e.g. the resolution is equal with the whole 32-bit range of position). In this case the master resolution is set to value 0x80000001.

Select **Generate new trajectory starting from actual values of position and speed reference** if you want the reference generator to compute the slave position starting from the actual values of the position and speed reference. Select **Generate new trajectory starting from actual values of load/motor position and speed** if you want the reference generator to compute the slave position starting from the actual values of the motor position and speed.

Choose **Execute Immediate** to enable the slave operation mode immediately when the motion sequence is encountered. Choose **Execute On Event** to start the slave operation mode when a programmable event occurs. Click **Change Event** to select the event type or **Edit Event** to modify the parameters of the selected event (see [Events](#) for details). Select **Setup motion data, but don't start execution** if you want to prepare the slave operation mode for a later execution.

OK: Close this dialogue and save the motion sequence in your motion sequence list.

Cancel: Close this dialogue without saving the motion sequence in your motion sequence list.

Help: Open this help page

See also:

[Cam Tables Selection](#)

[Cam Tables Edit](#)

[Electronic Camming – MPL Programming details](#)

[Electronic Camming –MPL Instruction and Data](#)

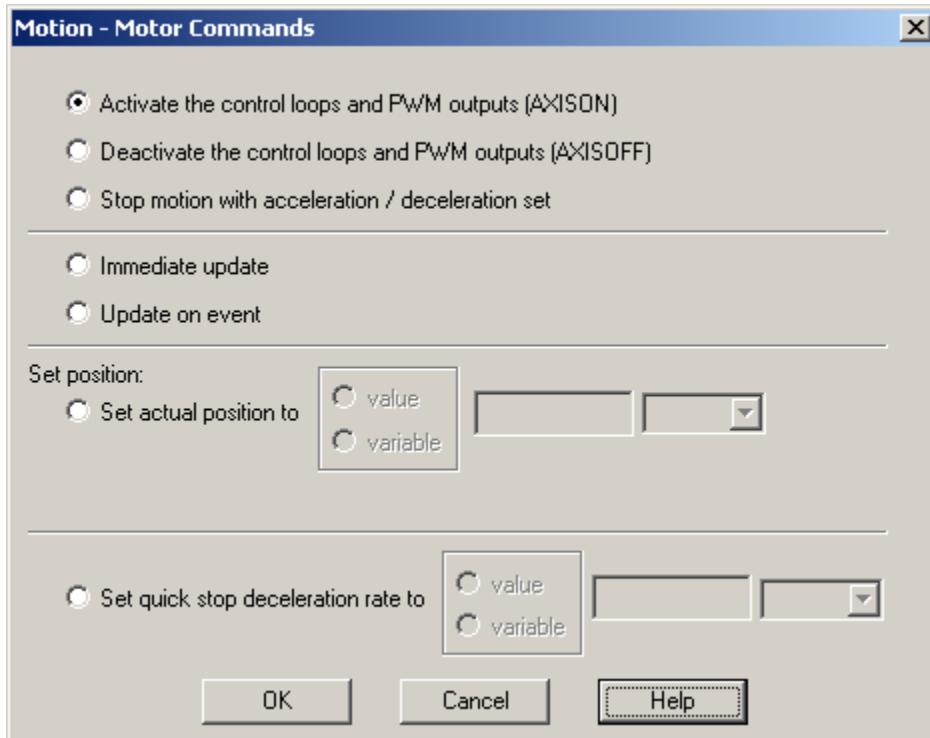
[Motion Programming](#)

[Internal Units and Scaling Factors](#)

6.1.9. Motor Commands

The “Motion - Motor Commands” dialogue allows you to apply one of following commands to the motor:

- Activate/deactivate the control loops and the power stage PWM output commands (**AXISON** / **AXISOFF**)
- Stop the motor with deceleration set in MPL parameter **CACC**
- Change the value of the motor position and position reference
- Set deceleration rate for quick stops



Select **Activate the control loops and PWM outputs (AXISON)** to restore normal drive operation after an **AXISOFF** command. Typically, this situation occurs at recovery from an error, following the fault reset command **FAULTR**, or after the drive/motor **ENABLE** input goes from status disabled to status enabled.

Select **Deactivate the control loops and PWM outputs (AXISOFF)** when a fault condition is detected, for example when a protection is triggered. This command disables the motor control (all the control loops), all the PWM output commands for the power stage (all the switching devices are off) and also the reference generator.

Fault conditions trigger MPL interrupts. Each drive/motor has a built-in set of MPL interrupt service routines (ISR) which are automatically activated after power-on. In these routines, the default action for fault conditions is an **AXISOFF** command. If needed, you may replace any built-in ISR with your own ISR and thus, adapt the fault treatment to your needs.

After a fault condition, the actual values of the load position and speed (which continue to be measured during the **AXISOFF** condition) may differ quite a lot from the values of the target position and speed as were last computed by the reference generator before entering in the **AXISOFF** condition. Therefore, a correct fault recovery sequence involves the following steps:

-
- Set the motion mode, even if it is the same. Motion mode commands, automatically set the target update mode zero (**TUMO**), which updates the target position and speed with the actual measured values of the load position and speed
 - Execute update command UPD
 - Execute **AXISON** command

Remark:

- In the Drive Status control panel, **SRL.15** shows the **AXISON/AXISOFF** condition and **SRH.15** shows a fault condition
- In MotionPRO Developer, **ENDINIT** and **AXISON** commands are automatically included in the MPL program, just before your first MPL command from the main section. Therefore you don't need to include them in your motion program.

Select **STOP** to stop the motor with the deceleration rate set in MPL parameter **CACC**. The drive/motor decelerates following a trapezoidal position or speed profile. If the **STOP** command is issued during the execution of an S-curve profile, the deceleration profile may be chosen between a trapezoidal or an S-curve profile (see S-curve dialogue settings). You can detect when the motor has stopped by setting a motion complete event and waiting until the event occurs. The **STOP** command can be used only when the drive/motor is controlled in position or speed.

Remarks:

- In order to restart after a **STOP** command, you need to set again the motion mode. This operation disables the stop mode and allows the motor to move
- When **STOP** command is sent via a communication channel, it will automatically stop any MPL program execution, to avoid overwriting the **STOP** command from the MPL program

Choose **Immediate Update** to generate an update command **UPD**. When this command is received, the last motion mode programmed together with the latest motion parameters are taken into consideration. The immediate update command is available in all the dialogues setting a motion mode and normally it is called from these dialogues. The immediate update command is useful when the motion mode is set in advance for a later execution, which is started with a separate update command. In a similar way you may use **Update on event**.

You can set / change the referential for position measurement by changing simultaneously the load position **APOS** and the target position **TPOS** values, while keeping the same position error any moment during motion. Use the edit field from **set actual position value** to specify the new motor position value.

Remark: In the case of steppers controlled in open loop, this command changes only the target position **TPOS** to the desired value.

The deceleration rate for quick stops can be set/change selecting the option **Set quick stop deceleration rates**. To assign an immediate value select option **value** and fill the associated field, if you want to assign the value of a variable select then **variable** and in the associated field write the name of the variable.

OK: Close this dialogue and save the motion sequence in your motion sequence list.

Cancel: Close this dialogue without saving the motion sequence in your motion sequence list.

Help: Open this help page.

See also:

[Motor Commands – MPL Programming Details](#)

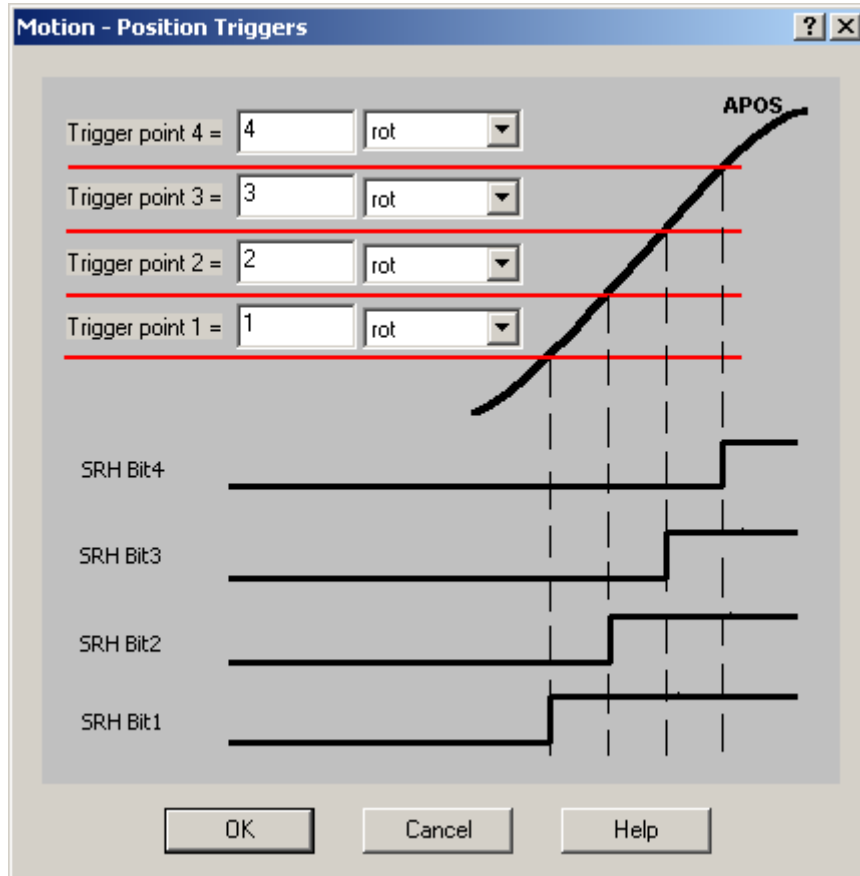
[Motor Commands – MPL Instructions and Data](#)

[Motion Programming](#)

[Internal Units and Scaling Factors](#)

6.1.10. Motion Position Triggers

The “Motion - Position Triggers” dialogue allows you to define 4 *position trigger* points. A *position trigger* is a position value with which the actual position is continuously compared. The compare result is shown in the Status Register High (**SRH**). If the actual position is below a position trigger, the corresponding bit from SRH is set to 0, else it is set to 1. You can change at any moment the value of a position trigger.



The actual position that is compared with the position triggers is:

- The **Load position feedback** (MPL variable **APOS_LD**) for configurations with position sensor
- The **position reference** (MPL variable **TPOS** – Target position) in the case of steppers controlled in open-loop

Remark: *The position triggers can be used to monitor the motion progress. If this operation is done from a host, you may program the drive/motor to automatically issue a message towards the host, each time when the status of a position trigger is changed.*

See also:

[Position Triggers – MPL Programming Details](#)

[Position Triggers – Related MPL Instructions and Data](#)

[Motion Programming](#)

[Internal Units and Scaling Factors](#)

6.1.11. Motion Homing

The “Motion – Homing” dialogue allows you choose a homing procedure and set its parameters. The homing is a sequence of motions, usually executed after power-on, through which the load is positioned into a well-defined point – the home position. Typically, the home position is the starting point for normal operation.

The search for the home position can be done in numerous ways. Therefore, a lot of homing procedures are possible. ElectroCraft provides for each programmable drive/motor a collection of up to 32 homing procedures. These are predefined MPL functions, which you may call after setting the homing parameters. You may use these homing procedures as they are, or you may modify them according with your application needs. From the list with all the [defined homing procedures](#) you can choose one or several to be used in your application. This represents the list of **selected homing procedures**.

Motion - Homing

Select homing parameters

Acceleration rate: 1000 rad/s²

Deceleration rate: 1000 rad/s² (for limit switch stop)

High speed: 1000 rpm

Low speed: 90 rpm

Home position: 0 rot

Read home input in variable:

Execute homing procedure: Homing1

Procedure description: Move negative until the limit switch is reached. Reverse and stop at first index pulse.

OK Cancel Help

Check **Select homing parameters** to set the following values:

- Acceleration/deceleration rate for the position or speed profiles done during homing
- Deceleration rate for quick stop when a limit switch is reached
- High/normal speed for the position or speed profiles done during homing
- Low speed for the final approach towards the home position
- New home position set at the end of the homing procedure

Check **Execute homing mode** and choose a homing procedure from the list of the **selected homing procedures**. During the execution of a homing sequence **SRL.8 = 1**. Hence you can find when a homing sequence ends, either by monitoring bit 8 from SRL or by programming the drive/motor to send a message to your host when **SRL.8** changes. As long as a homing sequence is in execution, you should not start another one. If this happens, the last homing is aborted and a warning is generated by setting **SRL.7 = 1**.

Remark: You can abort a homing sequence execution at any moment using MPL command **ABORT** (see [Decisions](#)).

You can also use this dialogue to read the status of the *home input*. The *home input* is one of the drive/motor inputs, which is used by the homing procedures. The *home input* is specific for each product and based on the setup data, MotionPRO Developer automatically generates the MPL code for reading the correct input. Check **Read home input in the variable** and fill the associated field with the name of the variable. After execution, the value of the variable will be 0 if the home input is zero (low) or 1 if the home input is 1 (high).

Remark: The source of the motion sequence for reading the home input is general and independent. The particular value of the home input, specific for each product, occurs only in the compiled version of this motion sequence, in the MPL code generated. Therefore, you can safely import the source code of this motion sequence into other applications where the target products have different home inputs.

OK: Close this dialogue and save the settings in your motion sequence list.

Cancel: Close this dialogue without saving the settings in your motion sequence list.

Help: Open this help page.

See also:

[Homing – MPL Programming Details](#)

[Homing – Related MPL Instructions and data](#)

[Motion Programming](#)

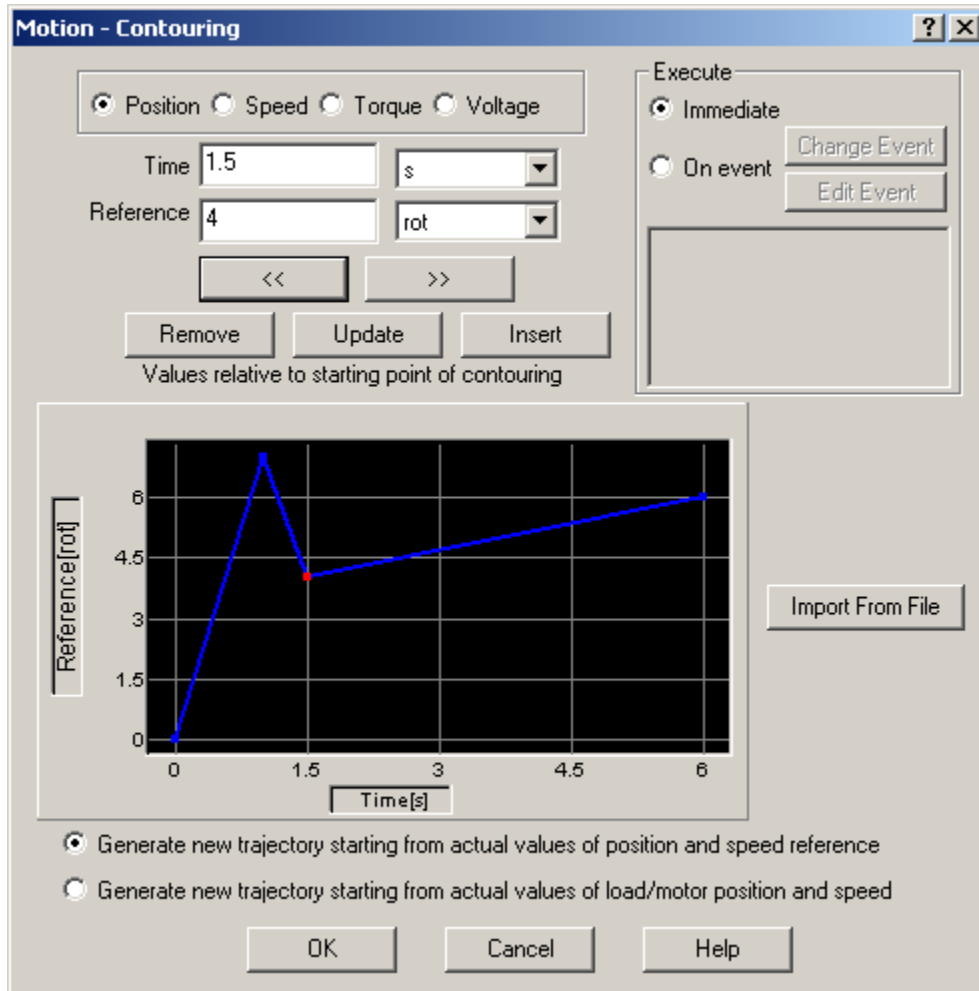
[Internal Units and Scaling Factors](#)

6.1.12. Motion Contouring

The “Motion - Contouring” dialogue allows you to program an arbitrary contour via a series of points. Between the points, linear interpolation is performed, leading to a contour described by a succession of linear segments. The contouring mode may be executed only from a MPL program. You can't send contouring points from a host via a communication channel, like in the case of the PT mode. Depending on the control mode chosen, four options are available:

- *Position contouring* – the load/motor is controlled in position. The path represents a position reference
- *Speed contouring* – the load/motor is controlled in speed. The path represents a speed reference.
- *Torque contouring* – the motor is controlled in torque. The path represents a current reference.
- *Voltage contouring* – the motor is controlled in voltage. The path represents a voltage reference.

Each contour point is defined by 2 values: the reference and the time. The contouring mode has been foreseen mainly for setup tests. However, you can also use the *position contouring* and the *speed contouring* for normal operation, as part of your motion application. You can switch at any moment to and from these 2 modes. The *torque contouring* and the *voltage contouring* have been foreseen only for setup tests. The *torque contouring* may be used, for example, to check the response of the current controllers to different input signals. Similarly, the *voltage contouring* may be used, for example, to check the motors behavior under a constant voltage or any other voltage shape.



Choose

Position for a *position contouring*,

Speed for a *speed contouring*,

Torque for a *torque contouring*

Voltage for a *voltage contouring*.

Remarks:

- *Position contouring option is disabled if the drive/motor is not setup for position control*
- *Speed contouring option is disabled if the drive/motor is not setup for speed control. This includes the case when position control is performed without closing the speed loop*
- *Torque contouring option is disabled for stepper drives working in open loop*

In the *position contouring* and the *speed contouring* the starting point has always the coordinates (0,0) and corresponds to the moment when the contouring mode is activated. **Therefore all the segments values (time and reference) are relative to the starting point of the contouring.** For example, lets suppose that a position contouring sequence has one segment with coordinates (1s, 10 rot) and the absolute position is 20 revolutions (initial position when the position contouring is activated). During the contour segment execution, the motor moves 10 revolutions in 1 second and stops on absolute position 30 revolutions.

In the *torque contouring* and *voltage contouring* the starting point has by default the initial value 0. However, you can also start with a different value, by setting in the first point a non-zero reference at time = 0.

You can introduce the contouring points in 2 ways:

- One by one, by setting for each point its **Time** and **Reference** values. Select the measuring units from the list on the right. The graphical tool included, will automatically update the contour as you introduce each point. A red spot, indicates the *active point*. Use buttons: **Remove**, **Update**, **Insert**, << and >> to navigate between the points and modify them.
- With **Import From File** to insert a set of contouring points previously defined. The file format is a simple text with 2 columns separated by space or tabs representing from left to right: time and reference values. The number of rows gives the number of points

Select **Generate new trajectory starting from actual values of position and speed reference** if you want the reference generator to compute the contour profile starting from the actual values of the position and speed reference. Select **Generate new trajectory starting from actual values of load/motor position and speed** if you want the reference generator to compute the contour profile starting from the actual values of the load/motor position and speed. When this option is used, at the beginning of each new contour profile the position and speed reference is updated with the values of the load/motor position and speed. Use this option for example at recovery from an error or any other condition that disables the motor control while the motor is moving. Updating the reference values leads to a “glitch” free recovery because it eliminates the differences that may occur between the actual load/motor position/speed and the last computed position/speed reference (before disabling the motor control).

Remark: *In open loop control of steppers, this option is ignored because there is no position and/or speed feedback.*

Choose **Execute Immediate** to start the contour profile immediately when the motion sequence is encountered. Choose **Execute On event** to start the motion when a programmable event occurs. Click

Change Event to select the event type or **Edit Event** to modify the parameters of the selected event (see [Events](#) for details).

OK: Close this dialogue and save the motion sequence in your motion sequence list.

Cancel: Close this dialogue without saving the motion sequence in your motion sequence list.

Help: Open this help page.

See also:

[Contouring – MPL Programming details](#)

[Contouring – MPL Instructions and Data](#)

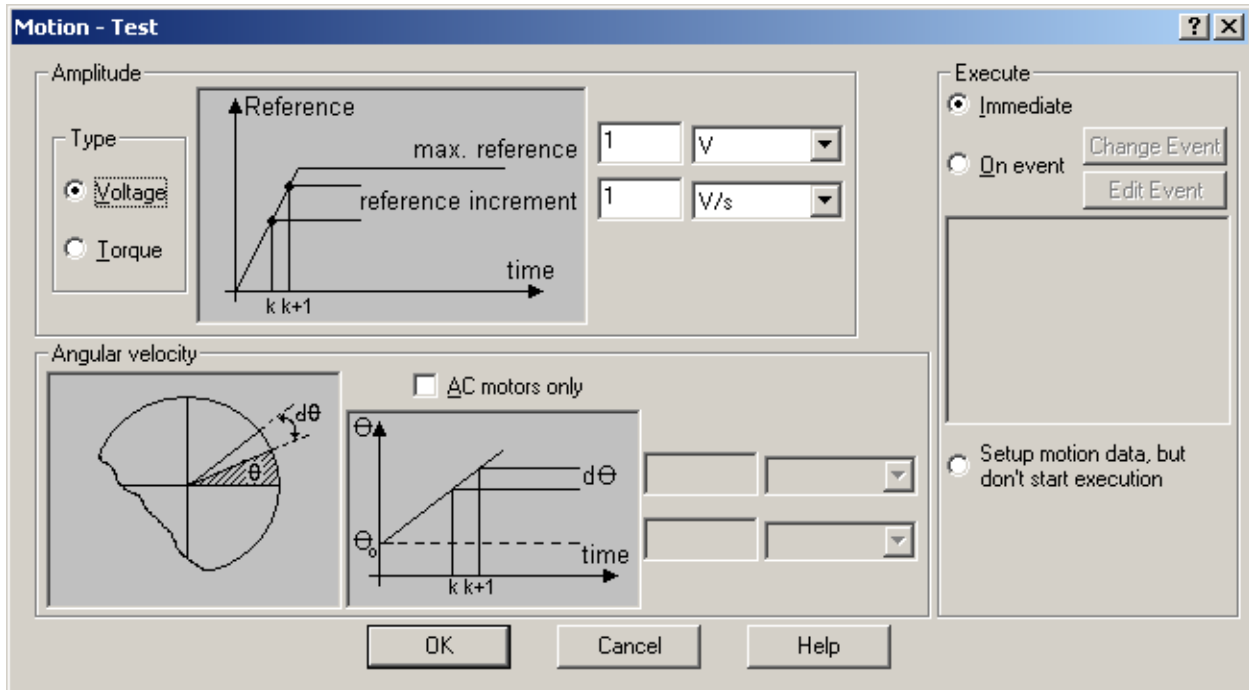
[Motion Programming](#)

[Internal Units and Scaling Factors](#)

6.1.13. Motion Test

The “Motion – Test” dialogue allows you to set the drives/motors in a special test configuration. This configuration is not supposed to be used during normal operation, but only during drive/motor setup.

In the test mode, either a voltage or a torque (current) command can be set using a test reference consisting of a limited ramp. For AC motors (like for example the brushless motors), the test mode offers also the possibility to rotate a voltage or current reference vector with a programmable speed. As a result, these motors can be moved in an “open-loop” mode without using the position sensor. The main advantage of this test mode is the possibility to conduct in a safe way a series of tests, which can offer important information about the motor parameters, drive status and the integrity of the its connections.



Select **Voltage** for voltage reference or **Torque** for torque reference. Insert the appropriate values for reference amplitude and reference increment in the corresponding fields and select the measurement unit.

For AC motors, check the option **AC motor only**. Insert the appropriate values for the reference vector initial position and the electrical angle increment in the corresponding fields and select the measurement unit.

Choose **Execute Immediate** to activate the external reference mode immediately when the motion sequence is encountered. Choose **Execute On Event** to activate the external reference when a programmable event occurs. Click **Change Event** to select the event type or **Edit Event** to modify the parameters of the selected event (see [Events](#) for details). Select **Setup motion data, but don't start execution** if you want to prepare the external reference mode for a later use.

OK: Close this dialogue and save the motion sequence in your motion sequence list.

Cancel: Close this dialogue without saving the motion sequence in your motion sequence list.

Help: Open this help page.

See also:

[Motion Test –MPL Programming details](#)

[Motion Test – Related MPL Instructions and Data](#)

[Motion Programming](#)

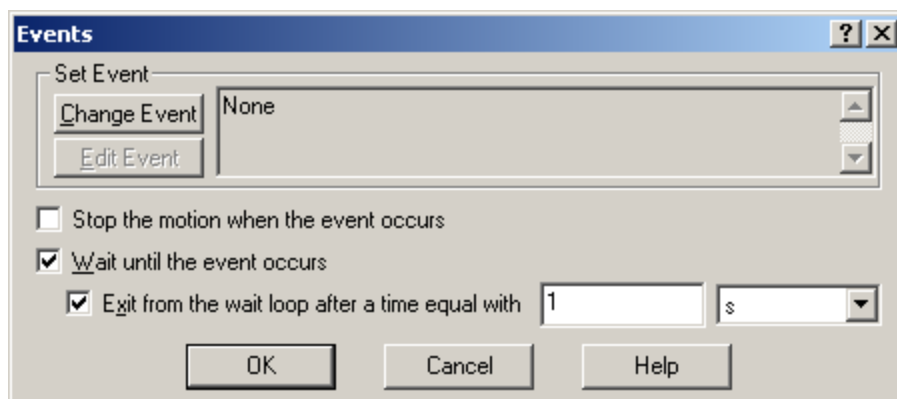
[Internal Units and Scaling Factors](#)

6.1.14. Events Dialogue

The “Events” dialogue allows you to define *events*. An *event* is a programmable condition, which once set, is monitored for occurrence. You can do the following actions in relation with an *event*:

- 1) Change the motion mode and/or the motion parameters, when the event occurs
- 2) Stop the motion when the event occurs
- 3) Wait for the programmed event to occur

Remark: *The programmed event is automatically erased if the event is reached, if the timeout for the wait is reached or if a new event is programmed.*



Only a single event can be programmed at a time. This can be:

- 1) When the actual motion is completed
- 2) When motor absolute position is equal or under a value or the value of a variable
- 3) When motor absolute position is equal or over a value or the value of a variable
- 4) When load absolute position is equal or under a value or the value of a variable
- 5) When load absolute position is equal or over a value or the value of a variable
- 6) When load/motor relative position is equal or under a value or the value of a variable
- 7) When load/motor relative position is equal or over a value or the value of a variable
- 8) When motor speed is equal or under a value or the value of a variable
- 9) When motor speed is equal or over a value or the value of a variable
- 10) When load speed is equal or under a value or the value of a variable
- 11) When load speed is equal or over a value or the value of a variable
- 12) After a wait time equal with a value or the value of a variable
- 13) When position reference is equal or under a value or the value of a variable
- 14) When position reference is equal or over a value or the value of a variable
- 15) When speed reference is equal or under a value or the value of a variable

-
- 16) When speed reference is equal or over a value or the value of a variable
 - 17) When torque reference is equal or under a value or the value of a variable
 - 18) When torque reference is equal or over a value or the value of a variable
 - 19) When 1st or 2nd encoder index goes low or high
 - 20) When the positive limit switch goes low or high
 - 21) When the negative limit switch goes low or high
 - 22) When a digital input goes low
 - 23) When a digital input goes high
 - 24) When a 32-bit variable is equal or under a 32-bit value or the value of another 32-bit variable
 - 25) When a 32-bit variable is equal or over a 32-bit value or the value of another 32-bit variable

Remark: *The load/motor relative position is computed starting from the beginning of the current movement.*

You can also program events in the following *motion dialogues*: [Trapezoidal Profiles](#), [S-curve Profiles](#), [PT](#), [PVT](#), [External](#), [Electronic Gearing](#), [Electronic Camming](#), [Contouring](#), [Test](#). Set events in these dialogues, if you want to activate the programmed motion mode and/or its motion parameters, when the programmed event occurs.

The event programming is done in the same way when it is done from a *motion dialogue* or from this dialogue. Press **Change Event** to open the [Event Selection](#) dialog which allows you to define the event / condition to be monitored. If you have already defined an event, use **Edit Event** button to modify its parameters or conditions.

When you set an event using one of the *motion dialogues*, you program the following operations:

- Definition of an event
- Programming of a new motion mode and/or new motion parameters
- Definition of the moment when the new motion mode and/or motion parameters must be updated (e.g. enabled) as the moment when the programmed event will occur

Remark: *After you have programmed a new motion mode and/or new motion parameters with update on event, you need to introduce a wait until the programmed event occurs. Otherwise, the program will continue with the next instructions that may override the event monitoring. In order to introduce a wait until the programmed event occurs, open this dialogue, select as event **None** and check **Wait until the event occurs**.*

In this dialogue, apart from programming an event, you can **Stop motion when the event occurs** and **Wait until the event occurs** by checking these options. You can also define a time limit for an event to occur. Check **Exit from the wait loop after a time equal with** and specify the time limit. If the monitored event doesn't occur in this time limit, the wait loop is interrupted and the MPL program passes to the next instruction.

Remarks:

- By default, the option **Wait until the event occurs** is checked. Typically, you define an event, than wait for the event to occur.
- If the option **Wait until the event occurs** is checked without a time limit, and the programmed event doesn't occur, the MPL program will remain in a loop. In order to exit from this loop, send via a communication channel a GOTO command, which moves the program execution outside the loop

OK: Close this dialogue and save the event programming in your motion sequence list.

Cancel: Close this dialogue without saving or updating the event programming in the motion sequence list.

Help: Open this help page.

See also:

[Events – MPL Programming Details](#)

[Event Selection](#)

[Motion Programming](#)

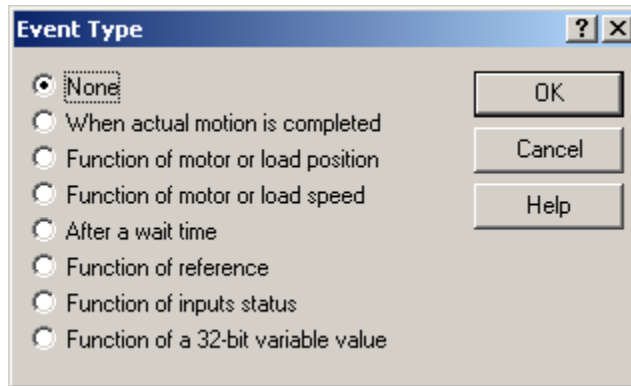
6.1.14.1. Event Type Selection

The “Event Type” dialogue allows you to select an event. An *event* is a programmable condition, which once set, is monitored for occurrence.

The “Event Type” dialogue may be opened from:

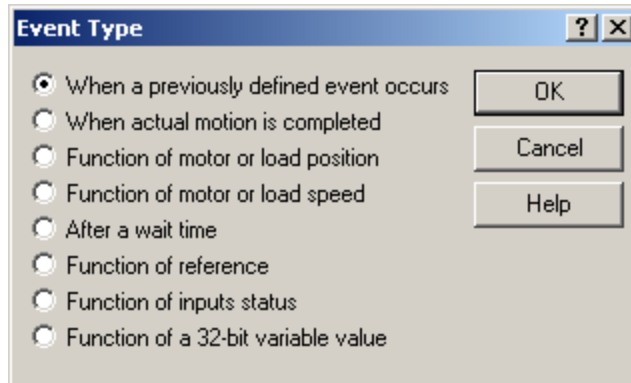
- Events dialogue:

Event Type – called from Events dialogue



- One of the following *motion dialogues*: [Trapezoidal Profiles](#), [S-curve Profiles](#), [PT](#), [PVT](#), [External](#), [Electronic Gearing](#), [Electronic Camming](#), [Contouring](#), [Test](#):

Event Type – called from a motion dialogue



The events are grouped into 8 categories:

None/ When a previously defined event occurs. The meaning of this case depends from where the “Event Type” dialogue was opened:

- **None** – appears when the dialogue is opened from the “Events” dialogue. Check this item if you have already defined an event and now you want to: a) program a stop when the event occurs and/or b) wait for the programmed event to occur.
- **When a previously defined event occurs** – appears when the dialogue is opened from one of the motion dialogues (see above). Check this item if you have already defined an event (in a previous motion sequence) and now you want to start the actual motion sequence when this event occurs.

[When actual motion is completed](#) – for programming the event: when the actual motion is completed.

[Function of motor or load position](#) – for programming the events: when the absolute or relative motor or load position is equal or over/under a value or the value of a variable.

[Function of motor or load speed](#) – for programming the events: when the motor or load speed is equal or over/under a value or the value of a variable.

[After a wait time](#) – for programming a time delay, using a time event. The monitored event is: when the relative time is equal with a value or the value of a variable

[Function of reference](#) – for programming the events: when the position or speed or torque reference is equal or over/under a value or the value of a variable.

[Function of inputs status](#) – for programming the events: when capture inputs or limit switch inputs or general purpose inputs change status: low to high or high to low.

[Function of a variable value](#) – for programming the events: when a selected variable is equal or over/under a value or the value of another variable.

OK: Close this dialogue and save selected event

Cancel: Close this dialogue without saving the selected event

Help: Open this help page.

See also:

[Events](#)

[Motion Programming](#)

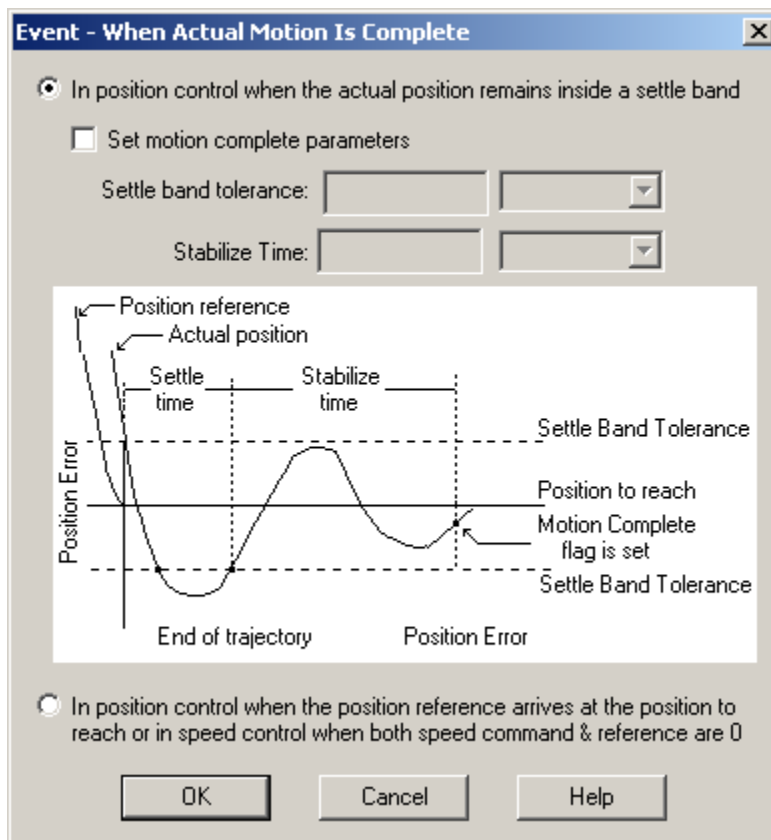
6.1.14.2. Event - When the actual motion is completed

This dialogue allows you to set the event: when a motion is completed. You can use, for example, this event to start the next move only after the actual one is finalized.

The motion complete condition is set in the following conditions:

- During position control:
 - With position feedback – when the position reference arrives at the position to reach (commanded position) and the position error remains inside a *settle band* for a preset *stabilize time* interval
 - Without position feedback (open-loop systems) – when the position reference arrives at the position to reach (commanded position)
- During speed control, when the speed reference arrives at the commanded speed

The motion complete condition is reset when a new motion is started.



In position control, choose **In position control when the actual position remains inside a settle band** for the first option. Check **Set motion complete parameters** if you want to modify the **Settle band tolerance** and the **Stabilize time** values. Select the measuring units from the list on the right. Leave **Set motion complete parameters** unchecked if you want to keep the motion complete parameters unchanged.

Choose **In position control when the position arrives at the position to reach or in speed control when speed command & reference are equal** in:

-
- Speed control
 - Position control with open-loop configurations or if you do not want to use first option

OK: Close this dialogue and save the event set

Cancel: Close this dialogue without saving the event set.

Help: Open this help page.

See also:

[Events – When actual motion is completed– MPL Programming details](#)

[Event Selection](#)

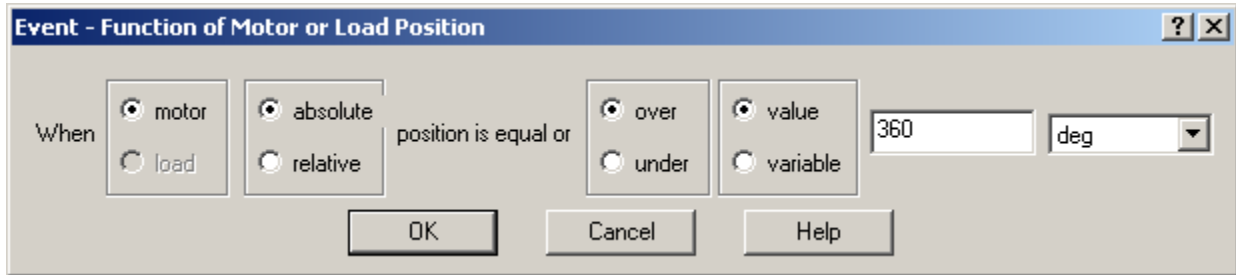
[Events](#)

[Motion Programming](#)

6.1.14.3. Event - Function of motor or load position

This dialogue allows you to program an event function of the motor or load position. The events can be: when the absolute or relative motor or load position is equal or over/under a value or the value of a variable

The absolute load or motor position is the measured position of the load or motor. The relative position is the load displacement from the beginning of the actual movement. For example if a position profile was started with the absolute load position 50 revolutions, when the absolute load position reaches 60 revolutions, the relative motor position is 10 revolutions.



Select **motor** or **load** position, its type: **absolute** or **relative**, the event condition: **over** (or equal) or **under** (or equal) and the comparison data: a **value** or the value of a **variable**.

OK: Close this dialogue and save the event set

Cancel: Close this dialogue without saving the event set.

Help: Open this help page.

See also:

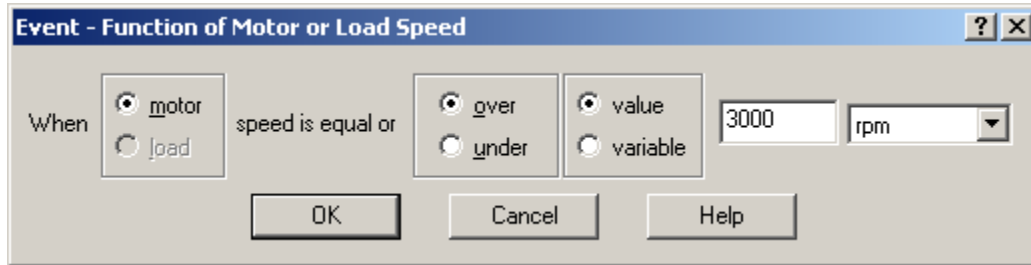
[Event – Function of motor or load position–MPL Programming Details](#)

[Event Selection](#)

[Motion Programming](#)

6.1.14.4. Event - Function of motor or load speed

This dialogue allows you to program an event function of the motor or load speed. The events can be: when the motor or load speed is equal or over/under a value or the value of a variable.



Select **motor** or **load** speed, the event condition: **over** (or equal) or **under** (or equal) and the comparison data: a **value** or the value of a **variable**.

OK: Close this dialogue and save the event set

Cancel: Close this dialogue without saving the event set.

Help: Open this help page.

See also:

[Event – Function of motor or load speed–MPL Programming Details](#)

[Event Selection](#)

[Events](#)

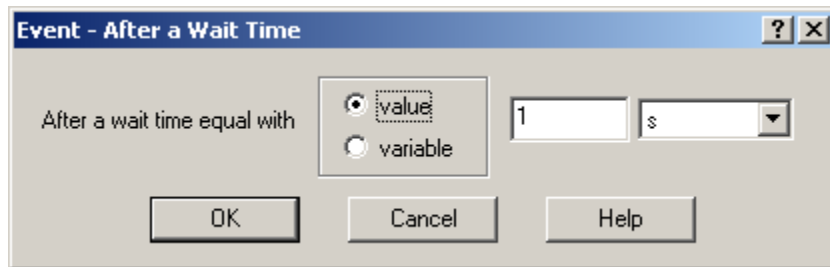
[Motion Programming](#)

6.1.14.5. Event– After a Wait Time

This dialogue allows you to introduce a programmable delay in the motion program execution of the motion controller/drive, using a time event. When you set this event, the motion controller/drive relative time is reset and it starts counting from zero and the monitored condition is: when the relative time is equal with a value or the value of a variable.

Remarks:

- *The event on time can be programmed only for the local axis.*
- *In order to effectively execute the time delay, you need to follow this command by a **Wait until the event occurs** command e.g. until the programmed relative time has elapsed.*



Select the comparison data: a **value** or the value of a **variable**.

OK: Close this dialogue and save the event set

Cancel: Close this dialogue without saving the event set.

Help: Open this help page.

See also:

[Event– After a Wait Time –MPL Programming Details](#)

[Event Selection](#)

[Events](#)

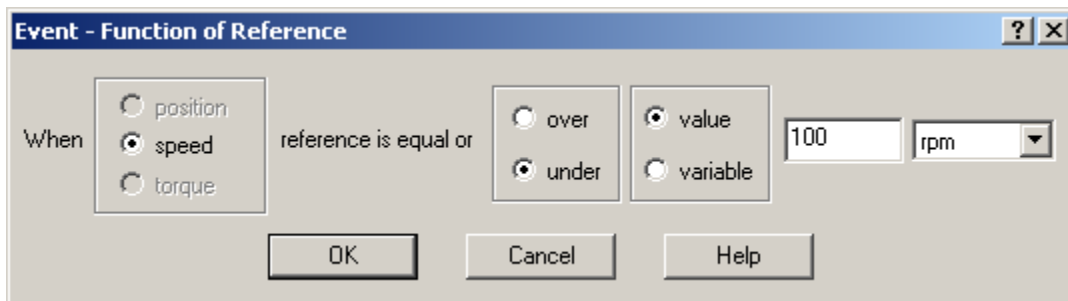
[Motion Programming](#)

6.1.14.6. Event - Function of reference

This dialogue allows you to program an event function of the position or speed or torque reference. The events can be: when the position/speed/torque reference is equal or over/under a value or the value of a variable. Use:

- Position reference events, only when position control is performed
- Speed reference events, only when speed control is performed
- Torque reference events, only when torque control is performed

Remark: Setting an event based on the position or speed reference is particularly useful for open loop operation where feedback position and speed is not available



Select the reference type: **position**, **speed** or **torque**, the event condition: **over** (or equal) or **under** (or equal) and the comparison data: a **value** or the value of a **variable**.

OK: Close this dialogue and save the event set

Cancel: Close this dialogue without saving the event set.

Help: Open this help page.

See also:

[Event – Function of reference –MPL Programming Details](#)

[Event Selection Events](#)

[Motion Programming](#)

6.1.14.7. Event - Function of inputs status

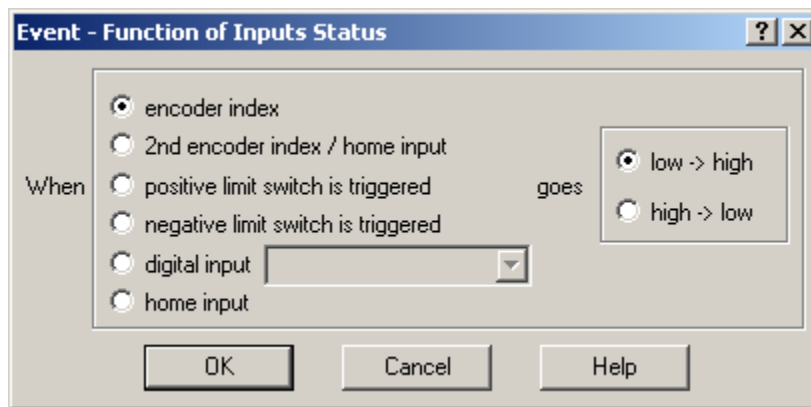
This dialogue allows you to program one of the following events:

- When a transition occurs on one of the 2 capture inputs, where are connected the 1st and 2nd encoder index signals (if available)
- When a transition occurs on one of the 2 limit switch inputs
- When a general purpose digital input changes its status
- When the home input changes its status

The capture inputs and the limit switch inputs can be programmed to sense either a low to high or high to low transition. When the programmed transition occurs on either of these inputs, the following happens:

- Motor position is captured and memorized in the MPL variable **CAPPOS**, except the case of open-loop systems, where the reference position is captured instead
- Master or load position is captured and memorized in the MPL variable **CAPPOS2**, except the case of steppers controlled open-loop with an encoder on the load, when load position is captured in **CAPPOS**.

The selection between master and load position is done as follows: load position is saved in **CAPPOS2** only for the setup configurations which use different sensors for load and motor and foresee a transmission ratio between them. For all the other setup configurations, the master position is saved in **CAPPOS2**. The master position is automatically computed when pulse and direction signals or quadrature encoder signals are connected to their dedicated inputs.



Select:

- **encoder index** to detect a transition on 1st capture/encoder index input
- **2nd encoder index** to detect a transition on 2nd capture/encoder index
- **positive limit switch** to detect a transition on limit switch input for positive direction
- **negative limit switch** to detect a transition on limit switch input for negative direction

and choose the transition type: **low -> high** or **high -> low**

Select **digital input** to set an event on one of the general-purpose digital input available. The event can be set when the input goes **high** or **low**. Select **home input** in order to set an event on the general purpose digital input assigned as *home input*. The *home input* is specific for each product and based on the setup data, MotionPRO Developer automatically generates the MPL code for reading the correct input.

OK: Close this dialogue and save the event set

Cancel: Close this dialogue without saving the event set.

Help: Open this help page.

See also:

[Events – Function of inputs status–MPL Programming Details](#)

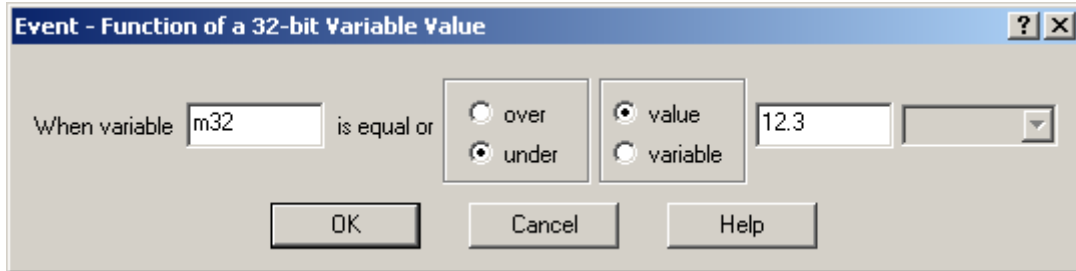
[Event Selection](#)

[Events](#)

[Motion Programming](#)

6.1.14.8. Event - Function of a variable value

This dialogue allows you to program an event function of the value of a selected variable. The events can be: when the selected variable is equal or over/under a value or the value of another variable. You may select any 32-bit MPL variable or parameter, long or fixed, for this event.



Introduce the **variable** name, the event condition: **over** (or equal) or **under** (or equal) and the comparison data: a **value** or the value of a **variable**.

Remark: If you choose a predefined MPL parameter or variable and as comparison a value, you'll see on the right list the measurement units associated with the selected variable.

OK: Close this dialogue and save the event set

Cancel: Close this dialogue without saving the event set.

Help: Open this help page.

See also:

[Event – Function of a variable value –MPL Programming Details](#)

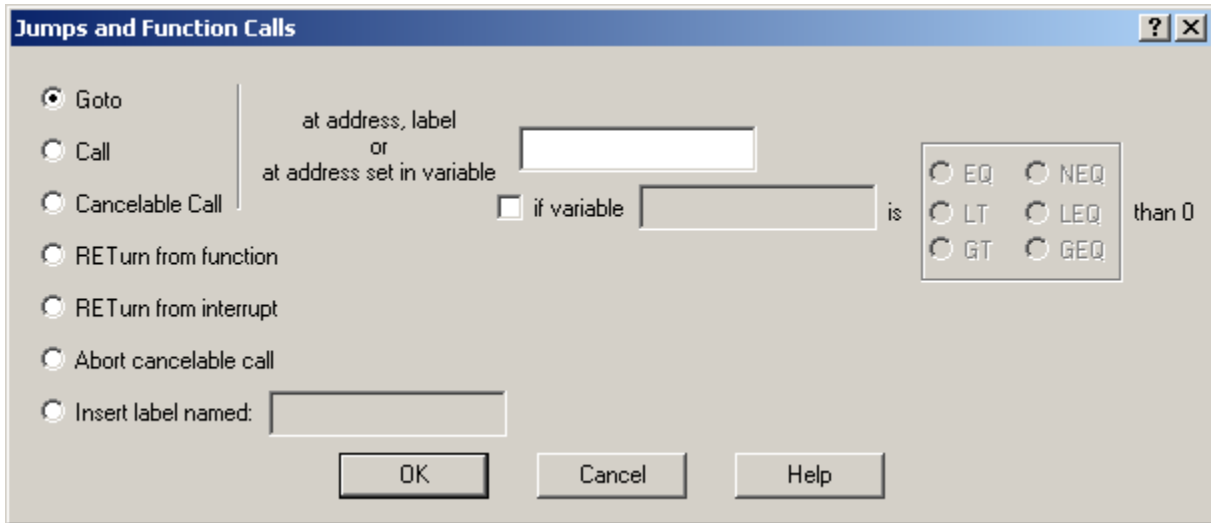
[Event Type Dialogue](#)

[Events](#)

[Motion Programming](#)

6.1.15. Jumps and Function Calls

The “Jumps and Function Calls” dialogue allows you to control the MPL program flow through unconditional or conditional jumps and unconditional, conditional or cancelable calls of MPL functions.



Select **Goto** and indicate the *jump address* in **address, label or address set in variable**. The *jump address* can be set directly as a numerical value (if it is known) or indirectly via:

- A **label**. Use **Insert label name** to place the label in the desired location. The label name can be any string of up to 32 characters, which starts with an alphanumeric character or with underscore.
- A 16-bit MPL **variable** whose value represents the jump address.

Remark: You may assign a label to a 16-bit integer variable. The variable takes the value of the label i.e. the address of the next instruction after label. Example: `user_var = jump_label;`

Leave **if variable** unchecked to execute an unconditional jump. Check **if variable** to execute a conditional jump and specify a *test variable* and a condition. The *test variable* is always compared with zero. The possible conditions are: < 0, <= 0, >0, >=0, =0, ≠ 0. If the condition is true the jump is executed, else the next MPL command is carried out.

Select **Call** and indicate the name of a MPL function in **address, label or address set in variable**. A MPL function starts with a **label** and ends with the **RET** instruction. The label gives the *MPL function address* and name. You can create, rename or delete MPL functions using the [Functions View](#).

Remark: The MPL functions are placed after the end of the main program

Similarly with the *jump address*, the *MPL function address* can be set directly, as a numerical value (if it is known), or indirectly via:

- The MPL function starting **label** (i.e. the function name)
- A 16-bit MPL **variable** whose value represents the *MPL function address*.

Leave **if variable** unchecked to execute an unconditional call. Check **if variable** to execute a conditional call and specify a *test variable* and a condition. The *test variable* is always compared with zero. The possible conditions are: < 0, <= 0, >0, >=0, =0, ≠ 0. If the condition is true the call is executed, else the next MPL command is carried out.

Choose **Cancelable Call** and indicate the *MPL function address* if the exit from the called function depends on conditions that may not be reached. In this case, using **Abort cancelable call** you can terminate the function execution and return to the next instruction after the call.

Select **RETurn from function** to insert the **RET** instruction, which ends a MPL function. When **RET** instruction is executed, the MPL program returns to the next instruction (motion sequence) after the MPL function call.

Select **RETurn from interrupt** to insert the **RETI** instruction, which ends a [MPL interrupt](#). When **RETI** instruction is executed, the MPL program returns to the point where it was before the MPL interrupt occurrence.

OK: Close this dialogue and save the motion sequence in your motion sequence list.

Cancel: Close this dialogue without saving the motion sequence in your motion sequence list.

Help: Open this help page.

See also:

[Jumps and Function Calls – MPL Programming Details](#)

[Functions View.](#)

[Motion Programming](#)

6.1.16. I/O General I/O (Firmware FxX)

The “I/O” dialogue allows you to program the following operations with the digital inputs and outputs:

- Read and save the status of a digital input into a variable
- Set low or high a digital output
- Read and save the status of multiple digital inputs into a variable
- Set multiple digital outputs according with the value of variable

The digital inputs and outputs are numbered: #0 to #39. Each programmable drive/motor has a specific number of inputs and outputs, therefore only a part of the 40 I/Os is used. The I/O numbering is common for all the products; hence each product has its own list of available I/Os. This is not an ordered list. For example, a product with 4 inputs and 4 outputs can use the inputs: #36, #37, #38 and #39 and the outputs #28, #29, #30 and #31.

I/O

Single I/O

Read an input

Read input line 2/LSP into variable user_var

Note: Input low -> variable is zero
Input high -> variable is not zero

Set an output

Set output line

status low high

Set I/O line type

Set as input

Read multiple inputs in variable

15	14	13	12	4	3	2	1	0
Enable	LSN	LSP	0	in#39	in#38	in#37	in#36	

Set multiple outputs to value of variable

15	14	13	4	3	2	1	0
Ready	Error		out#31	out#30	out#29	out#28	

OK Cancel Help

If you want to read the status of an input:

1. Select **Single I/O, Read input line**, choose the desired input from the list of available inputs and provide the name of an integer **variable** where to save the input status
2. Check **Set as input** if the input selected may also be used as an output (i.e. the input line number occurs in the outputs list too). Do this operation only once, first time when you use the input. Omit this check if the drive/motor has the inputs separated from the outputs (i.e. all have different line numbers)
3. Press **OK**

When this MPL command is executed, the variable where the input line status is saved, becomes:

- Zero if the input line was low
- Non-zero if the input line was high

Remark: Check the drive/motor user manual to find if the input line you are reading is directly connected or is inverted inside the drive/motor. If an input line is inverted, the variable where the input line is saved is inverted too: zero if the input is high (at connectors level), non-zero if the input is low (at connectors level).

If you want to set an output low or high:

1. Select **Single I/O**, choose **Set output line**, select the desired output from the list of available outputs and choose the output level: **low** or **high**
2. Check **Set as output** if the output selected may also be used as an input (i.e. the output line number occurs in the inputs list too). Do this operation only once, first time when you use the output. Omit this check if the drive/motor has the inputs separated from the outputs (i.e. all have different line numbers)
3. Press **OK**

Remark: The MPL code generated takes into account the possibility to have outputs inverted inside the drive/motor. This information, provided by the setup data, is used to inverse the output command logic: getting the output high (at connectors level) means setting the output low and to getting the output low (at connectors level) means setting the output high

Check **Read inputs in variable** to read simultaneously more inputs and specify the name of an integer variable where to save their status. The inputs are:

- Enable input – saved in bit 15
- Limit switch input for negative direction (LSN) - saved in bit 14
- Limit switch input for positive direction (LSP) - saved in bit 13
- General-purpose inputs #39, #38, #37 and #36 – save din bits 3, 2, 1 and 0

The bits corresponding to these inputs are set as follows: 0 if the input is low and 1 if the input is high. The other bits of the variable are set to 0.

Remark: If one of these inputs is inverted inside the drive/motor, the corresponding bit from the variable is inverted too. Hence, these bits always show the inputs status at connectors level (0 if input is low and 1 if input is high) even when the inputs are inverted.

Check **Set multiple outputs to a value of variable** to set simultaneously more outputs with the value of the specified variable. The outputs are:

- Ready output – set by bit 15
- Error output – set by bit 14
- General-purpose outputs: #31, #30, #29, #28 – set by bits 3, 2, 1, and 0

The outputs are set as follows: low if the corresponding bit in the variable is 0 and high if the corresponding bit in the variable is 1. The other bits of the variable are not used.

Remark: If one of these outputs is inverted inside the drive/motor, its command is inverted too. Hence, the outputs are always set at connectors level according with the bits values (low if bit is 0 and high if bit is 1) even when the outputs are inverted.

CAUTION: Do not use **Set multiple outputs to a value of variable** if any of the 6 outputs mentioned is not on the list of available outputs of your drive/motor. There are products that use some of these outputs internally for other purposes. Attempting to change these lines status may harm your product.

OK: Close this dialogue and save the motion sequence in your motion sequence list.

Cancel: Close this dialogue without saving anything in your motion sequence list.

Help: Open this help page.

See also:

[General-purpose I/O – MPL Programming Details](#)

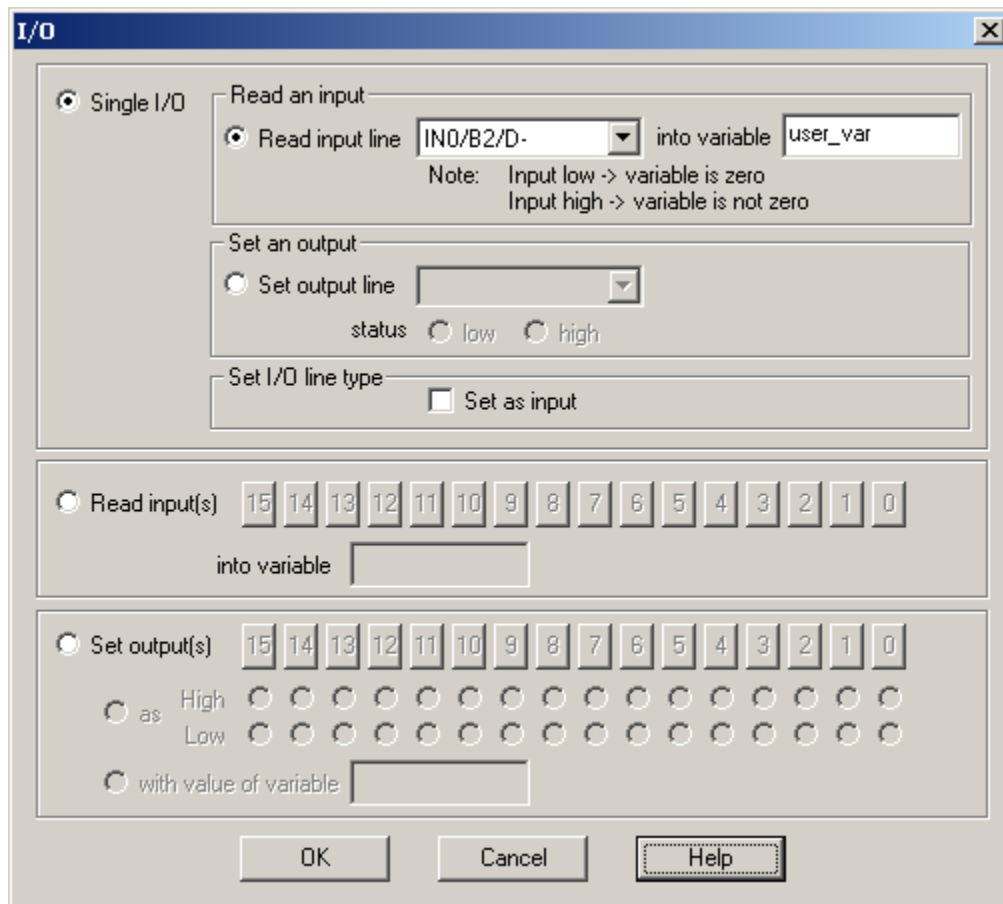
[Motion Programming](#)

6.1.17. I/O General I/O (Firmware FBxx)

The “I/O” dialogue allows you to program the following operations with the digital inputs and outputs:

- Read and save the status of a digital input into a variable
- Set low or high a digital output
- Read and save the status of multiple digital inputs into a variable
- Set multiple digital outputs according with an immediate value or the value of 16-bit variable

The digital inputs and outputs are numbered: 0 to 15. Each programmable drive/motor has a specific number of inputs and outputs, therefore only a part of the 16 inputs or outputs is used. The I/O numbering is common for all the products; hence each product has its own list of available I/Os. This is an ordered list. For example, a product with 4 inputs and 4 outputs can use the inputs: IN0, IN1, IN2 and IN3 and the outputs OUT0, OUT1, OUT2 and OUT3.



If you want to read the status of an input:

1. Select **Single I/O, Read input line**, choose the desired input from the list of available inputs and provide the name of an integer **variable** where to save the input status
2. Check **Set as input** if the input selected may also be used as an output. Do this operation only once, first time when you use the input. Omit this check if the drive/motor has the inputs separated from the outputs (i.e. all have different line numbers)
3. Press **OK**

When this MPL command is executed, the variable where the input line status is saved, becomes:

- Zero if the input line was low
- Non-zero if the input line was high

Remark: Check the drive/motor user manual to find if the input line you are reading is directly connected or is inverted inside the drive/motor. If an input line is inverted, the variable where the input line is saved is inverted too: zero if the input is high (at connectors' level), non-zero if the input is low (at connectors' level).

If you want to set an output low or high:

1. Select **Single I/O**, choose **Set output line**, select the desired output from the list of available outputs and choose the output level: **low** or **high**
2. Check **Set as output** if the output selected may also be used as an input. Do this operation only once, first time when you use the output. Omit this check if the drive/motor has the inputs separated from the outputs.
3. Press **OK**

Remark: The MPL code generated takes into account the possibility to have outputs inverted inside the drive/motor. This information, provided by the setup data, is used to inverse the output command logic: getting the output high (at connectors' level) means setting the output low and to getting the output low (at connectors' level) means setting the output high

Check **Read inputs in variable** to read simultaneously more inputs and specify the name of an integer variable where to save their status. The bits corresponding to these inputs are set as follows: 0 if the input is low and 1 if the input is high. The other bits of the variable are set to 0.

Remark: If one of these inputs is inverted inside the drive/motor, the corresponding bit from the variable is inverted too. Hence, these bits always show the inputs status at connectors level (0 if input is low and 1 if input is high) even when the inputs are inverted.

Check **Set outputs** to set simultaneously more outputs with the value of 16-bit mask or variable. Select the outputs you want to command and specify how they are set:

- with the mask generated after setting **as High** or **Low** each of the selected outputs
- **with the value** of the specified 16-bit variable.

The outputs are set as follows: low if the corresponding bit in the mask or variable is 0 and high if the corresponding bit in the mask or variable is 1. The other bits of the mask or variable are not used.

Remark: If one of these outputs is inverted inside the drive/motor, its command is inverted too. Hence, the outputs are always set at connectors level according with the bits values (low if bit is 0 and high if bit is 1) even when the outputs are inverted.

OK: Close this dialogue and save the motion sequence in your motion sequence list.

Cancel: Close this dialogue without saving anything in your motion sequence list.

Help: Open this help page.

See also:

[General-purpose I/O – MPL Programming Details](#)

[Motion Programming](#)

6.1.18. Assignment & Data Transfer - Setup 16-bit variable

The “Assignment and Data Transfer – 16 bit Integer Data” dialogue helps you to:

1. Assign a value to a [16-bit integer MPL parameter/variable](#)
2. Transfer in a memory location, a 16-bit value or the value of a 16-bit integer MPL parameter or variable

The dialog box "Assignment & Data Transfer - 16 bit Integer Data" contains the following elements:

- Set 16-bit variable:
- With value / 16 bit variable / label:
- With:
 - data
 - program
 - E2ROMmemory contents, located at address set in pointer variable: then increment the pointer variable
- With:
 - low
 - highpart of 32-bit variable:
- With the inverse (-) of variable:
- Using AND mask: h and OR mask: h
- With checksum of data located in:
 - data
 - program
 - E2ROMmemory between address: h and: h
- Set:
 - data
 - program
 - E2ROMmemory contents, located at address set in pointer variable: with value/variable: then increment the pointer variable

Buttons: OK, Cancel, Help

Select **Set 16-bit variable** to assign a value to a 16-bit integer MPL parameter or variable. Introduce its name and choose one of the possible sources:

- **With value / 16 bit variable / label:** A 16-bit *value* or the value of a *16-bit variable* or the value of a *label*. Introduce in the associated field the value or the variable/label name.
- **With data / program / E2ROM memory contents located at address set in pointer variable:** The value of a memory location whose address is set in another 16-bit (pointer) variable. Introduce in the associated field the pointer variable name. Check **then increment the pointer variable** to automatically increment by one the pointer value, after the assignment is done. This option is particularly useful for repetitive assign operations where source is placed in successive memory locations. The memory type is split into 3 categories: **data** – for the RAM area for MPL data, **program** – for the RAM area for MPL programs and **E2ROM** – for the EEPROM area for MPL programs.

Remark: The **data** memory may be used to extend the number of user-defined variables. By data exchanges with MPL variables, the data memory locations may be used as a temporary

buffer. Work for example for these operations with the RAM locations reserved but not used by the cam tables.

- **With low / high part of a 32-bit variable:** The low or high 16-bit part of a 32-bit MPL parameter or variable. Introduce in the associated field the variable name.
- **With inverse (-) of variable:** The inverse (negate) value of a 16-bit MPL parameter or variable. Introduce in the associated field the variable name
- **Using AND mask...and OR mask ...:** The result of a logical operations:
 - AND between the selected variable and the AND mask value
 - OR between the above result and the OR mask value
- **With checksum of data located in data / program / E2ROM memory between address ... and ...:** The result of a checksum performed with all the locations situated between the 2 specified memory addresses. The memory type is split into 3 categories like in the case of indirect addressing: **data** – for the RAM area for MPL data, **program** – for the RAM area for MPL programs and **E2ROM** – for the EEPROM area for MPL programs.

***Remark:** The checksum is the sum modulo 65536 of all the memory values, including those from the limits. The address limits are hexadecimal values.*

Select **Set data / program / E2ROM memory contents located at address set in the pointer variable** to transfer in a memory location, a 16-bit value or the value of a 16-bit integer MPL parameter or variable. The memory location address is provided by another 16-bit (pointer) variable. Introduce in the associated fields the pointer variable name and the 16-bit value or the variable name. Check **then increment the pointer variable** to automatically increment by one the pointer value, after the assignment is done. This option is particularly useful for repetitive assign operations where destination is placed in successive memory locations. The memory type is split into 3 categories: **data** – for the RAM area for MPL data, **program** – for the RAM area for MPL programs and **E2ROM** – for the EEPROM area for MPL programs.

OK: Close this dialogue and save the assignment or data transfer in your motion sequence list.

Cancel: Close this dialogue without anything in your motion sequence list.

Help: Open this help page.

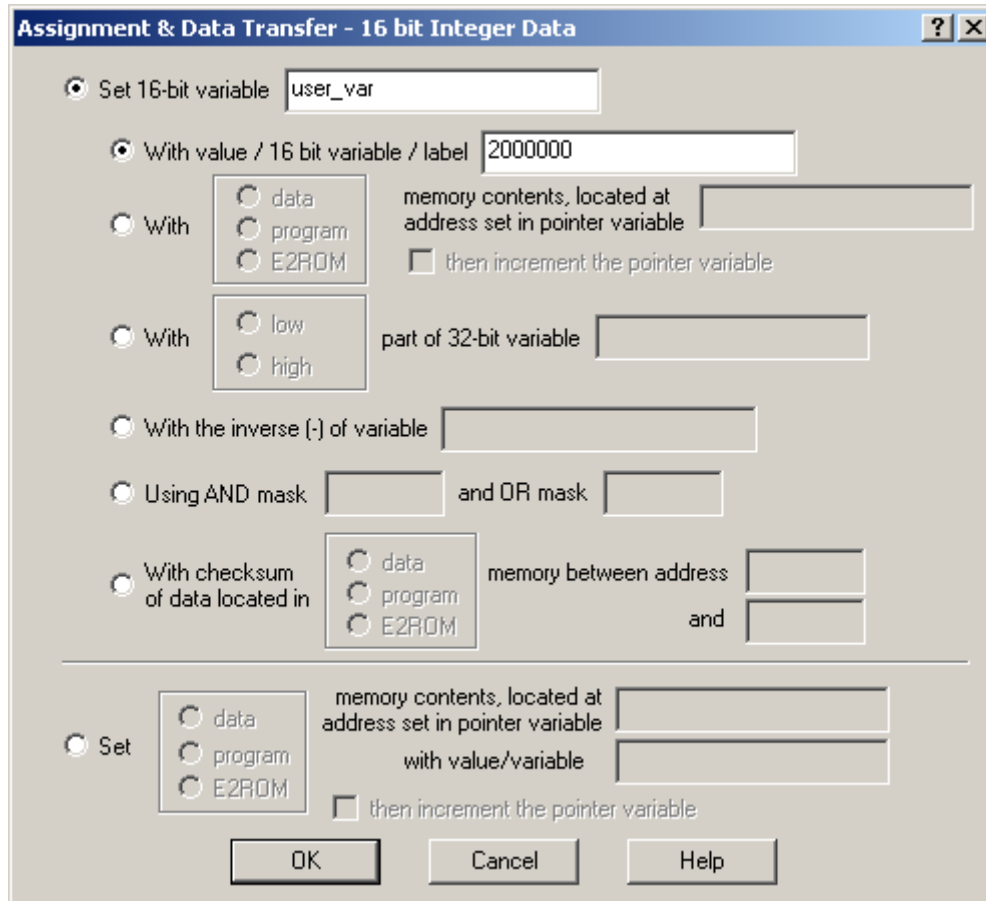
See also:

[Assignment and Data Transfer: 16-bit data – MPL Programming Details](#)
[Motion Programming](#)

6.1.19. Assignment & Data Transfer - Setup 32-bit variable

The “Assignment and Data Transfer – 32-bit Long or Fixed Data” dialogue helps you:

1. Assign a value to a [32-bit long or fixed MPL parameter/variable](#)
2. Assign a value to the high (16MSB) or low (16LSB) part of a 32-bit long or fixed data
3. Transfer in 2 consecutive memory locations, a 32-bit value or the value of a 32-bit long or fixed MPL parameter or variable



Select **Set 32-bit variable** to assign a value to a 32-bit long or fixed MPL parameter or variable. Introduce its name and choose one of the possible sources:

- **With value / 32 bit variable:** A 32-bit *value* or the value of a *32-bit variable*. Introduce in the associated field the value or the variable name.
- **With data / program / E2ROM memory contents located at address set in pointer variable:** The value of 2 consecutive memory locations. The first memory address (the lowest) is provided by another 16-bit (pointer) variable. Introduce in the associated field the pointer variable name. Check **then increment the pointer variable** to automatically increment by two the pointer value, after the assignment is done. This option is particularly useful for repetitive assign operations where source is placed in successive memory locations. The memory type is split into 3 categories: **data** – for the RAM area for MPL data, **program** – for the RAM area for MPL programs and **E2ROM** – for the EEPROM area for MPL programs.

Remark: The **data** memory may be used to extend the number of user-defined variables. By data exchanges with MPL variables, the data memory locations may be used as a temporary buffer. Work for example for these operations with the RAM locations reserved but not used by the cam tables.

- **With inverse (-) of variable:** The inverse (negate) value of a 32-bit MPL parameter or variable. Introduce in the associated field the variable name
- **With 16-bit value of variable...left shifted with:** The value of a 16-bit MPL data, left shifted with 0 to 16 bits. Introduce in the associated fields the variable name and the shift value.

Select **Set low / high part of 32-bit variable... with value/16-bit variable...** to copy a 16-bit data into the higher or lower 16-bits or a 32-bit MPL data. The 16-bit data can be either an immediate value or a 16-bit MPL data. Choose **low** or **high** part and introduce in the associated field the value or the variable name.

Remarks:

- The left shift operation is done with sign extension. If you intend to copy the value of an integer MPL data into a long MPL data preserving the sign use this operation with left shift 0
- If you intend to copy the value of a 16-bit unsigned data into a 32-bit long variable, assign the 16-bit data in low part of the long variable and set the high part with zero.

Select **Set data / program / E2ROM memory contents located at address set in the pointer variable** to transfer in 2 consecutive memory locations, a 32-bit value or the value of a 32-bit integer MPL parameter or variable. The first memory address (the lowest) is provided by another 16-bit (pointer) variable. Introduce in the associated field the pointer variable name and the 16-bit value or the variable name. Check **then increment the pointer variable** to automatically increment by two the pointer value, after the assignment is done. This option is particularly useful for repetitive assign operations where destination is placed in successive memory locations. The memory type is split into 3 categories: **data** – for the RAM area for MPL data, **program** – for the RAM area for MPL programs and **E2ROM** – for the EEPROM area for MPL programs.

***Remark:** When this operation is performed having as source an immediate value, the MPL compiler checks the type and the dimension of the immediate value and based on this generates the binary code for a 16-bit or a 32-bit data transfer. Therefore if the immediate value has a decimal point, it is automatically considered as a fixed value. If the immediate value is outside the 16-bit integer range (-32768 to +32767), it is automatically considered as a long value. However, if the immediate value is inside the integer range, in order to execute a 32-bit data transfer it is necessary to add the suffix **L** after the value, for example: **200L** or **-1L**.*

OK: Close this dialogue and save the assignment or data transfer in your motion sequence list.

Cancel: Close this dialogue without anything in your motion sequence list.

Help: Open this help page.

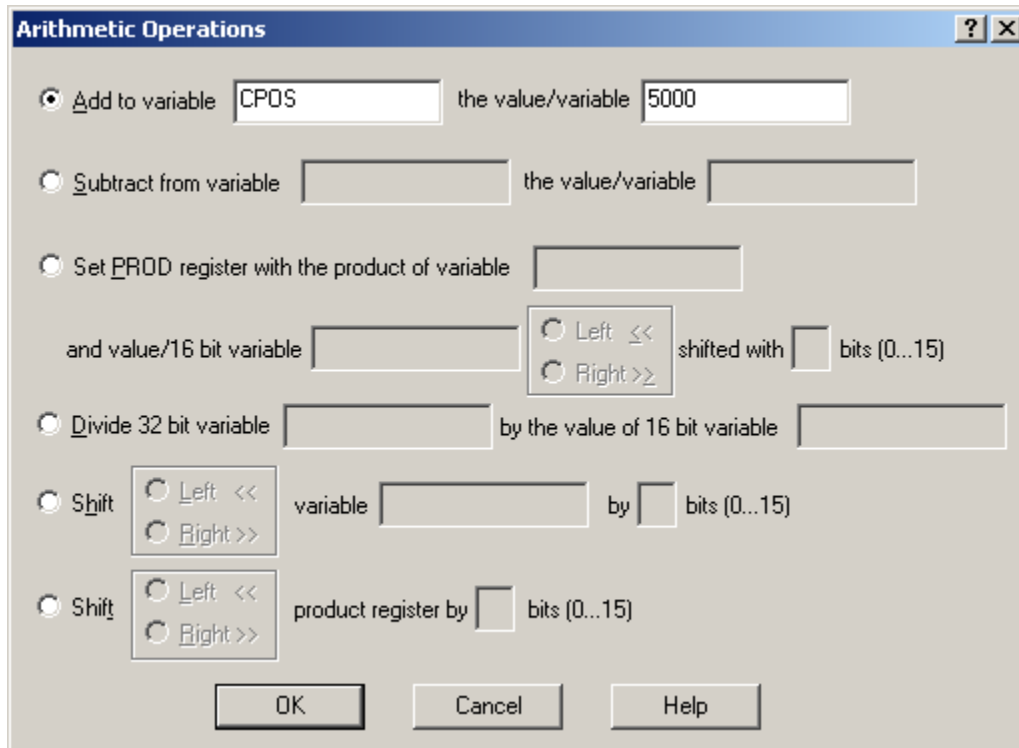
See also:

[Assignment and Data Transfer: 32-bit data – MPL Programming Details](#)

[Motion Programming](#)

6.1.20. Assignment & Data Transfer - Arithmetic Operations

The “Arithmetic Operations” dialogue allows you to program the following arithmetic operations: addition, subtraction, multiplication and division, plus the left and right shifting. All these operations are signed i.e. the operands are treated as signed numbers. Except the multiplication, the result is saved in the left operand. For multiplication, the result is saved in the dedicated product register.



Select **Add to variable** and introduce the name of the left operand to perform an addition. Indicate the right operand in **the value/variable** field. The left operand may be a 16-bit or 32-bit MPL data. The right operand may be an immediate value or another MPL data, of the same type as the left operand.

Remark: When the left operand is a 32-bit long or fixed MPL data and the right operand is a 16-bit integer value, it is treated as follows:

- Sign extended to a 32-bit long value, if the left operand is a 32-bit long
- Set as the integer part of a fixed value, if the left operand is a 32-bit fixed

Select **Subtract from variable** and introduce the name of the left operand to perform a subtraction. Indicate the right operand in **the value/variable** field. The left operand may be a 16-bit or 32-bit MPL data. The right operand may be an immediate value or another MPL data, of the same type as the left operand.

Remark: When the left operand is a 32-bit long or fixed MPL data and the right operand is a 16-bit integer value, it is treated as follows:

- Sign extended to a 32-bit long value, if the left operand is a 32-bit long
- Set as the integer part of a fixed value, if the left operand is a 32-bit fixed

Select **Set PROD register with the product of variable** and introduce the name of the first operand to perform a multiplication. Indicate the second operand in **with value / 16 bit variable** field. The first

operand may be a 16-bit or 32-bit MPL data. The second operand may be a 16-bit value or a 16-bit MPL data. The multiplication result is saved left or right shifted in a dedicated 48-bit product register. Choose the shift type **Left** or **Right** and number of shift bits: 0 to 15. Use 0 to perform no shift.

Remark: *The result is placed in the product register function of the left operand. When shift is 0:*

- *In the 32 least significant bits, when the left operand is a 16-bit integer. The result is a 32-bit long integer*
- *In all the 48 bits, when the left operand is a 32-bit fixed. The result has the integer part in the 32 most significant bits and the fractional part in the 16 least significant bits*
- *In all the 48 bits, when the left operand is a 32-bit long. The result is a 48-bit integer*

The MPL variable **PRODH** contains the 32 most significant bits of the product register. The MPL variable **PROD** contains the 32 least significant bits of the product register.

Select **Divide variable** and introduce the name of the left operand: the dividend, to perform a division. Indicate the right operand: the divisor, in the **by the value of variable** field. The dividend is a 32-bit MPL data. The divisor is 16-bit MPL data.

Remark: *The result, saved in first operand, is a fixed value with the integer part in the 16 most significant bits and the fractional part in the 16 least significant bits.*

Choose **Shift Left / Right** and introduce the name of the MPL data to be shifted left or right in the **variable** field, followed by the number of shift bits: 0 to 15. The MPL data can be any 16-bit or 32-bit MPL data.

Choose **Shift Left / Right product register by** and introduce the number of shift bits: 0 to 15, to perform a left or right shift of the 48-bit product register.

Remark: *At right shifts, high order bits are sign-extended and the low order bits are lost. At left shifts, high order bits are lost and the low order bits are zeroed.*

OK: Close this dialogue and save the arithmetic or logic operation in your motion sequence list.

Cancel: Close this dialogue without anything in your motion sequence list.

Help: Open this help page.

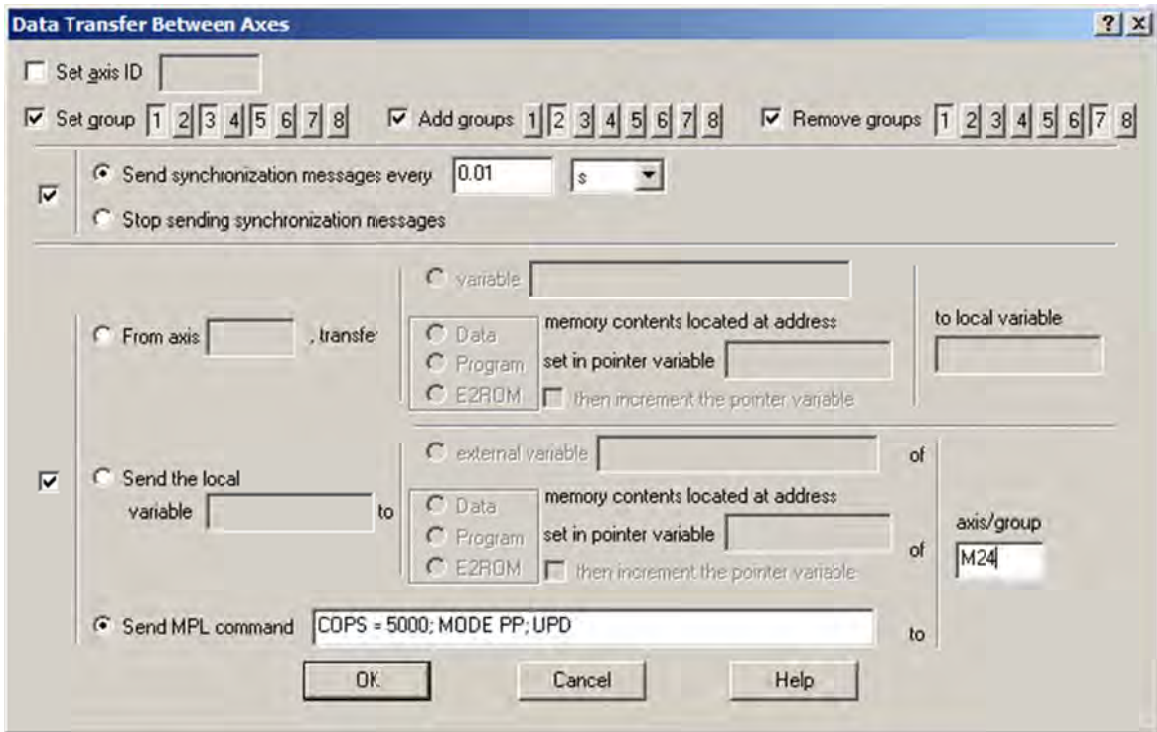
See also:

[Arithmetic and logic operations. MPL Programming Details](#)

[Motion Programming](#)

6.1.21. Assignment & Data Transfer - Data Transfer Between Axes

The “Data Transfer Between Axes” dialog allows you to program data transfer operations between drives/motors connected in a network. From this dialog, you can also change the **axis ID** – the drive/motor network address, and the **groups** it belongs for multicast transmissions as well as to activate/deactivate the synchronization between axes.



Check **Set axis ID** if you want to change the axis ID and set a new value. The axis ID is a value between 1 and 255. It is initially set at power on using the following algorithm:

- With the value read from the EEPROM setup table containing all the setup data. If this value is 0, the axis ID is set with the value read from the hardware switches/jumpers or in their absence according with d)
- If the setup table is invalid, with the last axis ID value read from a valid setup table
- If there is no axis ID set by a valid setup table, with the value read from the hardware switches/jumpers for axis ID setting
- If the drive/motor has no hardware switches/jumpers for axis ID setting, with the default axis ID value which is 255.

Remark: Typically, the axis ID is kept constant during operation at the value established during the setup phase. However, if needed, you can change the axis ID to any of the 255 possible values, using the above command

In MotionPRO Developer, each application has associated an Axis Number, set in **Application General Information**. When an application is selected, all the data exchange operations are performed with the drive/motor having the same axis ID as the application Axis Number. An axis ID change may create communication problems, if this is performed during operation i.e. if the drive/motor starts with one axis ID and later on switches to another axis ID.

Check **Set group** if you want set the groups to which a drive/motor belongs. A group is way to identify a number of drives, for a multicast transmission. Each drive can be programmed to be member of one or several of the 8 possible groups (up to all). A drive will accept all the messages sent to any of the groups it belongs. Push the buttons for the groups the drive/motor will belong. Use **Add groups** or **Remove groups** to add or remove your drive/motor from one or several groups.

Remark: A message can be:

- Sent to an axis defined by an **Axis ID**
- Multicast to one group of axes defined by a **Group ID**. The **Group ID** is an 8-bit value, where each bit set represents a group. For example, a multicast to **Group ID = 4** (100b) will be received by all drives from group 3.
- Broadcast to all nodes, if the **Group ID = 0**.

Check **Synchronization** group to activate/deactivate the synchronization procedure. This procedure requires activating one axis as a synchronization master. The other axes are deactivated and are synchronization slaves. Select **Send synchronization messages every...** and set the time interval between synchronization messages, to activate the synchronization master. Recommended starting value for the time interval is 20ms. When synchronization procedure is active, the execution of the control loops on the slaves is synchronized with those of the master within a 10 μ s time interval. Due to this powerful feature, drifts between master and slave axes are eliminated. Deactivate the synchronization procedure by choosing **Stop sending synchronization messages**. This will disable the synchronization master and set the axis as a synchronization slave. In the absence of a master, the synchronization process is stopped.

The data transfer operations may be split into three categories:

1. Read data from a remote axis. A variable or a memory location from the remote axis is saved into a local variable
2. Write data to a remote axis or group of axes. A variable or a memory location of a remote axis or group of axes is written with the value of a local variable
3. Send MPL commands from local drive to a remote drive or group of drives

Check data transfer commands, and select **From axis** to read from the remote axis specified, the value of a **variable** or the **data / program / E2ROM memory contents located at an address set in a pointer variable**. The data is saved in the local MPL variable indicated in **to local variable** field. The local variable can be either a 16-bit or a 32-bit MPL data. Its type, dictates the data transfer size. Check **then increment the pointer variable** to automatically increment the pointer by one or two function of the local variable type, after the transfer is performed. The memory type is split into 3 categories: **data** – for the RAM area for MPL data, **program** – for the RAM area for MPL programs and **E2ROM** – for the EEPROM area for MPL programs.

Select **Send the local variable** to copy on a remote axis or group of axes, the value of the local variable specified. The data is saved either in an **external/remote variable** or in the **data / program / E2ROM memory** location(s) from **address set in the pointer variable** indicated. The local variable can be either a 16-bit or a 32-bit MPL data. Its type, dictates the data transfer size. Check **then increment the pointer variable** to automatically increment the pointer by one or two function of the local variable type, after the transfer is performed. The memory type is split into 3 categories: **data** – for the RAM area for MPL data, **program** – for the RAM area for MPL programs and **E2ROM** – for the EEPROM area for MPL programs. The destination specified at **axis/group** can be:

- An **axis ID** set with a number between 1 and 255
- A **group** set with letter **G** followed by a number between 1 and 8. Examples: G1, G7
- A **broadcast** to all axes set with letter **B**

Select **Send MPL command** to program the local axis to transmit the MPL command(s) you type in the associated field towards the destination specified in the **axis/group** field. The transmission is done when the command is executed.

Remarks:

- This command offers a very powerful tool through which one drive/motor may control other drives/motors from the network. For example it can start or stop the other drives motion or check their status
- You may type multiple MPL commands separated by semicolon (;). These will be sent one by one in the order of occurrence in the edit.
- Via this type of messages, you can send all the MPL instructions having an instruction code of maximum 4 words. In this category enter most of the MPL commands (see [MPL Instruction Coding](#) and the detailed description of the [MPL Instructions](#)).

OK: Close this dialogue and save the operations selected in your motion sequence list.

Cancel: Close this dialogue without saving anything in your motion sequence list.

Help: Open this help page.

See also:

[Axis Identification](#)

[Data Transfer Between Axes – MPL Programming Details](#)

[Remote Control](#)

[Motion Programming](#)

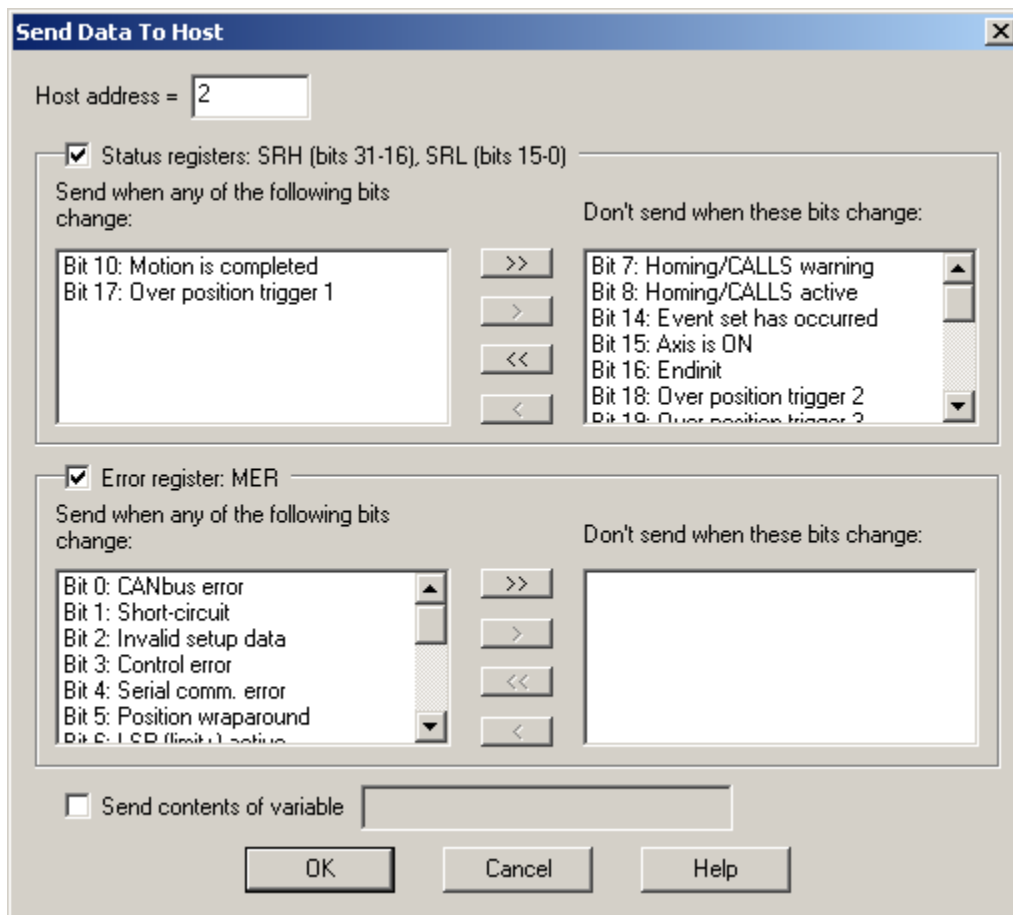
6.1.22. Send data to host

The “Send Data to Host” dialogue allows you to program when the drive/motor will send messages to your host. The message transmission can be triggered by:

- Conditions which change the status or error registers
- The execution of a dedicated MPL command from your MPL program. Through this command you can send to your host the contents of any MPL data

In the first case, you can select the status or the error register bits, which will trigger a message when are changed. The selection is done via masks, one for each register, where for each bit you can choose if to trigger or not a transmission when it is changed.

When the transmission is triggered by a bit change in a status register **SRH** (high part) or **SRL** (low part), the message sent contains these 2 registers grouped together as a single 32-bit register/data. When the transmission is triggered by a bit change in the error register **MER**, the message sent contains this register.



In the **Host address** write the axis ID of the host.

Check **Status Register** to enable transmission on status register bit changes. From the right list, select a bit whose change you want to trigger a message transmission and press the [<] button. The selected bit will appear on the left list. Repeat the operation for the other bits. Use the button [<<] to select all the bits. Choose a bit from the left list and press the [>] button to move it back to the right list. Use the [>>] button to remove all the bits from the left list.

Check **Error Register** to enable transmission on error register bit changes. From the right list, select a bit whose change you want to trigger a message transmission and press the [<] button. The selected bit will appear on the left list. Repeat the operation for the other bits. Use the button [<<] to select all the bits. Choose a bit from the left list and press the [>] button to move it back to the right list. Use the [>>] button to remove all the bits from the left list.

Remark: After power on, the 2 masks are empty i.e. none of the status or error bits is selected to trigger a transmission on change.

Check **Send contents of variable** and indicate the name of the MPL data to be sent when this MPL command is executed. The MPL data can be any 16-bit or 32-bit data: MPL registers, parameters, variable or user variables.

Remark: By default, at power on, the host address is set equal with the drive/motor axis ID. Therefore, the messages will be sent via RS-232 serial communication. If the host address is different from the drive/motor axis ID, the messages are sent via the other communication channels: CAN bus, RS485, etc.

See also:

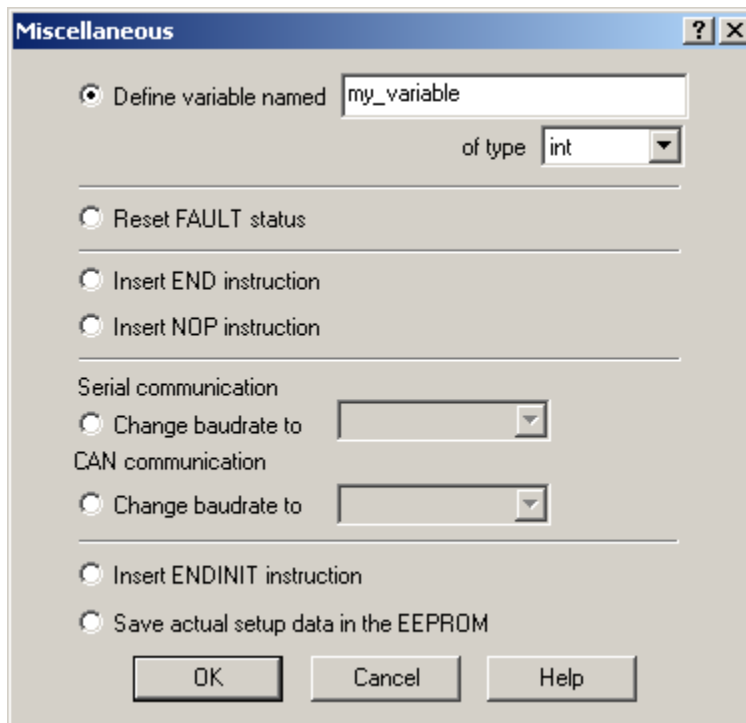
[Messages sent to the host – MPL Programming Details](#)

[Motion Programming](#)

6.1.23. Assignment & Data Transfer - Miscellaneous

The “Miscellaneous” dialogue allows you to:

- Declare user variables
- Reset/exit the drive/motor from the [FAULT status](#)
- Execute less frequently used MPL commands like: END, NOP, ENDINIT
- Change the CAN bus and serial RS232 / RS485 communication settings
- Save actual setup data from RAM into the EEPROM in the setup table



Select **Define variable named** if you want to define a new variable. Specify the variable name in the next field and choose the variable type from the list. The options are: **int**, **fixed** or **long**. A variable of type **int** is a 16-bit signed integer. A variable of type **long** is a 32-bit signed integer. A variable of type **fixed** is 32-bit wide and is used for signed fixed-point representations with 16MSB the integer part and 16LSB the fractionary part.

Select **Reset FAULT status** to exit a drive/motor from the [FAULT status](#) in which it has entered due to an error. After a fault reset command, most of the bits from error register **MER** are cleared (set to 0), ready output is set to ready level, error output is set to no error level and drive/motor returns to normal operation.

Remarks:

- *The FAULT reset command does not change the status of MER.15 (enable input on disabled level), MER.7 (negative limit switch input active), MER.6 (positive limit switch input active) and MER.2 (invalid setup table)*
- *The drive/motor will return to FAULT status if there are errors when the FAULTR command is executed*

Select **Insert END instruction** to introduce in your MPL program the instruction END. When END is executed, the MPL program execution is stopped.

Remark: *It is mandatory to end the main section of a MPL program with an END command. All the MPL functions and the MPL interrupt service routines must follow after the END command. MotionPRO Developer automatically handles these requirements, when it generates the MPL program to be compiled and downloaded into the drive.*

Select **Insert NOP instruction** to introduce a NOP (No operation) instruction. It can be used as a delay between two motion sequences / instructions.

In the **Serial communication** section, choose **Change baudrate to** if you want to change the drive baud rate for RS-232 and RS-485 communication. Choose from the drop list one of the available baud rates: 9600, 19200, 38400, 56000 and 115200.

Remarks:

1. *The drives/motors default serial baud rate after power on is 9600 baud, unless another value was saved in the setup table. When you start MotionPRO Developer, the drives/motors serial baud rate is automatically adjusted to the last value selected at **Communication | Setup** in the **Baud Rate** field.*
2. *Use this command when a drive/motor operates in AUTORUN (after power on starts to execute the MPL program from the EEPROM) and it must communicate with a host at a baud rate different from the default value. In this case, the MPL program must start with a serial baud rate change.*
3. *An alternate solution to the above case is to set via MotionPRO Developer the desired baud rate and then to save it the EEPROM, with command **SAVE**. After a reset, the drive/motor starts directly with the new baud rate, if the setup table was valid. Once set, the new default baud rate is preserved, even if the setup table is later on disabled, because the default serial baud rate is stored in a separate area of the EEPROM.*

In the **CAN communication** section, choose **Select Set CAN baudrate to** if you want to change the baud rate for CAN-bus communication. Choose from the drop list one of the available CAN baud rates: 125kb, 250kb, 500kb, 800kb, 1Mb.

Remarks:

1. *The drives/motors default CAN baud rate after power on is 500kb, unless another value was saved in the setup table. In MotionPRO Developer, at **Communication | Setup**, in the **Baud Rate** field, you must choose the same value as the default CAN baud rate of the drives/motors value. This selection refers ONLY to the CAN bus interface of your PC*
2. *Use this command when a drive/motor operates in AUTORUN (after power on starts to execute the MPL program from the EEPROM) and it must communicate with a host at a baud rate different from the default value. In this case, the MPL program must start with a CAN baud rate change.*
3. *An alternate solution to the above case is to set via MPL command **CANBR** the desired CAN baud rate and then to save it the EEPROM, with command **SAVE**. After a reset, the drive/motor starts directly with the new CAN baud rate, if the setup table was valid. Once set, the new default CAN baud rate is preserved, even if the setup table is later on disabled, because the default CAN baud rate is stored in a separate area of the EEPROM.*

Select **Insert ENDINIT instruction** to introduce an **ENDINT** (end of initialization) instruction. This command uses the available setup data to perform key initializations, but does not activate the controllers or the PWM outputs. These are activated with the **AXISON** command

Remarks:

-
1. After power on, the **ENDINIT** command may be executed only once. Subsequent **ENDINIT** commands are ignored.
 2. The **AXISON** command must be executed after the **ENDINIT** command
 3. Typically, the **ENDINIT** command is executed at the beginning of a MPL program and may be followed by the **AXISON** command even if no motion mode was set. In the absence of any programmed motion, the drive applies zero voltage to the motor.
 4. In MotionPRO Developer, **ENDINIT** and **AXISON** commands are automatically included when a MPL program is generated. Hence you can start directly with the motion programming

Select **Save actual setup data in the EEPROM** to insert a SAVE instruction in the MPL program. When SAVE instruction is executed, the actual values of the MPL parameters are copied from the RAM memory into the EEPROM memory, in the setup table. Through this command, you can save all the setup modifications done, after power on initialization.

OK: Close this dialogue and save the MPL commands in your motion sequence list.

Cancel: Close this dialogue without saving anything in your motion sequence list.

Help: Open this help page

See also:

[Miscellaneous commands – MPL Programming Details](#)

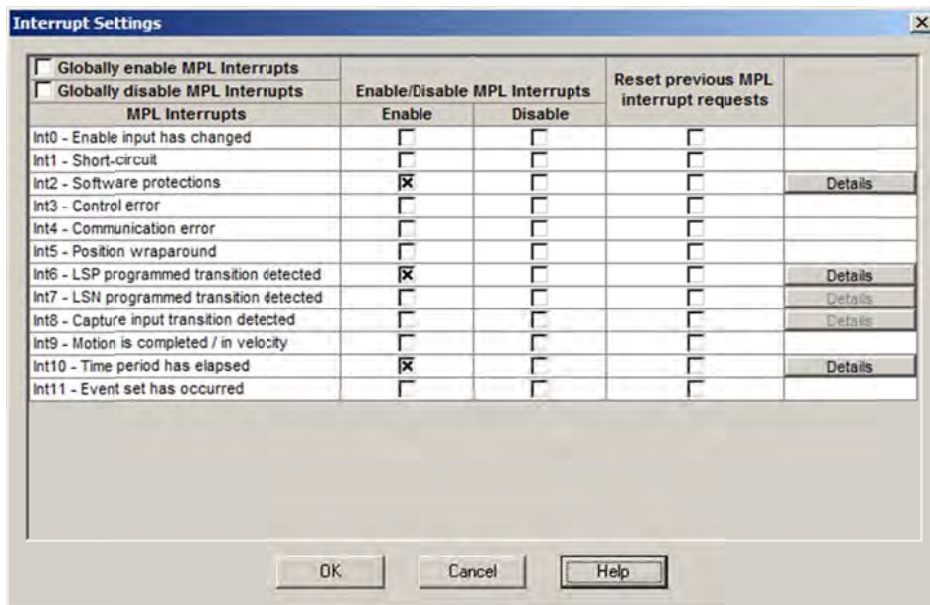
[Motion Programming](#)

6.1.24. MPL Interrupt Settings

The “Interrupt Settings” dialogue allows you to activate and/or deactivate the MPL (ElectroCraft Motion Program Language) interrupts. When a MPL interrupt occurs, the normal MPL program execution is suspended to execute a MPL function associated with the interrupt, called **Interrupt Service Routine** (in short **ISR**). The MPL interrupt mechanism is the following:

- The drive continuously monitors 12 conditions that may generate MPL interrupts. In case of motion controller applications there is a 13th condition related to slave error status.
- The motion controller has an additional condition that triggered interrupt when an error on the slaves occurs
- When an interrupt condition occurs, a flag (bit) is set in the ISR (Interrupt Status Register)
- If the interrupt is unmasked e.g. the same bit (as position) is set in the ICR (Interrupt Control Register) and also if the interrupts are globally enabled (EINT instruction was executed), the interrupt condition is qualified and it generates a MPL interrupt
- The interrupt causes a jump to the associated interrupt service routine. On entry in this routine, the MPL interrupts are globally disabled (DINT) and the interrupt flag is reset
- The interrupt service routine must end with the MPL instruction RETI, which returns to normal program execution and in the same time globally enables the MPL interrupts.

Interrupt settings dialog for drive/motor



The 12 conditions are:

1. **Int0 – Enable input has changed:** either transition sets the interrupt flag
2. **Int1 – Short-circuit:** when the drive/motor hardware protection for short-circuit is triggered
3. **Int2 – Software protections:** when any of the following protections is triggered:
 - a) Over current
 - b) I2t motor
 - c) I2t drive
 - d) Over temperature motor
 - e) Over temperature drive
 - f) Over voltage
 - g) Under voltage
4. **Int3 – Control error:** when the control error protection is triggered
5. **Int4 – Communication error:** when a communication error occurs
6. **Int5 – Wrap around:** when the target position computed by the reference generator wraps around because it bypasses the limit of the 32-bit long integer representation
7. **Int6 – LSP programmed transition detected:** when the programmed transition is detected on the limit switch input for positive direction (LSP)
8. **Int7 – LSN programmed transition detected:** when the programmed transition is detected on the limit switch input for negative direction (LSN)
9. **Int8 – Capture input transition detected:** when the programmed transition is detected on the 1st capture/encoder index input or on the 2nd capture/encoder index input
10. **Int9 – Motion is completed:** in position control, when *motion complete* condition is set and in speed control when target speed reaches zero.
11. **Int10 – Time period has elapsed:** periodic time interrupt with a programmable time period
12. **Int11 – Event set has occurred:** when last defined event has been occurred

After power-on, the MPL interrupts are globally enabled together with the first 4 interrupts: **Int 0** to **Int 3**. For **Int 2**, all the protections are activated, except over temperature motor, which depends on the presence or not of a temperature sensor on the motor; hence this protection may or may not be activated. For each of these 4 interrupts there is a default ISR which is executed when the corresponding interrupt occurs. You can view the contents of the default ISR in the [MPL Interrupt Service Routines](#) view. From this view you may also modify the default ISR for these interrupts and/or define ISR for the other MPL interrupts.

Before using other MPL interrupts, you need to enable them from this dialogue. Keep in mind that the interrupt flags are set independently of the activation or not of the MPL interrupts. Therefore, as a general rule, before enabling an interrupt, reset the corresponding flag. This operation will avoid triggering an interrupt immediately after activation, due to an interrupt flag set in the past.

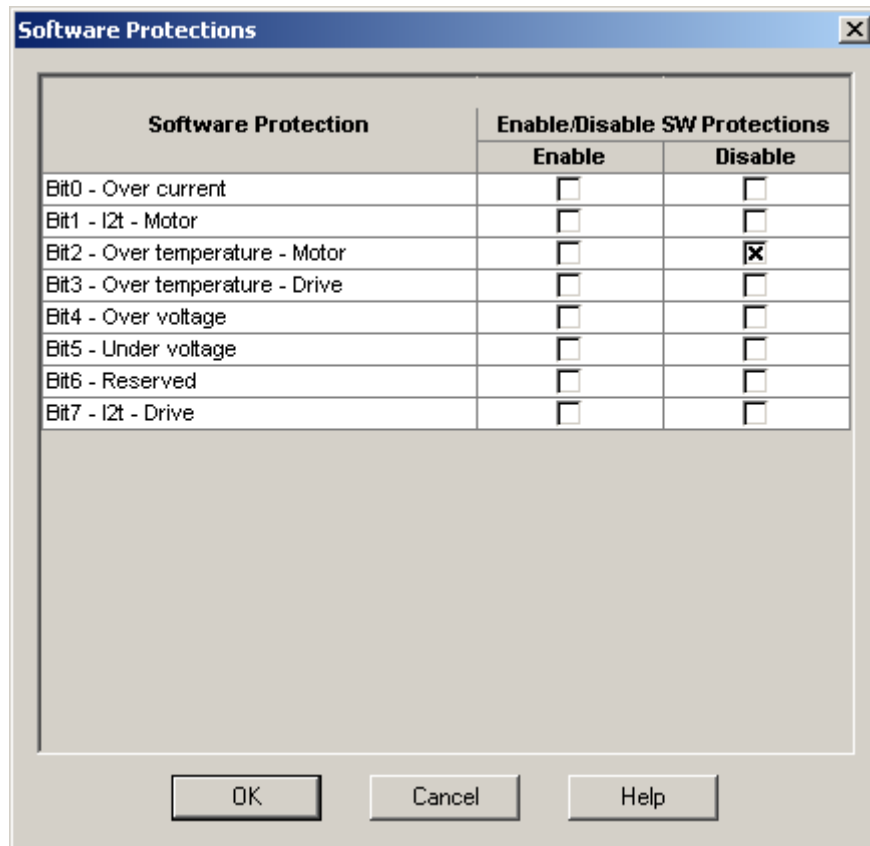
Remarks:

- *On entry in an ISR, the MPL interrupts are globally disabled. If you want to enable during the ISR execution any of the other interrupts, set accordingly the interrupt mask in the ICR register and insert the EINT instruction that globally enables the interrupts*

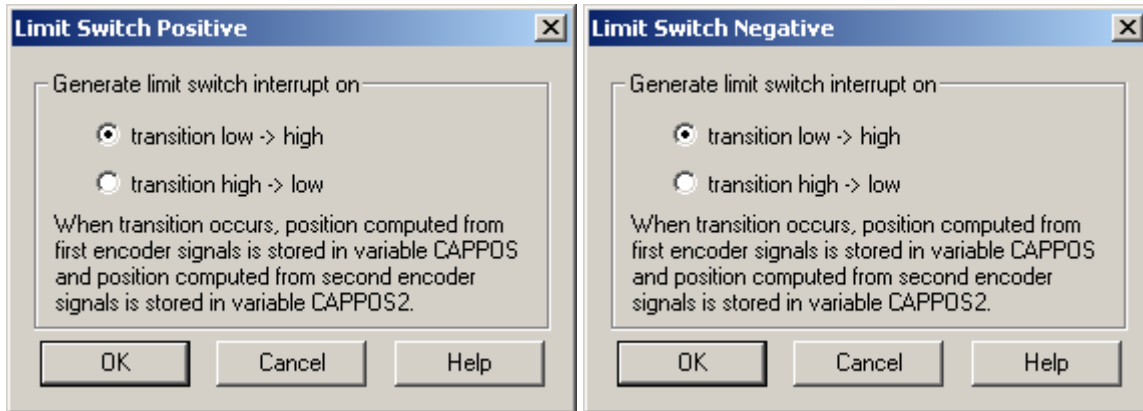
- The interrupt service routines are similar with the MPL functions, except for the return instruction: RETI (RETurn from Interrupt) instead of RET (RETurn from subroutine). Like the MPL functions, the MPL interrupt service routines must be positioned after the end of the main program. MotionPRO Developer handles automatically this aspect.

Check **Globally Enable MPL interrupts** to globally enable the MPL interrupts. Check **Globally Disable MPL interrupts** to globally disable the MPL interrupts. At **Enable/Disable MPL interrupt** choose one or several interrupts and select either **Enable** or **Disable** to activate or deactivate them. The status of the other interrupts remains unchanged. For the interrupts enables, check also **Reset previous MPL interrupt request** to reset the corresponding interrupt flag(s) set in the past.

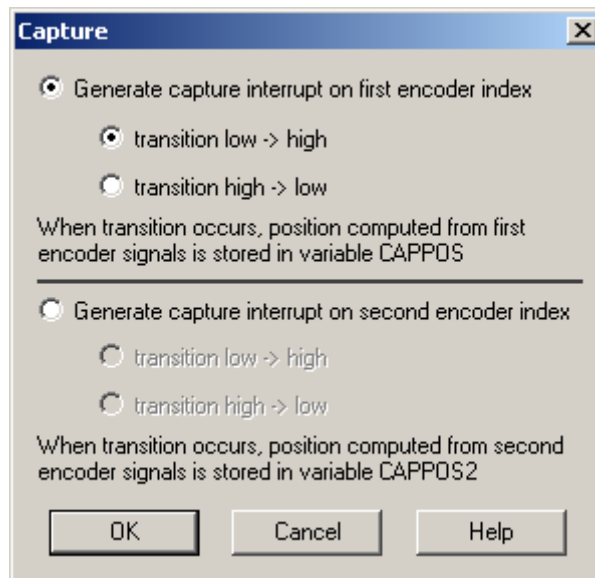
For **Int2 – Software protections**, select **Enable** and press **Details** to modify the status (enabled or disabled) of the protections triggering this interrupt.



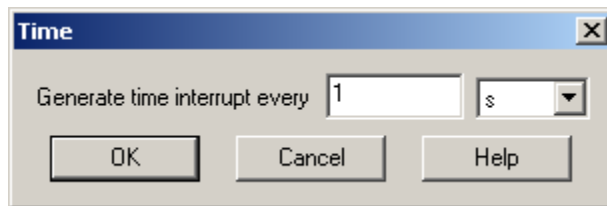
For **Int 6 - LSP programmed transition detected** and **Int 7 - LSN programmed transition detected** select **Enable** and press **Details** to select the monitored transition: high to low or low to high.



For **Int8 – Capture input transition detected** select **Enable** and press **Details** to select the monitored transition: high to low or low to high and the capture/encoder input to use: 1st or 2nd



For **Int10 – Time period has elapsed** select **Enable** and press **Details** to set the time period.



Remark: Some of the drive/motor protections may not work properly if the MPL Interrupts are handled incorrectly. In order to avoid this situation keep in mind the following rules:

- The MPL interrupts must be kept globally enabled to allow execution of the ISR for those MPL interrupts triggered by protections. As during a MPL interrupt execution, the MPL interrupts are globally disabled, you should keep the ISR as short as possible, without waiting loops. If this is not possible, you must globally enable the interrupts with EINT command during your ISR execution.
- If you modify the interrupt service routines for **Int 0** to **Int 4**, make sure that you keep the original MPL commands from the default ISR. Put in other words, you may add your own commands, but these should not interfere with the original MPL commands. Moreover, the original MPL commands must be present in all the ISR execution paths.

OK: Close this dialogue and save the interrupt settings in your motion sequence list.

Cancel: Close this dialogue without saving anything in your motion sequence list.

Help: Open this help page.

See also:

[MPL Interrupts – MPL Programming Details](#)

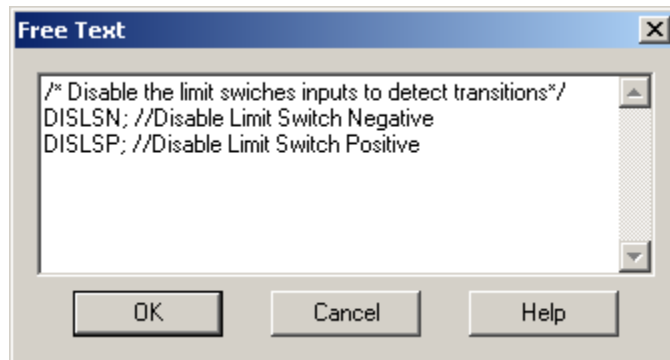
[MPL Interrupt Service Routines](#)

[Motion Programming](#)

6.1.25. Free text

The “Free Text” dialogue allows you to add comments to your MPL programs in order to improve their readability and therefore make them easier to understand and debug. A comment can include any characters. A multi line comment must start with “/*” and finish with “*/”. A single line comment can be preceded by “//”.

Through this dialogue you can also insert directly MPL commands, if you know their syntax. Note that all the MPL commands must ended with a semicolon “;”. Labels must start from the first column of a new line and end with a colon “:”. For readability, leave at least one space before starting a MPL command in a new line. This way you can quickly distinguish them from the labels.



Remark: *The motion dialogues cover all the MPL commands you typically need in an application. There is however a small number of MPL instructions that can't be generated from the motion dialogues and which may be used in some special cases. If ever needed, you can set these MPL commands via this dialogue.*

OK: Close this dialogue and add the comments / MPL commands in your motion sequence list.

Cancel: Close this dialogue without saving anything.

Help: Open this help page.

See also:

[Motion Programming](#)

6.2. ElectroCraft Motion Language

6.2.1. Basic Concepts

6.2.1.1. Overview

The ElectroCraft Motion Program Language (MPL) is a high-level language allowing you to:

- Setup a ElectroCraft Programmable [drive/motor](#) for a given application
- Program and execute motion sequences

The **setup part** consists in assigning the right values for the MPL registers and parameters. Through this process you can:

- Describe your application configuration (as motor and sensors type)
- Select specific operation settings (as motor start mode, PWM mode, sampling rates, etc.)
- Setup the controllers' parameters (current, speed, position), etc.

The output of the setup process is a set of values – the setup data – to be written in the MPL registers and parameters. The setup data can be:

- a) Stored in the drive/motor non-volatile EEPROM, from where it is automatically loaded into the MPL registers and parameters at power-on, if the data integrity check is passed
- b) Included at the beginning of a MPL program as a set of assignment instructions through which the MPL registers and parameters are initialized with the desired values

Remark: *PROconfig – the latest generation setup tool for ElectroCraft Programmable drives/motors – handles the setup process according with option a).*

The **motion programming part** allows you to:

- Set various motion modes (profiles, PVT, PT, electronic gearing or camming, etc.)
- Change the motion modes and/or the motion parameters
- Execute homing sequences
- Control the program flow through:
 - Conditional jumps and calls of MPL functions
 - MPL interrupts generated on pre-defined or programmable conditions (protections triggered, transitions on limit switch or capture inputs, etc.)
 - Waits for programmed events to occur
- Handle digital I/O and analogue input signals
- Execute arithmetic and logic operations
- Perform data transfers between axes
- Control motion of an axis from another one via motion commands sent between axes

-
- Send commands to a group of axes (multicast). This includes the possibility to start simultaneously motion sequences on all the axes from the group
 - Synchronize all the axes from a network

Due to a powerful instruction set, the motion programming in MPL is quick and easy even for complex motion applications. The result is a high-level motor-independent program which once conceived may be used in other applications too.

Basic Concepts next topics:

[MPL Environment](#)

[Program Execution](#)

[MPL Program Structure](#)

[MPL Instruction Coding](#)

[MPL Data](#)

[Memory Map – Firmware version FAxx](#)

[Memory Map – Firmware version FBxx](#)

[AUTORUN mode](#)

See also:

[MPL Description](#)

6.2.1.2. MPL Environment

The MPL environment includes three basic components:

1. “MPL processor”
2. Trajectory generator
3. Motor control kernel

The software-implemented “MPL processor” represents the core of the MPL environment. It decodes and executes the MPL commands. Like any processor, it includes specific elements as program counter, stack, ALU, interrupt management and registers.

The trajectory generator computes the position, speed, torque or voltage reference at each sampling step, depending on the selected motion mode.

The motor-control kernel implements the control loops including: the acquisition of the feedback sensors, the controllers, the PWM commands, the protections, etc.

When the “motion processor” executes a motion command, it translates them into actions upon the trajectory generator and/or the motor control kernel.

Basic Concepts next topics:

[Program Execution](#)

[MPL Program Structure](#)

[MPL Instruction Coding](#)

[MPL Data](#)

[Memory Map– Firmware Fx](#)

[Memory Map – Firmware FBx](#)

[AUTORUN mode](#)

See also:

[Basic Concepts](#)

[MPL Description](#)

6.2.1.3. Program Execution

The MPL programs are executed sequentially, one instruction after the other. A 16-bit instruction pointer (IP) controls the program flow. As the binary code of a MPL instruction may have up to 5 words, during its execution the IP is increased accordingly. When the execution of a MPL instruction ends, the IP always points to the next MPL instruction, or more exactly to the first word of its binary code.

The sequential execution may be interrupted by one of the following causes:

- A MPL command received through a communication channel (on-line commands);
- A branch to the interrupt service routine (ISR) when a MPL interrupt occurs;
- The need to send the master position to the slave axes when the current axis is set as master for electronic gearing or camming
- A **GOTO**, **CALL** or **CALLS** instruction;
- A return from a MPL function – **RET** or from a MPL interrupt – **RETI**;
- During the execution of the instructions: **WAIT!** (wait event), **SEG** (new contour segment), **PVTP** or **PT** (new PVT or PT point) if the buffer is full, and data transfers between axes of type `local_variable = [x]remote_variable`, which all keep the IP unchanged (i.e. loop on the same instruction) until a specific condition is achieved
- After execution of the **END** instruction.

The on-line commands have the highest priority and act like interrupts: when an on-line command is received through any communication channel, it starts to be executed immediately after the current MPL instruction is completed.

If an on-line command is received during a wait loop, the wait loop is temporary suspended, to permit the execution of the on-line command.

The MPL works with 3 types of commands, presented in table below.

Type of MPL commands

MPL Command Type	Execution	
	From a MPL program	Sent via communication
Immediate	✓	✓
Sequential	✓	–
On-line	–	✓

The immediate commands may be sent via a communication channel, or can reside in a MPL program. These commands don't require any wait loops to complete. Their execution is straightforward and can't be interrupted by other MPL commands.

The sequential commands require a wait loop to complete i.e. will not permit the IP to advance until the wait condition becomes true. In this category enter commands like:

```
WAIT! ; // Wait a programmed event to occur

SEG Time, Increment; // Set a contour segment with parameters Time and
                      //Increment to be executed when the previous one ends

local_variable = [x]remote_variable; // Get value of remote_variable from
```

`//axis x and put it in local_variable`

The sequential commands can reside only in a MPL program saved in the local memory.

Remark: *If a sequential command is sent via a communication channel, it is immediately executed as if the wait loop condition is always true.*

The on-line commands may be sent only via a communication channel. These commands can't be included in a MPL program. The on-line commands do not have an associated mnemonic and syntax rules as they do not need to be recognized by the MPL compiler. Their code is known only by the "MPL processor".

Remark: *Some of the on-line commands are implemented in debugging tools like the **Command Interpreter**, which was specifically designed to allow sending commands via a communication channel. These commands are presented with a "mnemonic" like that used in the Command Interpreter. The **Command Interpreter** is a component present in all the ElectroCraft applications for [drives/motors](#) setup and MPL Programming: **PROconfig**, **MotionPRO Developer**.*

Basic Concepts next topics:

[MPL Program Structure](#)

[MPL Instruction Coding](#)

[MPL Data](#)

[Memory Map – Firmware version FAxx](#)

[Memory Map – Firmware version FBxx](#)

[AUTORUN mode](#)

See also:

[Basic Concepts](#)

[MPL Description](#)

6.2.1.4. MPL Program Structure

The main section of a MPL program starts with the instruction **BEGIN** and ends with the instruction **END**. It is divided into two parts:

- Setup part
- Motion programming part

The setup part starts after **BEGIN** and lasts until the **ENDINIT** instruction, meaning “END of INITIALIZATION”. This part of the MPL program consists mainly of assignment instructions, which shall set the MPL registers and the MPL parameters in accordance with your application data. When the **ENDINIT** command is executed, key features of the MPL environment are initialized according with the setup data. After the **ENDINIT** execution, the basic configuration involving the motor and sensors types or the sampling rates cannot be changed unless a reset is performed.

***Remark:** The setup part can be void when setup data is saved in the EEPROM. In this case, the setup data is automatically loaded into the MPL registers and parameters, at power-on. However, even in this case in some situations it may still be necessary to perform some setup operations like:*

- Copy of cam tables from the [drive/motor](#) EEPROM into the working RAM memory
- Copy of the whole MPL program into the RAM in special cases where the EEPROM memory can't be used during run time

The motion programming part starts after the **ENDINIT** instruction until the **END** instruction. All the MPL programs (the main section) should end with the MPL instruction **END**. When **END** instruction is encountered, the sequential execution of a MPL program is stopped.

Apart from the main section, a MPL program also includes the MPL interrupt vectors table, the interrupt service routines (ISRs) for the MPL interrupts and the MPL functions. A typical structure for a MPL program is presented in figure below.

Typical structure of a MPL Program

```
BEGIN;           // program start
...
                // Setup part of the main section
...
ENDINIT;        // end of initialization
...
                // Motion programming part of the main section
...
END;            // end of the main section

InterruptTable: // start of the interrupt vectors table
@Int0_Axis_disable_ISR;
@Int1_PDPINT_ISR;
@Int2_Software_Protection_ISR;
@Int3_Control_Error_ISR;
@Int4_Communication_Error_ISR;
@Int5_Wrap_Around_ISR;
@Int6_Limit_Switch_Positive_ISR;
@Int7_Limit_Switch_Negative_ISR;
@Int8_Capture_ISR;
@Int9_Motion_Complete_ISR;
@Int10_Update_Contour_Segment_ISR;
@Int11_Event_Reach_ISR;
Int0_Axis_disable_ISR: // Int0_Axis_disable_ISR body
...
    RETI;             // RETURN from ISR
...
Int11_Event_Reach_ISR: // Int11_Event_Reach_ISR body
...
    RETI;             // RETURN from ISR
Function1:            // Start of the first function named Function1
...
    RET;              // RETURN from function named Function 1
...
FunctionX:            // Start of the last function named FunctionX
...
    RET;              // RETURN from the last function named Function X
```

Basic Concepts next topics:

[MPL Instruction Coding](#)

[MPL Data](#)

[Memory Map – Firmware version FAxX](#)

[Memory Map – Firmware version FBxX](#)

[AUTORUN mode](#)

See also:

[Basic Concepts](#)

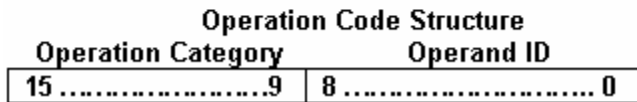
[MPL Description](#)

6.2.1.5. MPL Instruction Coding

The MPL instruction code consists of 1 to 5, 16-bit words. The first word is the operation code. The rest of words (if present) represent the instruction data words. The operation code is divided into two fields: Bits 15-9 represent the code for the operation category.

For example all MPL instructions that perform addition of two integer variables share the same operation category code. The remaining bits 8-0 represent the operand ID that is specific for each instruction.

Operation Code
Data (1)
...
Data (4)



Basic Concepts next topics:

[MPL Data](#)

[Memory Map – Firmware version Fx](#)

[Memory Map – Firmware version FBx](#)

[AUTORUN mode](#)

See also:

[Basic Concepts](#)

[MPL Description](#)

6.2.1.6. MPL Data

The MPL works with the following categories of data:

- [MPL Registers](#)
- [MPL Parameters](#)
- [MPL Variables](#)
- [User Variables](#)

All MPL data are identified by their name. The names of the MPL registers, parameters or variables are predefined and do not require to be declared. The names of the user variables are at your choice. You need to declare the user variables before using them.

The MPL uses the following data types:

- `int` 16-bit signed integer
- `uint` 16-bit unsigned integer
- `fixed` 32-bit fixed-point data with the 16MSB for the integer part and the 16LSB for the fractionary part.
- `long` 32-bit signed integer
- `ulong` 32-bit unsigned integer

The data type `uint` or `ulong` are reserved for the MPL predefined data. The user-defined variables are always signed. Hence you may declare them of type: `int`, `fixed` or `long`.

Remark: *An unsigned MPL data means that in the firmware its value is interpreted as unsigned. Typical examples: register values, time-related variables, protection limits for signals that may have only positive values like temperature or supply voltage, etc. However, the same data will interpreted as signed if it is used in a MPL instruction whose operands are treated as signed values.*

Each MPL data has an associated address. This represents the address of the data memory location where the MPL data exists. Address ranges for MPL registers, variables and parameters are from 0x0200 to 0x03AF and from 0x0800 to 0x09FF. For user-defined variables the address range is between 0x03B0 and 0x03FF. In MPL the data components may be addressed in several ways:

- **direct**, using their name in the MPL instruction mnemonic

Example:

```
CPOS = 2000; // write 2000 in CPOS parameter (command position)
```

- **indirect**, using a pointer variable. The pointer value is the address of the data component to work with

Example:

```
user_var = 0x29E;            // write hexadecimal value 0x29E representing CPOS address in  
                              // the user-defined pointer variable user_var  
(user_var), dm = 2000;      // write 2000 in the data memory address pointed by  
                              // user_var i.e. in the CPOS parameter
```

- **direct with extended address**, using the MPL data name

Example:

```
CPOS, dm = 2000; // write 2000 in CPOS using direct mode with extended address
```

In the MPL instructions the operands (variables) are grouped into 2 categories:

- **V16.** In this category enter all the 16-bit data from all the categories: MPL registers, MPL parameters, MPL variables, and user parameters. From the execution point of view, the MPL makes no difference between them.
- **V32.** In this category enter all the 32-bit data either long or fixed from all the categories: MPL registers, MPL parameters, MPL variables, and user parameters. From the execution point of view, the MPL makes no difference between them.

Remarks:

- *It is possible to address only the high or low part of a 32-bit data, using the suffix (H) or (L) after the variable name.*

Examples:

```
CPOS(L) = 0x4321; // write hexadecimal value 0x4321 in low part of CPOS
CPOS(H) = 0x8765; // write hexadecimal value 0x8765 in high part of CPOS
// following the last 2 commands, CPOS = 0x87654321
```

- *The MPL compiler always checks the data type. It returns an error if an operand has an incompatible data type or if the operands are not of the same type*
- *A write operation using indirect addressing is performed on one or two words function of the data type. If the data is a 16-bit integer, the write is done at the specified address. If the data is fixed or long the write is performed at the specified address and the next one. A fixed data is recognized by the presence of the dot, for example: 2. or 1.5. A long variable is automatically recognized when its size is outside the 16-bit integer range or in case of smaller values by the presence of the suffix L, for example: 200L or -1L.*

Examples:

```
user_var = 0x29E; // write CPOS address in pointer variable user_var
(user_var), dm = 1000000; // write 1000000 (0xF4240) in the CPOS parameter i.e.
// 0x4240 at address 0x29E and 0xF at next address
0x29F
(user_var), dm = -1; // write -1 (0xFFFF) in CPOS(L). CPOS(H) remains unchanged
(user_var), dm = -1L; // write -1 seen as a long variable (0xFFFFFFFF) in CPOS i.e.
// CPOS(L) = 0xFFFF and CPOS(H) = 0xFFFF
user_var = 0x2A0; // write CSPD address in pointer variable user_var
(user_var), dm = 1.5; // write 1.5 (0x18000) in the CSPD parameter i.e.
// 0x8000 at address 0x2A0 and 0x1 at next address 0x2A1
```

- *In an indirect addressing, if the pointer variable is followed by + sign, it is automatically incremented by 1 or 2 depending on the data type: 1 for integer, 2 for fixed or long data.*

Examples:

```
user_var = 0x29E; // write CPOS address in pointer variable user_var
(user_var+),dm = 1000L; // write 1000 seen as long in CPOS, then increment
                    // user_var by 2
(user_var+),dm = 1000; // write 1000 seen as int at address 0x29A (0x29E+2) ,
                    // then increment user_var by 1
```

MPL Data categories:

[MPL Registers](#)

[MPL Parameters](#)

[MPL Variables](#)

[User Variables](#)

Basic Concepts next topics:

[Memory Map – Firmware version FAxx](#)

[Memory Map – Firmware version FBxx](#)

[AUTORUN mode](#)

See also:

[Basic Concepts](#)

[MPL Description](#)

6.2.1.6.1. MPL Registers

There are 3 categories of MPL registers:

- Configuration registers
- Command registers
- Status registers

The configuration registers contain essential configuration information like motor and sensors type, or basic operation settings like PWM mode, motor start method, etc. The configuration registers must be set up during the setup part before the **ENDINIT** instruction

The command registers hold configuration settings that may be changed during motion. These settings refer to the activation/deactivation of software protections, to the use of MPL interrupts and to communication options.

The status registers provide information about [drive/motor](#) condition: errors and protections triggered, communication, active motion mode and control loops, MPL interrupts. The status registers can be used to detect events and to make decisions in a MPL program.

Configuration registers (R/W):

[ACR](#) – Auxiliary Control Register. Defines extra settings like: the configuration for automatic start and the external reference, operation options for the S-curve and the electronic camming modes.

[OSR](#) – Operating Settings Register. Defines some specific operating settings regarding motor control and data acquisition

[SCR](#) – System Configuration Register. Defines the basic application configuration regarding the motor type and the feedback sensors used

[UPGRADE](#) – Upgrade Register. Defines new options and extended features which are activated when their corresponding bits are set to 1

Command registers (R/W):

[CCR](#) – Communication Control Register. Contains settings for the SPI link with the EEPROM

[ICR](#) – Interrupt Control Register. Enables/disables MPL interrupts

[MCR](#) – Motion Command Register. Configures the motion modes: reference mode, active control loops, positioning type - absolute or relative, etc.

[PCR.7-0](#) – Protections Control Register. Activates different drive/motor protections like: over-current, I2t drive and motor, over- and under-voltage and over-temperature.

Status registers (RO):

[AAR](#) – Axis Address Register. Contains the Axis ID and the group ID of the drive/motor.

[CBR](#) – CAN Baud rate Register. Contains the current settings for CANbus baud-rate.

[CER](#) – Communication Error Register. Contains error flags for the communication channels.

[CSR](#) – Communication Status Register. Contains status flags for the communication channels.

[ISR](#) – Interrupt Status Register. Contains interrupt flags set by the MPL interrupt conditions.

MER – Motion Error Register. Groups all the errors conditions.

MSR – Motion Status Register. Gives indications about the motion status and some specific events like: control error condition, position wrap-around, limit switches and captures triggered by programmed transitions, etc.

PCR.14-8 – Protections Control Register. Contains flags set by the protections set in PCR.7-0.

SRL – Status Register Low. Low part of a 32-bit register grouping together all the key status information concerning the drive/motor

SRH – Status Register High. High part of a 32-bit register grouping together all the key status information concerning the drive/motor

SSR – Slave Status Register. Groups initialization information related to slave axes commanded by the motion controller

The MPL registers are treated like any other MPL parameter or variable in the MPL program. The configuration and command registers may be read or written. The status registers may only be read.

Remark: *The setup tools set automatically the configuration and command registers. The most important status information is grouped in 2 registers: **MER** - the Motion Error Register and **SRL, SRH** – the Status Register Low and High part. They have been specifically designed to provide you all the key information about the drive/motor status.*

See also:

[MPL Data](#)

[MPL Parameters](#)

[MPL Variables](#)

[User Variables](#)

6.2.1.6.2. MPL Parameters

The MPL parameters allow you to setup the parameters of the MPL environment according with your application data. Though most of the MPL parameters have their own address, there are some that share the same memory address. They are used in application configurations that exclude each other, and thus are not needed at the same time.

Some MPL parameters must be setup during the initialization phase. They are used to define the real-time kernel, including the PWM frequency and the control loops sampling periods, and should not be changed after the execution of the **ENDINIT** command. The other parameters can be initialized, used and changed any time, before or after the **ENDINIT** command.

See also:

[MPL Data](#)

[MPL Registers](#)

[MPL Variables](#)

[User Variables](#)

6.2.1.6.3. MPL Variables

The MPL variables provide you status information about the MPL environment like the motor position, speed and current, the position, speed and current commands, etc. These values may be used to take decisions in the motion program or for analysis and debug.

The MPL variables are read-only (RO). Modifying their value during motion execution may cause an improper operation of the drive/motor. There are however, specific situations when some MPL variables may also be written (R/W variables).

Most of the MPL variables are internally initialized after power-on, or during the setup phase up to the execution of the **ENDINIT** command.

Activating the on-chip logger module, real-time data tracking can also be implemented for any of these variables.

See also:

[MPL Data](#)

[MPL Registers](#)

[MPL Parameters](#)

[User Variables](#)

6.2.1.6.4. MPL User Variables

Besides the MPL pre-defined variables, you can also define your own user variables. You can use your variables in any MPL instruction accepting variables of the same type.

The user variables type can be: integer, fixed (point) or long (integer) (see table below).

MPL data type

Type	Format	Representation	Range
Int	Signed integer	16 bits	-32768 , 32767 (0x8000 , 0x7FFF)
Long	Signed long integer	32 bits	-2147483648 , 2147483647 (0x80000000 , 0x7FFFFFFF)
Fixed	(Integer part).(fractional part)	32 bits	-32768.999969 , 32767.999969 (0xFFFF.FFFF , 0x7FFF.FFFF)

The address of the user variables is automatically set in the order of declaration starting with 0x03B0. First integer variable takes address 0x3B0, next one 0x3B1, etc. An `int` variable takes one memory location. A `long` or `fixed` variable takes 2 consecutive memory locations. In this case the variable address is the lowest one.

Example:

```
int user_var1;           // user_var1 address is 0x3B0
long user_var2;         // user_var2 address is 0x3B1
fixed user_var3;        // user_var3 address is 0c3B3
int user_var4;          // user_var4 address is 0x3B5
```

Remark: you have to declare a user variable before using it first time.

See also:

[MPL Data](#)

[MPL Registers](#)

[MPL Parameters](#)

[MPL Variables](#)

6.2.1.7. Memory Map - Firmware FAxx

ElectroCraft [drives/motors](#) work with 2 separate address spaces: one for MPL programs and the other for data. Each space accommodates a total of 64K 16-bit word.

The first 16K of the MPL program space (0 to 3FFFh) is reserved and can't be used. The next 16K, from 4000h to 7FFFh are mapped to a serial SPI-connected EEPROM with the maximum size 32K bytes (seen as 16K 16-bit words). The exact amount of EEPROM memory is specific for each drive/motor. This space is used to store MPL programs, cam tables, the setup data and the product ID.

The recommended way to use the EEPROM memory space is:

- MPL programs from the beginning of the EEPROM, starting with first address 4000h
- Cam tables, after the MPL program, until the beginning of the setup data
- Setup data and product ID. Other data until the end of the EEPROM

Remarks:

- *The space needed for the setup data and the product ID is automatically computed by PROconfig*
- *The overall dimension of a MPL program includes apart from the main section, the MPL interrupt vectors table, the interrupt service routines (ISRs) for the MPL interrupts and the MPL functions*

For most of the ElectroCraft drives/motors, the next 2K of the MPL program space from 8000h to 87FFh represent the drive/motor internal RAM memory. From it, the first 270h, from 8000h to 826Fh are reserved for the internal use. The rest from 8270h to 87FFh may be used to temporary store MPL programs. The remaining MPL program space from 8800h to FFFFh is invalid. Some ElectroCraft drives have an extended internal RAM going from 8000h to FFFFh. From it, the first 270h are reserved for the internal use. In this case, the MPL programs space goes from 8270h to FFFFh.

The data memory space is used to store the MPL data (registers, parameters, variables), the cam tables during runtime (after being copied from the EEPROM memory) and for data acquisitions. The MPL data are stored in a reserved area, while the others are using the same internal RAM memory used for MPL programs. Though physically the RAM memory is the same for both, the MPL programs and data, the first 2K are mapped at different address ranges: The MPL program space from 8000h to 87FFh is seen in the data space from 800h to 9FFh. As the first 270h from it are reserved, the effective data memory space goes from A70h to FFFh. Apart from this space, the drives with extended internal RAM have another 32k of data memory, from 0x8000 to 0xFFFF.

Remark: *As the same RAM memory is used both for MPL programs and for data, it is the user responsibility to decide how to split these spaces in order to avoid their overlap.*

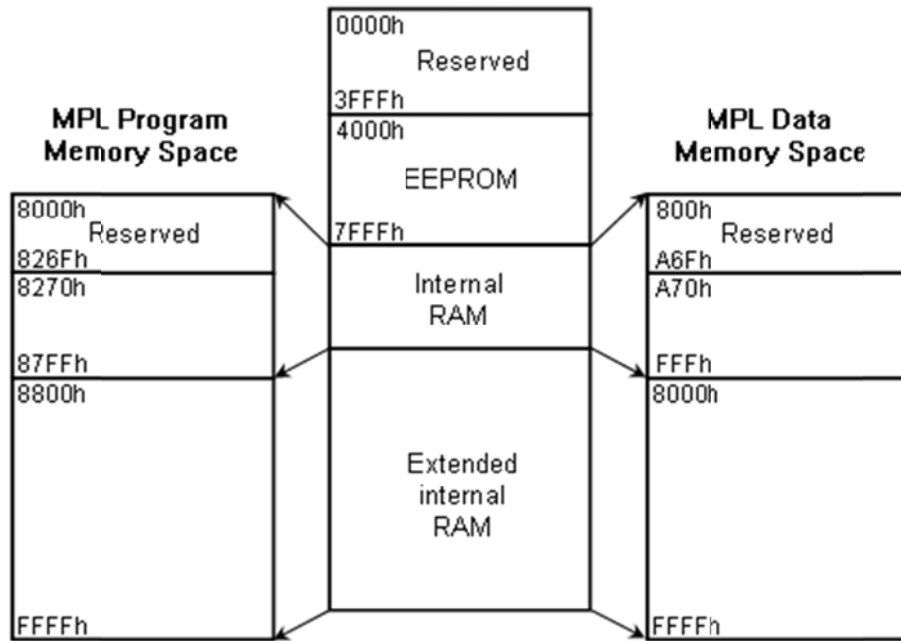
The recommended way to use the RAM memory (both for MPL programs and data) is:

- MPL programs from the beginning of the SRAM memory
- Data acquisitions, after the MPL programs
- Cam tables, after data acquisitions, until the end of the RAM

In the case of the drives/motors with normal RAM memory, you should start by checking if or how much space you need to reserve for cam tables, and use the rest of the SRAM for data acquisitions. As concerns the MPL programs, it is highly preferable to store them in the EEPROM.

Remark: *In configurations with feedback devices like the SSI or EnDat encoders, the MPL programs must execute from SRAM memory. This is because these feedback devices are using the same SPI interface to read the feedback position like the EEPROM, which is disabled after the execution of*

ENDINIT command. Therefore, at power-on, the MPL program needs to be copied from the EEPROM into the RAM where it is executed



Basic Concepts next topics:

[AUTORUN mode](#)

See also:

[Basic Concepts](#)

[MPL Description](#)

6.2.1.8. Memory Map - Firmware FBxx

ElectroCraft drives/motors work with 2 separate address spaces: one for MPL programs and the other for data. Each space accommodates a total of 64K 16-bit word.

The first 16K of the MPL program space (0 to 3FFFh) is reserved and can't be used. The next 16K, from 4000h to 7FFFh are mapped to a serial SPI-connected EEPROM with the maximum size 32K bytes (seen as 16K 16-bit words). The exact amount of EEPROM memory is specific for each drive/motor. This space is used to store MPL programs, cam tables, the setup data and the product ID.

The recommended way to use the EEPROM memory space is:

- MPL programs from the beginning of the EEPROM, starting with first address 4000h
- Cam tables, after the MPL program, until the beginning of the setup data
- Setup data and product ID. Other data until the end of the EEPROM

Remarks:

- *The space needed for the setup data and the product ID is automatically computed by MotionPRO Developer*
- *The overall dimension of a MPL program includes apart from the main section, the MPL interrupt vectors table, the interrupt service routines (ISRs) for the MPL interrupts and the MPL functions*

For most of the ElectroCraft drives/motors, the next 4K of the MPL program space, from 9000h to 9FFFh, represents the drive/motor internal RAM memory. The memory space may be used to temporary store MPL programs.

The data memory space is used to store the PVT buffer, the cam tables during runtime (after being copied from the EEPROM memory) and for data acquisitions. The MPL data are stored in a reserved area, while the others are using the same internal RAM memory used for MPL programs.

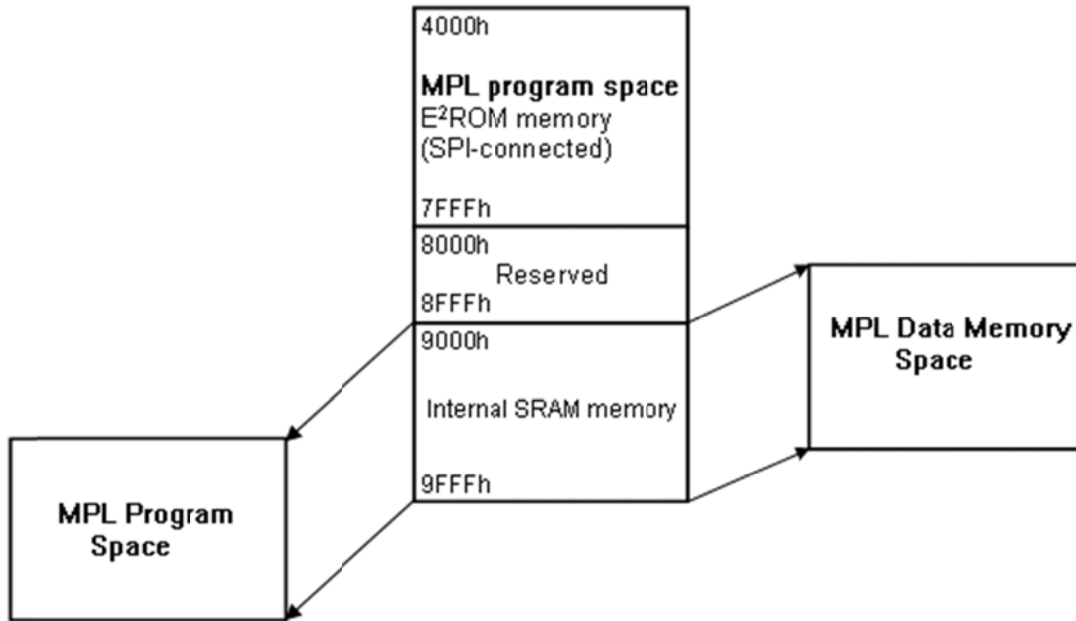
Remark: *As the same RAM memory is used both for MPL programs and for data, it is the user responsibility to decide how to split these spaces in order to avoid their overlap.*

The recommended way to use the RAM memory (both for MPL programs and data) is:

- MPL programs from the beginning of the SRAM memory
- Data acquisitions, after the MPL programs
- Cam tables, after data acquisitions, until the end of the RAM

You should start by checking if or how much space you need to reserve for cam tables, and use the rest of the SRAM for data acquisitions. As concerns the MPL programs, it is highly preferable to store them in the EEPROM.

Remark: *In configurations with feedback devices like the SSI or EnDat encoders, the MPL programs must execute from SRAM memory. This is because these feedback devices are using the same SPI interface to read the feedback position like the EEPROM, which is disabled after the execution of ENDINIT command. Therefore, at power-on, the MPL program needs to be copied from the EEPROM into the RAM where it is executed*



Basic Concepts next topics:

[AUTORUN mode](#)

See also:

[Basic Concepts](#)

[MPL Description](#)

6.2.1.9. AUTORUN Mode

The ElectroCraft [drives/motors](#) have 2 startup modes, at power on: **AUTORUN** and **slave**

In the **AUTO**(matic) **RUN**(ning) mode, the drive/motor reads the first EEPROM memory location at address 0x4000 and checks if the binary code is **0x649C** corresponding to the MPL instruction **BEGIN**. If this condition is true, the MPL program saved in the EEPROM memory is executed starting with the next instruction after **BEGIN**. If the condition is false, the drive/motor enters in the slave mode and waits to receive commands from a host via a communication channel. The AUTORUN mode, offers the possibility to execute automatically after power-on a MPL program saved into the drive/motor EEPROM memory.

In the slave mode, even if there is a valid MPL program in the EEPROM, this is not executed, because the drive/motor forces the execution of the **END** command which stops the MPL program execution.

Some of the ElectroCraft drives/motors are automatically set in the AUTORUN mode. Others have a dedicated switch or jumper through which you can set either the AUTORUN mode or the slave mode.

During a MPL program execution, a drive/motor can enter in the **slave** mode and stop the MPL program execution in the following cases:

- After the execution of the **END** command
- After receiving a **STOP** command from an external device, via a communication channel
- After an entering in [FAULT status](#), due to a protection triggered

Remark: When a drive/motor is set in AUTORUN mode, to change the MPL program you have to do to the following operations:

*Send via a communication channel the MPL command **END**, to stop the current program execution, followed by **AXISOFF** to disable the drive power stage*

Download the new MPL program

Reset the drive. The new MPL program will start to execute.

See also:

[Basic Concepts](#)

[MPL Description](#)

6.2.2. MPL Description

6.2.2.1. Overview

The MPL provides instructions for the following categories of operations:

- **Motion programming and control.** These instructions allow you to program ElectroCraft motion controllers or programmable [drives/motors](#) in order to set different motion modes and trajectories. These are divided into 2 categories function of how the motion reference is generated:
 - Motion modes with reference provided by an external device via an analog input, pulse & direction signals, a master encoder or via a communication channel
 - Motion modes with reference computed by the internal reference generator. In this category enter all the other motion modes

You can program one of the following motion modes:

- [Trapezoidal Position Profile](#)
- [Trapezoidal Speed Profile](#)
- [S-Curve Profile](#)
- [Position-Time \(PT\) Interpolated](#)
- [Position-Velocity-Time \(PVT\) Interpolated](#)
- [External](#)
- [Electronic Gearing \(alone or superposed with another motion mode\)](#)
- [Electronic Camming](#)
- [Homing](#)
- [Contouring](#)
- [Test](#)
- [Linear Interpolation](#)
- [Vector Mode](#)

and control their execution via a set of [Motor Commands](#).

Remark: *The Linear Interpolation and Vector Mode are coordinated motion modes available in applications developed for ElectroCraft Motion Controller.*

- **Program flow control.** In the MPL you can control the program execution in 3 ways:
 - By setting an [event](#) to be monitored and waiting the event occurrence
 - Through [jumps and MPL function calls](#)
 - Through the [MPL interrupts](#) which can be triggered in certain conditions
- **I/O handling**

Firmware [FAxx](#)

- [General-purpose I/O](#)
- [Special I/O](#): enable, capture and limit switch inputs

Firmware [FBxx](#)

- [General-purpose I/O](#)
- [Special I/O](#) MC3: enable, capture and limit switch inputs

- **Assignment and data transfer**

- [Setup 16 bit variable](#)
- [Setup 32 bit variable](#)

- **[Arithmetic and logic manipulation](#)**

- **Multi axis control**

- [Axis identification](#)
- [Axis synchronization](#)
- [Data transfer between axes](#)
- [Remote control](#)

- **Monitoring.** You can check the motion progress as well as the drive/motor status via

- [Position Triggers](#)
- [Status Register](#)
- [Error Register](#)
- [Messages sent to the host](#)

- **Slaves Management.** From the motion controller application you can perform:

- [Slaves Initialization](#)
- [Program events on slave axes and wait for their occurrence](#)
- [Homing and Function Calls from slave axes](#)
- [Slave Error Handling](#)

- **[Miscellaneous](#)** including:

- Declare user variable
- Reset [FAULT status](#)

-
- Save actual setup data from RAM into EEPROM in the setup table
 - Change the CAN bus and serial RS232 / RS485 communication settings

See also:

[Basic Concepts](#)

6.2.2.2. Motion programming – drives with built-in Motion Controller

6.2.2.2.1. Trapezoidal Position Profiles - MPL Programming Details

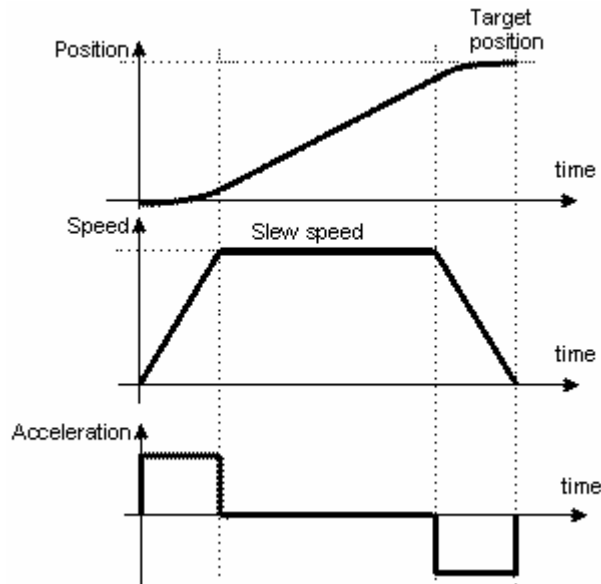
In the trapezoidal position profile, the load/motor is controlled in position. The built-in reference generator computes a position profile with a *trapezoidal* shape of the speed, due to a limited acceleration. You specify either a position to reach in absolute mode or a position increment in relative mode, plus the slew (maximum travel) speed and the acceleration/deceleration rate. In relative mode, the position to reach can be computed in 2 ways: standard (default) or additive. In standard relative mode, the position to reach is computed by adding the position increment to the instantaneous position in the moment when the command is executed. In the additive relative mode, the position to reach is computed by adding the position increment to the previous position to reach, independently of the moment when the command was issued. The additive relative mode is activated by setting **ACR.11** = 1.

Remarks:

- The motion mode and its parameters become effective after the execution of the update command **UPD**
- The additive relative mode is automatically disabled after an the update command **UPD**, which sets **ACR.11** = 0 restoring the standard relative mode

You can switch at any moment, including during motion, from another motion mode to the trapezoidal position profile. This operation is possible due to the target update mode 0 **TUM0** which is automatically activated when a new motion mode is set.

During motion, you can [change on the fly](#) the position command, the slew speed and the acceleration/deceleration rate. These changes become effective at next *update* command **UPD**.



Position profile with trapezoidal shape of the speed

Once set, the motion parameters are memorized. If you intend to use the same values as previously defined for the acceleration rate, the slew speed, the position increment or the position to reach, you don't need to set their values again in the following trapezoidal profiles.

Remark: The additive mode for relative positioning is not memorized and must be set each time a new additive relative move is set.

See also:

[Trapezoidal Position Profiles – Related MPL Instruction and Data](#)

[Trapezoidal Position Profiles – On the fly change of the motion parameters](#)

[Trapezoidal Position Profiles – Automatic elimination of round-off errors](#)

[MPL Description](#)

6.2.2.2.2. Trapezoidal Position Profiles - Related MPL Instructions and Data

Parameters

- CPOS** Command position – desired position (absolute or relative) for the load. Measured in [position units](#).
- CSPD** Command speed – desired slow speed for the load. The command speed can have only positive values. Measured in [speed units](#).
- CACC** Command acceleration – desired acceleration / deceleration for the load. The command acceleration can have only positive values. Measured in [acceleration units](#)
- ACR** Auxiliary Control Register – includes several MPL Programming options

Variables

- TPOS** Target load position – position reference computed by the reference generator at each slow loop sampling period. Measured in [position units](#)
- TSPD** Target load speed – speed reference computed by the reference generator at each slow loop sampling period. Measured in [speed units](#)
- TACC** Target load acceleration – acceleration/deceleration reference computed by the reference generator at each slow loop sampling period. Measured in [acceleration units](#)
- APOS_LD** Actual load position. Measured in [position units](#). Alternate name: [APOS](#)
- ASPD_LD** Actual load speed – measured in [speed units](#)
- APOS_MT** Actual motor position. Measured in [motor position units](#)
- ASPD_MT** Actual motor speed. Measured in [motor speed units](#) Alternate name: [ASPD](#)

Instructions

- CPR** Command position is relative
- CPA** Command position is absolute
- MODE_PP** Set position profile mode
- TUM1** Target Update Mode 1 (TUM1). Generates new trajectory starting from the actual values of position and speed reference (i.e. don't update the reference values with [load/motor](#) position and speed)
- TUM0** Target Update Mode 0 (TUM0). Generates new trajectory starting from the actual values of load/motor position and speed (i.e. update the reference values with load/motor position and speed)
- UPD** Update motion parameters and start new motion mode
- STOP** Stop the motion
- SRB** Set/reset bits from a MPL data

Remarks:

- **CSPD** and **CACC** must be positive. Negative values are taken in modulus
- The difference between **CPOS** and **TPOS** values in modulus must be maximum 231-1.
- The sum between **CSPD** and **CACC** values must be maximum 32767.99998 (0x7FFF.FFFF) i.e. the maximum value for fixed number
- Once a position profile is started, you can find when the motion is completed, by setting an event on motion complete and waiting until this event occurs.
- In order to activate the **TUM1** mode, execute the MPL instruction **TUM1** AFTER the **MODE PP** command and BEFORE the **UPD** command. When **MODE PP** is executed, it automatically sets **TUM0** mode. However, as the new motion mode becomes effective only after the **UPD** command, a **TUM1** command will overwrite the **TUM0** mode
- Under **TUM0** mode, at the **UPD** command **TPOS=APOS_LD** and **TSPD=ASPD_LD**. In open loop control of steppers, **TUM0** is ignored as there is no position and/or speed feedback
- In setup configurations where there is no transmission ratio between the motor and the load, it is supposed that these are directly connected. In these cases: **APOS_MT=APOS_LD** and **ASPD_MT=ASPD_LD**

Programming Example

```
// Position profile. Position feedback: 500 lines incremental
// encoder (2000 counts/rev)
CACC = 0.3183; //acceleration rate = 1000[rad/s^2]
CSPD = 33.3333; //slew speed = 1000[rpm]
CPOS = 6000; //position command = 3[rot]
CPR; //position command is relative
SRB ACR 0xFFFF, 0x800; // and additive
MODE PP; // set trapezoidal position profile mode
TUM1; //set Target Update Mode 1
UPD; //execute immediate
!MC; WAIT!; //wait for completion
```

See also:

[Trapezoidal Position Profiles – MPL Programming Details](#)

[Trapezoidal Position Profiles – On the fly change of the motion parameters](#)

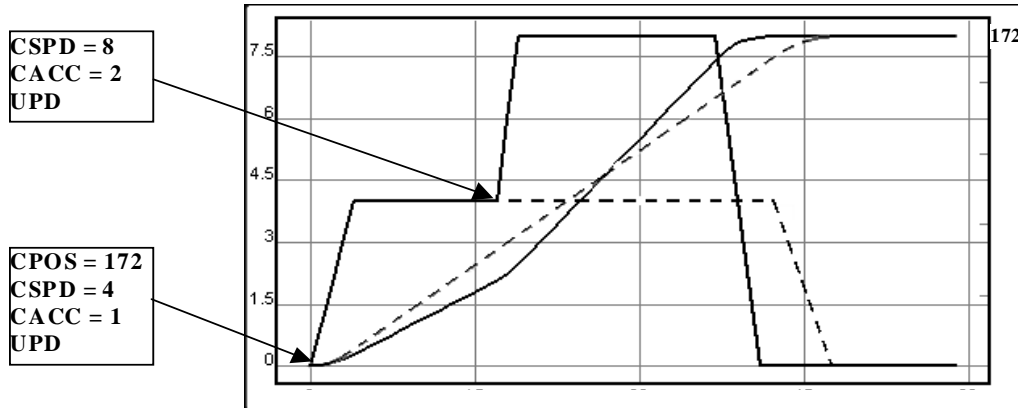
[Trapezoidal Position Profiles – Automatic elimination of round-off errors](#)

[MPL Description](#)

6.2.2.2.3. Trapezoidal Position Profiles - On the fly change of the motion parameters

In the trapezoidal position profile mode, the motion parameters **CPOS**, **CSPD**, **CACC** can be changed any time during motion. The reference generator automatically re-computes the position trajectory in order to reach the new commanded position, using the new values for slew speed and acceleration.

The figure below shows an example where slew speed and acceleration rate are changed, while the commanded position is kept the same.



Trapezoidal position profile. On-the-fly change of motion parameters

Programming Example

```
// Position profile already set. CACC and CSPD
// are changed during motion
CACC = 2; //acceleration rate = 2 [internal units]
CSPD = 8; //slew speed = 8 [internal units]
UPD; //execute immediate
```

If the trapezoidal position profile is already set and you intend to change only the motion parameters, you don't need to set again neither the motion mode with MPL instruction **MODE PP**, nor the target update mode 1 (when required) with MPL instruction **TUM1**.

If during motion, a new position command is issued that requires reversing the motor movement, the reference generator does automatically the following operations:

- stops the motor with the programmed deceleration rate
- accelerates the motor in the opposite direction till the slew speed is reached, or till the motor has to decelerate
- stops the motor on the commanded position

See also:

[Trapezoidal Position Profiles – MPL Programming Details](#)

[Trapezoidal Position Profiles – Related MPL Instruction and Data](#)

[Trapezoidal Position Profiles – Automatic elimination of round-off errors](#)

[MPL Description](#)

6.2.2.2.4. Trapezoidal Position Profiles - Automatic elimination of round-off errors

In trapezoidal position profile mode, the reference generator automatically eliminates the round-off errors, which may occur when the commanded position cannot be reached with the programmed slew speed and acceleration/deceleration rate. This situation is illustrated by the example below, where the position feedback is an incremental encoder. Therefore, the internal units for position are [encoder counts], for speed are [encoder counts / slow loop sampling], for acceleration are [encoder counts / square of slow loop sampling]

Example:

The commanded position is 258 counts, with the slew speed 18 counts/sampling and the acceleration rate 4 counts/sampling². To reach the slew speed, two options are available:

- Accelerate to 16 in 4 steps, then from 16 to 18 in a 5th step. Acceleration space is 49 counts
- Accelerate from 0 to 2 in 1st step, then from 2 to 18 in 4 steps. Acceleration space is 41 counts

For the deceleration phase, the options and spaces are the same. But, no matter which option is used for the acceleration and deceleration phases, the space that remains to be done at constant speed is not a multiple of 18, i.e. the position increment at each step.

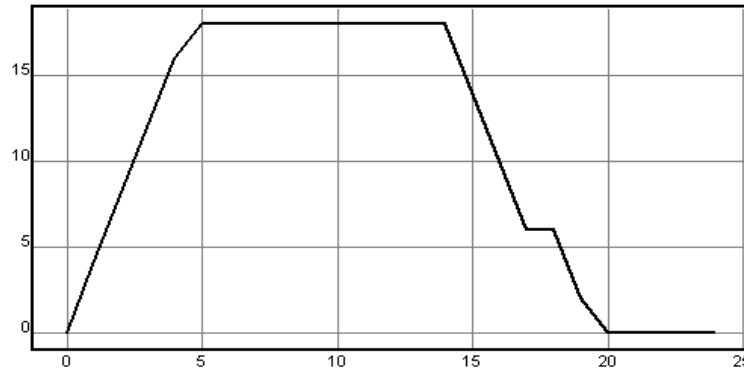
So, when to start the deceleration phase? Table below presents the possible options, and the expected errors.

Round-off error example. Options and expected errors.

Acceleration Space [counts]	Deceleration Space [counts]	Space to do at constant speed [counts]	Time to go at constant speed [sampling steps]	Deceleration starts after [samplings]	Target position Error [counts]
49 counts	49 counts	$258 - 2 * 49 = 160$ counts	$160/18 = 8.8$	$5 + 8 = 13$	- 16
				$5 + 9 = 14$	+ 2
49 counts	41 counts	$258 - 49 - 41 = 168$ counts	$168/18 = 9.3$	$5 + 9 = 14$	- 6
				$5 + 10 = 15$	+ 12
41 counts	49 counts	$258 - 41 - 49 = 168$ counts	$168/18 = 9.3$	$5 + 9 = 14$	- 6
				$5 + 10 = 15$	+12
41 counts	41 counts	$258 - 2 * 41 = 176$ counts	$176/18 = 9.7$	$5 + 9 = 14$	-14
				$5 + 10 = 15$	+4

MPL comes with a different approach. It monitors the round-off errors and automatically eliminates them by introducing, during deceleration phase, short periods where the target speed is kept constant. Hence, the target position is always reached precisely, without any errors.

CPOS=258
CSPD=18
CACC=4



Trapezoidal Position profile. Automatic elimination of round-off errors

The figure above shows the target speed generated by MPL for the above example. During the deceleration phase, the target speed:

- decelerates from 18 to 6 in 3 steps (target position advances by 36 counts)
- is kept constant for 1 step (target position advances by 6 counts)
- decelerates from 6 to 2 in one step (target position advances by 4 counts)
- decelerates from 2 to 0 in the last step (target position advances by 1 count)

Hence the deceleration space is 47 counts, which, added to 49 counts for acceleration phase and to the 162 counts for constant speed, gives exactly the 258-count commanded position.

See also:

[Trapezoidal Position Profiles – MPL Programming Details](#)

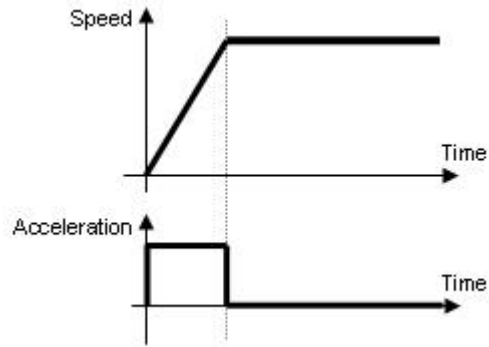
[Trapezoidal Position Profiles – Related MPL Instruction and Data](#)

[Trapezoidal Position Profiles – On the fly change of the motion parameters](#)

[MPL Description](#)

6.2.2.5. Trapezoidal Speed Profiles - MPL Programming Details

In the speed profile, the load/motor is controlled in speed. The built-in reference generator computes a speed profile with a *trapezoidal* shape, due to a limited acceleration. You specify the jog speed (speed sign specifies the direction) and the acceleration/deceleration rate. The load/motor accelerates until the jog speed is reached. During motion, you can change on the fly the slew speed and/or the acceleration/deceleration rate. The motion will continue until a **STOP** command. An alternate way to stop motion is to set the jog speed to zero.



Speed profile with trapezoidal shape

Remark: *The motion mode and its parameters become effective after the execution of the update command **UPD**. Changes of the slew speed and/or acceleration/deceleration rate also become effective at next update command.*

You can switch at any moment, including during motion, from another motion mode to the trapezoidal speed profile. This operation is possible due to the target update mode 0 **TUM0** which is automatically activated when a new motion mode is set.

Once set, the motion parameters are memorized. If you intend to use the same values as previously defined for the acceleration rate and the jog speed, you don't need to set their values again in the following trapezoidal profiles.

See also:

[Trapezoidal Speed Profiles – Related MPL Instructions and Data](#)

[MPL Description](#)

6.2.2.2.6. Trapezoidal Speed Profiles - Related MPL Instructions and Data

Parameters

CSPD Command speed – desired slow speed for the load. The sign specifies the direction. Measured in [speed units](#)

CACC Command acceleration – desired acceleration / deceleration for the load. The command acceleration can have only positive values. Measured in [acceleration units](#).

Variables

TPOS Target load position – position reference computed by the reference generator at each slow loop sampling period. TPOS is computed by integrating the target speed TSPD. Measured in [position units](#)

TSPD Target load speed – speed reference computed by the reference generator at each slow loop sampling period. Measured in [speed units](#)

TACC Target load acceleration – acceleration/deceleration reference computed by the reference generator at each slow loop sampling period. Measured in [acceleration units](#)

APOS_LD Actual load position. Measured in [position units](#). Alternate name: **APOS**.

ASPD_LD Actual load speed – measured in [speed units](#).

APOS_MT Actual motor position. Measured in [motor position units](#).

ASPD_MT Actual motor speed. Measured in [motor speed units](#). Alternate name: **ASPD**

Instructions

MODE SP Set speed profile mode

TUM1 Target Update Mode 1 (TUM1). Generates new trajectory starting from the actual values of position and speed reference (i.e. don't update the reference values with [load/motor](#) position and speed)

TUM0 Target Update Mode 0 (TUM0). Generates new trajectory starting from the actual values of load/motor position and speed (i.e. update the reference values with load/motor position and speed)

UPD Update motion parameters and start new motion mode

STOP STOP the motion.

Remarks:

- The sum between **CSPD** and **CACC** values must be maximum 32767.99998 (0x7FFF.FFFF) i.e. the maximum value for [fixed](#) number.
- After a **STOP** command or after setting jog speed command to zero, you can find when the motion is completed, by setting an [event on motion complete](#) and waiting until this event occurs.
- In order to activate the **TUM1** mode, execute the MPL instruction **TUM1** AFTER the **MODE SP** command and BEFORE the **UPD** command. When **MODE SP** is executed, it automatically sets **TUM0** mode. However, as the new motion mode becomes effective only after the **UPD** command, a **TUM1** command will overwrite the **TUM0** mode

-
- Under **TUM0** mode, at the **UPD** command **TPOS=APOS_LD** and **TSPD=ASPD_LD**. In open loop control of steppers, **TUM0** is ignored as there is no position and/or speed feedback
 - In setup configurations where there is no transmission ratio between the motor and the load, it is supposed that these are directly connected. In these cases: **APOS_MT=APOS_LD** and **ASPD_MT=ASPD_LD**

Programming Example

```
// Speed profile with position feedback on motor: 500 lines
// incremental encoder (2000 counts/rev)
CACC = 0.1591;//acceleration rate = 500[rad/s^2]
CSPD = 40;//jog speed = 1200[rpm]
MODE SP; // set trapezoidal speed profile mode
TUM1; //set Target Update Mode 1
UPD; //execute immediate
```

See also:

[Trapezoidal Speed Profiles – MPL Programming Details](#)

[MPL Description](#)

6.2.2.2.7. S-curve Profiles - MPL Programming Details

In the S-curve profile, the load/motor is controlled in position. The built-in reference generator computes a position profile with an *S-curve* shape of the speed. This shape is due to the jerk limitation, leading to a trapezoidal or triangular profile for the acceleration and an S-curve profile for the speed. You specify either a position to reach in absolute mode or a position increment in relative mode, plus the slew (maximum travel) speed, the maximum acceleration/deceleration rate and the jerk rate. The jerk rate is set indirectly via the *jerk time*, which represents the time needed to reach the maximum acceleration starting from zero.

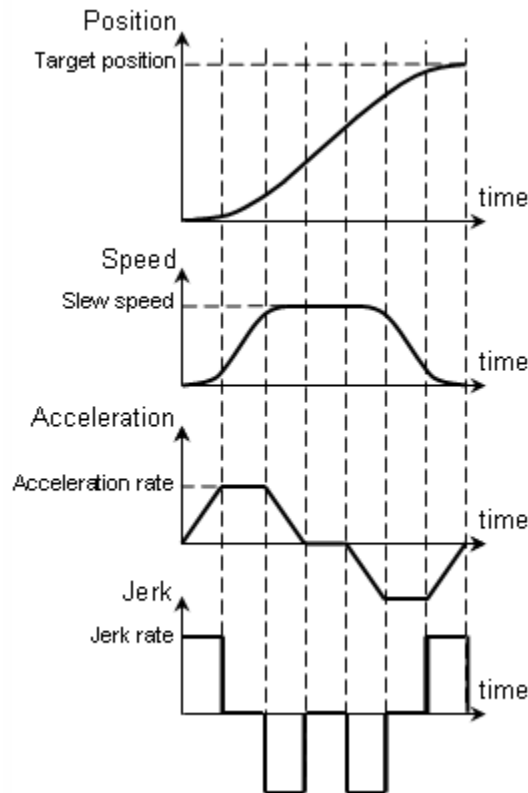
Remarks:

- The motion mode and its parameters become effective after the execution of the update command **UPD**
- The jerk rate results by dividing the maximum acceleration rate to the jerk time.

An *S-curve profile* must begin when load/motor is not moving. During motion the parameters should not be changed. Therefore when executing successive S-curve commands, you should wait for the previous motion to end before setting the new motion parameters and starting next motion. During an S-curve execution, you can switch at any moment to another motion mode (except PVT and PT interpolated modes) or stop the motion with a STOP command.

Following a STOP command, the deceleration phase can be done in 2 ways:

- Smooth, using an S-curve speed profile, when **ACR.1 = 0** (default), or
- Fast using a trapezoidal speed profile, when **ACR.1 = 1**



Position profile with S-curve shape of the speed

Once set, the motion parameters are memorized. If you intend to use the same values as previously defined for the acceleration rate, the slew speed, the position increment or the position to reach, you don't need to set their values again in the following trapezoidal profiles.

See also:

[S Curve Profile – Related MPL Instructions and Data](#)

[MPL Description](#)

6.2.2.2.8. S Curve Profile - Related MPL Instructions and Data

Parameters

CPOS Command position – desired position (absolute or relative) for the load. Measured in [position units](#)

CSPD Command speed – desired slew speed for the load. Measured in [speed units](#)

CACC Command acceleration – maximum desired acceleration / deceleration for the load. Measured in [acceleration units](#)

CDEC Command deceleration for quick stop mode. Measured in [acceleration units](#)

TJERK Jerk time needed to accelerate from zero up to the **CACC** value. Measured in [time units](#)

ACR Auxiliary Control Register – includes several MPL Programming options

Variables

TPOS Target load position – position reference computed by the reference generator at each slow loop sampling period. Measured in [position units](#)

TSPD Target load speed – speed reference computed by the reference generator at each slow loop sampling period. Measured in [speed units](#)

TACC Target load acceleration – acceleration/deceleration reference computed by the reference generator at each slow loop sampling period. Measured in [acceleration units](#)

APOS LD Actual load position. Measured in [position units](#). Alternate name: **APOS**

ASPD LD Actual load speed – measured in [speed units](#)

APOS MT Actual motor position. Measured in [motor position units](#).

ASPD MT Actual motor speed. Measured in [motor speed units](#). Alternate name: **ASPD**

Instructions

CPR Command position is relative

CPA Command position is absolute

MODE PSC Set S-curve mode.

UPD Update motion parameters and start new motion mode

STOP Stop the motion

SRB Set/reset bits from a MPL data

Remarks:

- **CSPD**, **CACC** and **TJERK** must be positive
- The difference between **CPOS** and **TPOS** values in modulus must be maximum 231-1.
- The sum between **CSPD** and **CACC** values must be maximum 32767.99998 (0x7FFF.FFFF) i.e. the maximum value for [fixed](#) number

-
- Once a position profile is started, you can find when the motion is completed, by setting an [event on motion complete](#) and waiting until this event occurs.
 - The S-curve profile uses always **TUM1** mode, i.e. preserves the values of **TPOS** and **TSPD**. If these values don't match with the actual feedback values, precede the S-curve command with another motion command accepting **TUM0** to update **TPOS** and **TSPD**. This command may be for example a trapezoidal profile that keeps position unchanged
 - In setup configurations where there is no transmission ratio between the motor and the load, it is supposed that these are directly connected. In these cases: **APOS_MT=APOS_LD** and **ASPD_MT=ASPD_LD**

Programming Example

```
// S-curve profile. Position feedback: 500 lines incremental
// encoder (2000 counts/rev)
TJERK = 50;//jerk = 2e+004[rad/s^3]
CACC = 0.3183;//acceleration rate = 1000[rad/s^2]
CSPD = 33.3333;//slew speed = 1000[rpm]
CPOS = 20000;//position command = 10[rot]
CPR; //position command is relative
MODE PSC; // set S-curve profile mode
SRB ACR, 0xFFFFE, 0x0000; //Stop using an S-curve profile
UPD; //execute immediate
!MC; WAIT!; //wait for completion
```

See also:

[S Curve Profile – MPL Programming Details](#)

[MPL Description](#)

6.2.2.2.9. Position-Time (PT) Interpolated - MPL Programming Details

In the PT motion mode the load/motor is controlled in position. The built-in reference generator computes a positioning path using a series of points. Each point specifies the desired **Position**, and **Time**, i.e. contains a **PT** data. Between the PT points the reference generator performs a linear interpolation.

The PT Interpolated mode is typically used together with a host, which sends PT points via a communication channel. Due to the interpolation, the PT mode offers the possibility to describe an arbitrary position contour using a reduced number of points. It is particularly useful when the motion reference is computed on the fly by the host like for example in vision systems. By reducing the number of points, both the computation power and the communication bandwidth needed are substantially reduced optimizing the costs. When the PT motion mode is used simultaneously with several [drives/motors](#) having the time synchronization mechanism activated, the result is a very powerful multi-axis system that can execute complex synchronized moves.

The PT motion mode can be started only when the previous motion is complete. However, you can switch at any moment to another motion mode. The PT mode can be relative (following a **CPR** command) or absolute (following a **CPA** command). In the absolute mode, each PT point specifies the position to reach. The initial position may be either the current position reference **TPOS** or a preset value read from the MPL parameter **PVTPOS0**. In the relative mode, each PT point specifies the position increment relative to the previous point. In both cases, the time is relative to the previous point i.e. represents the duration of a PT segment. For the first PT point, the time is measured from the starting of the PT mode.

Each time when a new PT point is read from a MPL program or received via a communication channel, it is saved into the PT buffer. The reference generator empties the buffer as the PT points are executed. The PT buffer is of type FIFO (first in, first out). The default length of the PT buffer is 7 PT points. The drive/motor automatically sends messages to the host when the buffer is full, low or empty. The messages contain the PT status (MPL variable **PVTSTS**). The host address is taken from the MPL parameter **MASTERID**. The buffer full condition occurs when the number of PT points in the buffer is equal with the buffer size. The buffer low condition occurs when the number of PT points in the buffer is less or equal with a programmable value – the low level. The buffer empty condition occurs when the buffer is empty and the execution of the last PT point is over. When the PT buffer becomes empty the drive/motor keeps the position reference unchanged.

Remarks:

- *The PVT and PT modes share the same buffer. Therefore the MPL parameters and variables associated with the buffer management are the same.*
- *Before activating the PT mode, you must place at least one PT point in the buffer*
- *The buffer low condition is set by default when the last PT point from the buffer is read and starts to be executed*
- *Both the PT buffer size and its start address are programmable via MPL parameters **PVTBUFBEGIN** and **PVTBUFLN**. Therefore if needed, the PT buffer size can be substantially increased.*

Each PT point also includes a 7-bit *integrity counter*. The integrity counter value must be incremented by the host by one, each time a new PT point is sent to the drive/motor. If the integrity counter error checking is activated, the drive compares its internally computed integrity counter value with the one sent with the PT point (i.e. with the PTP command). This comparison is done every time a PTP instruction is received. If the values of the two integrity counters do not match, the integrity check error is triggered, the drive/motor sends the **PVTSTS** to the host with **PVTSTS.12 = 1** and the received PT point is discarded. Each time a PT point is accepted (the integrity counters match or the integrity counter error checking is disabled), the drive automatically increments its internal integrity counter. The default value of the internal

integrity counter after power up is 0. Its current value can be read from the MPL variable **PVTSTS** (bits 6..0). The integrity counter can also be set to any value using MPL command **SETPT**.

See also:

[PT – Related MPL Instructions and Data](#)

[MPL Description](#)

6.2.2.2.10. Position-Velocity-Time(PVT) Interpolated - MPL Programming Details

In the PVT motion mode the load/motor is controlled in position. The built-in reference generator computes a positioning path using a series of points. Each point specifies the desired **Position**, **Velocity** and **Time**, i.e. contains a **PVT** data. Between the PVT points the reference generator performs a 3rd order interpolation.

The PVT Interpolated mode is typically used together with a host, which sends PVT points via a communication channel. Due to the 3rd order interpolation, the PVT mode offers the possibility to describe complex position contours using a reduced number of points. It is particularly useful when the motion reference is computed on the fly by the host like for example in vision systems. By reducing the number of points, both the computation power and the communication bandwidth needed are substantially reduced optimizing the costs. When the PVT motion mode is used simultaneously with several drives/motors having the time synchronization mechanism activated, the result is a very powerful multi-axis system that can execute complex synchronized moves.

A key factor for getting a correct positioning path in PVT mode is to set correctly the distance in time between the points. Typically this is 10-20ms, the shorter the better. If the distance in time between the PVT points is too big, the 3rd order interpolation may lead to important variations compared with the desired path.

The PVT motion mode can be started only when the previous motion is complete. However, you can switch at any moment to another motion mode. The PVT mode can be relative (following a **CPR** command) or absolute (following a **CPA** command). In the absolute mode, each PVT point specifies the position to reach. The initial position may be either the current position reference **TPOS** or a preset value read from the MPL parameter **PVTPOS0**. In the relative mode, each PVT point specifies the position increment relative to the previous point. In both cases, the time is relative to the previous point i.e. represents the duration of a PVT segment. For the first PVT point, the time is measured from the starting of the PVT mode.

Each time when a new PVT point is read from a MPL program or received via a communication channel, it is saved into the PVT buffer. The reference generator empties the buffer as the PVT points are executed. The PVT buffer is of type FIFO (first in, first out). The default length of the PVT buffer is 7 PVT points. Each entry in the buffer is made up of 9 words, so the default length of the PVT buffer in terms of how much memory space is reserved is 63 (3Fh) words. The drive/motor automatically sends messages to the host when the buffer is full, low or empty. The messages contain the PVT status (MPL variable **PVTSTS**). The host address is taken from the MPL parameter **MASTERID**. The buffer full condition occurs when the number of PVT points in the buffer is equal with the buffer size. The buffer low condition occurs when the number of PVT points in the buffer is less or equal with a programmable value – the low level. The buffer empty condition occurs when the buffer is empty and the execution of the last PVT point is over. When the PVT buffer becomes empty the drive/motor:

- Remains in PVT mode if the velocity of last PVT point executed is zero and waits for new points to receive
- Enters in quick stop mode if the velocity of last PVT point executed is not zero

Therefore, a correct PVT sequence must always end with a last PVT point having velocity zero.

Remarks:

- *The PVT and PT modes share the same buffer. Therefore the MPL parameters and variables associated with the buffer management are the same.*
- *Before activating the PVT mode, you must place at least one PVT point in the buffer*

-
- The buffer low condition is set by default when the last PVT point from the buffer is read and starts to be executed
 - Both the PVT buffer size and its start address are programmable via MPL parameters **PVTBUFBEGIN** and **PVTBUFLEN**. Therefore if needed, the PVT buffer size can be substantially increased.

Each PVT point also includes a 7-bit *integrity counter*. The integrity counter value must be incremented by the host by one, each time a new PVT point is sent to the drive/motor. If the integrity counter error checking is activated, the drive compares its internally computed integrity counter value with the one sent with the PVT point (i.e. with the PVTP command). This comparison is done every time a PVTP instruction is received. If the values of the two integrity counters do not match, the integrity check error is triggered, the drive/motor sends the **PVTSTS** to the host with **PVTSTS.12** =1 and the received PVT point is discarded. Each time a PVT point is accepted (the integrity counters match or the integrity counter error checking is disabled), the drive automatically increments its internal integrity counter. The default value of the internal integrity counter after power up is 0. Its current value can be read from the MPL variable **PVTSTS** (bits 6..0). The integrity counter can also be set to any value using MPL command **SETPVT**.

See also:

[PVT – Related MPL Instructions and Data](#)

[MPL Description](#)

6.2.2.2.11. Motion PT - MPL Instructions and Data

Parameters

MASTERID Contains the axis ID of the host/master where the drive/motor must send the PT messages. It must be set before starting the PT mode. The **MASTERID** value must be set as: host ID << 4 + 1, where host ID is a number between 1 and 255 representing the host ID. By default, after power-on the host ID is set equal with the drive address causing all the PT messages to be sent via RS-232

PVTBUFBEGIN Specifies the start address of the PT buffer

PVTBUFLEN Specifies the PT buffer length expressed in PT points

PVTPOS0 Specifies for absolute mode, the initial position from which to start computing the distance to move up to the first PT point. An alternate option is to consider **TPOS** as initial position. Selection between these 2 options is done at PT initialization via MPL command **SETPT**. The default value of **PVTPOS0** is 0.

PVTSENDOFF When set to 1, disables transmission of messages during PT mode. By default is set to 0 and the transmission is enabled

Variables

PVTSTS Contains the PT motion mode status.

PVTSTS bit description

BIT	VALUE	DESCRIPTION
15	0	PT buffer is not empty
	1	PT buffer is empty – there is no PT point in the buffer and the execution of the current PT segment is over. If PVTSENOFF = 0 (default), the drive/motor will send the PVTSTS each time this bit goes from 0 to 1
14	0	PT buffer is not low
	1	PT buffer is low – the number of PT points from the buffer is equal or less than the low limit set using SETPT . If PVTSENOFF = 0 (default), the drive will send the PVTSTS each time this bit goes from 0 to 1
13	0	PT buffer is not full
	1	PT buffer is full – the number of PT points from the buffer is equal with the buffer dimension. If PVTSENOFF = 0 (default), the drive will send the PVTSTS each time this bit goes from 0 to 1
12	0	No integrity counter error
	1	Integrity counter error. If the integrity counter error checking is enabled and PVTSENOFF = 0 (default), the drive will send the PVTSTS each time this bit goes from 0 to 1
11	0	Reserved
10	0	Normal operation. Data received are PT points
	1	A PVT point was received while PT mode is active. The PVT point was discharged. If PVTSENOFF = 0 (default), the drive/motor will send the PVTSTS each time this bit goes from 0 to 1
9..7	0	Reserved
6..0	0..127	Current integrity counter value

- PVTMODE** PVT operation mode as was set with the **SETPT** command
- TPOS** Target load position – position reference computed by the reference generator at each slow loop sampling period. Measured in [position units](#)
- TSPD** Target load speed – speed reference computed by the reference generator at each slow loop sampling period. Measured in [speed units](#)
- TACC** Target load acceleration – acceleration/deceleration reference computed by the reference generator at each slow loop sampling period. Measured in [acceleration units](#)
- APOS_LD** Actual load position. Measured in [position units](#). Alternate name: **APOS**
- ASPD_LD** Actual load speed – measured in [speed units](#)
- APOS_MT** Actual motor position. Measured in [motor position units](#).
- ASPD_MT** Actual motor speed. Measured in [motor speed units](#). Alternate name: **ASPD**

Instructions

SETPT value Set PT operation as specified by **value**:

*PT operation mode (a copy of **value** is saved in the MPL variable **PVTMODE**)*

BIT	VALUE	DESCRIPTION
15	0	Nothing
	1	Clear PVT buffer and reinitialize the buffer internal variables. This command is mandatory after changing the start address of the PT buffer
14	0	Enable the integrity counter error checking
	1	Disable the integrity counter error checking. When integrity error checking is disabled, the drive stops sending messages when PT buffer becomes full, low or empty.
13	0	No change to the internal integrity counter
	1	Change internal integrity counter with the value specified in bits 0 to 6
12	0	If PT mode is set under CPA (absolute positioning), the initial position is taken from MPL parameter PVTPOS0 (default = 0). The initial position is used to compute the distance to move up to the first PT point
	1	If PT mode is set under CPA (absolute positioning), the initial position is considered the current value of MPL variable TPOS . The initial position is used to compute the distance to move up to the first PT point
11..8	0..15	New parameter for buffer low signaling. When the number of entries in the PT buffer is equal or less than buffer low value, PVTSTS.14 is set to one.
7	0	No change in the buffer low parameter
	1	Change the buffer low parameter with the value specified in bits 8 to11
6..0	0..127	New integrity counter value

CPR PT mode is relative

CPA PT mode is absolute

MODE PT Set PT motion mode.

TUM1 Target Update Mode 1 (**TUM1**). Generates new trajectory starting from the actual values of position and speed reference (i.e. don't update the reference values with load/motor position and speed)

TUM0 Target Update Mode 0 (**TUM0**). Generates new trajectory starting from the actual values of load/motor position and speed (i.e. update the reference values with load/motor position and speed)

PTP Position, Time, Counter Defines a PT point, where:

Position – is the PT point position, measured in position units. In absolute mode, it is the position to reach. In relative mode, it is the position increment from the previous PT point. The position value is a 32-bit long integer.

Time – is the PT point time measured in time units. The time value is a 16-bit unsigned integer

Counter – is the PVT point integrity counter. It is a 7-bit unsigned integer with values between 0 and 127.

UPD Update motion parameters and start new motion mode

STOP Stop motion

Remarks:

- When a PT sequence of points is executed from a MPL program, the first **PTP** commands are processed one after the other, until the PT buffer fills up. At this point the MPL program stops until the PT buffer starts to empty. Therefore, the next **PTP** commands are processed in the cadence of the PT points execution. At the end of the sequence, the PT buffer starts to empty and next MPL instructions start to execute. This may lead to incorrect operation if for example a new motion mode is set while there are still points in the PT buffer waiting to be executed. In order to avoid this situation, it is **mandatory** to end the PVT sequence with an [event on motion complete](#) and wait until this event occurs.
- In order to activate the **TUM1** mode, execute the MPL instruction **TUM1** AFTER the **MODE PT** command and BEFORE the **UPD** command. When **MODE PT** is executed, it automatically sets **TUM0** mode. However, as the new motion mode becomes effective only after the **UPD** command, a **TUM1** command will overwrite the **TUM0** mode
- Under **TUM0** mode, at the **UPD** command **TPOS=APOS_LD** and **TSPD=ASPD_LD**. In open loop control of steppers, **TUM0** is ignored as there is no position and/or speed feedback
- In setup configurations where there is no transmission ratio between the motor and the load, it is supposed that these are directly connected. In these cases: **APOS_MT=APOS_LD** and **ASPD_MT=ASPD_LD**

Programming Example

```
// PT sequence. Position feedback: 500 lines incremental
// encoder (2000 counts/rev)
SETPVT 0xC000; //Clear PT buffer, disable counter check
           //Don't change counter & buffer low condition
MODE PT; // Set PT Mode
TUM1; //Start from actual value of position reference
CPR;
PTP 2000L, 100U, 0; //PT(1[rot], 0.1[s])
UPD; //Execute immediate
PTP 0L, 100U, 0; //PT(1[rot],0.2[s])
PTP -2000L, 100U, 0; //PT(0[rot],0.3[s])
!MC; WAIT!; //wait for completion
```

See also:

[PT – MPL Programming Details](#)

[MPL Description](#)

6.2.2.2.12. Mode PVT - Related MPL Instructions and Data

Parameters

MASTERID Contains the axis ID of the host/master where the drive/motor must send the PVT messages. It must be set before starting the PVT mode. The **MASTERID** value must be set as: host ID << 4 + 1, where host ID is a number between 1 and 255 representing the host ID. By default, after power-on the host ID is set equal with the drive address causing all the PVT messages to be sent via RS-232

PVTBUFBEGIN Specifies the start address of the PVT buffer

PVTBUFLN Specifies the PVT buffer length expressed in PVT points

PVTPOS0 Specifies for absolute mode, the initial position from which to start computing the distance to move up to the first PVT point. An alternate option is to consider **TPOS** as initial position. Selection between these 2 options is done at PVT initialization via MPL command **SETPVT**. The default value of **PVTPOS0** is 0.

PVTSENDOFF When set to 1, disables transmission of messages during PVT mode. By default is set to 0 and the transmission is enabled

Variables

PVTSTS Contains the PVT motion mode status.

PVTSTS bit description

BIT	VALUE	DESCRIPTION
15	0	PVT buffer is not empty
	1	PVT buffer is empty – there is no PVT point in the buffer and the execution of the current PVT segment is over. If PVTSENOFF = 0 (default), the drive/motor will send the PVTSTS each time this bit goes from 0 to 1
14	0	PVT buffer is not low
	1	PVT buffer is low – the number of PVT points from the buffer is equal or less than the low limit set using SETPVT . If PVTSENOFF = 0 (default), the drive will send the PVTSTS each time this bit goes from 0 to 1
13	0	PVT buffer is not full
	1	PVT buffer is full – the number of PVT points from the buffer is equal with the buffer dimension. If PVTSENOFF = 0 (default), the drive will send the PVTSTS each time this bit goes from 0 to 1
12	0	No integrity counter error
	1	Integrity counter error. If integrity counter error checking is enabled and PVTSENOFF = 0 (default), the drive will send the PVTSTS each time this bit goes from 0 to 1
11	0	The drive has kept the PVT motion mode after a PVT buffer empty condition, because the velocity of the last PVT point was 0
	1	The drive has performed a Quick stop, following a PVT buffer empty condition, because the velocity of the last PVT point was different from 0
10	0	Normal operation. Data received are PVT points
	1	A PT point was received while PVT mode is active. The PT point was discharged. If PVTSENOFF = 0 (default), the drive/motor will send the PVTSTS each time this bit goes from 0 to 1
9..7	0	Reserved
6..0	0..127	Current integrity counter value

PVTMODE PVT operation mode as was set with the **SETPVT** command

TPOS Target load position – position reference computed by the reference generator at each slow loop sampling period. Measured in [position units](#)

TSPD Target load speed – speed reference computed by the reference generator at each slow loop sampling period. Measured in [speed units](#)

TACC Target load acceleration – acceleration/deceleration reference computed by the reference generator at each slow loop sampling period. Measured in [acceleration units](#)

APOS_LD Actual load position. Measured in [position units](#). Alternate name: **APOS**

ASPD_LD Actual load speed – measured in [speed units](#)

APOS_MT Actual motor position. Measured in [motor position units](#).

ASPD_MT Actual motor speed. Measured in [motor speed units](#). Alternate name: **ASPD**

Instructions

SETPVT **value** Set PVT operation as specified by **value**:

*PVT operation mode (a copy of **value** is saved in the MPL variable **PVTMODE**)*

BIT	VALUE	DESCRIPTION
15	0	Nothing
	1	Clear PVT buffer and reinitialize the buffer internal variables. This command is mandatory after changing the start address of the PVT buffer
14	0	Enable the integrity counter error checking
	1	Disable the integrity counter error checking
13	0	No change to the internal integrity counter
	1	Change internal integrity counter with the value specified in bits 0 to 6
12	0	If PVT mode is set under CPA (absolute positioning), the initial position is taken from MPL parameter PVTPOS0 (default = 0). The initial position is used to compute the distance to move up to the first PVT point
	1	If PVT mode is set under CPA (absolute positioning), the initial position is considered the current value of MPL variable TPOS . The initial position is used to compute the distance to move up to the first PVT point
11..8	0..15	New parameter for buffer low signaling. When the number of entries in the PVT buffer is equal or less than buffer low value, PVTSTS.14 is set to one.
7	0	No change in the buffer low parameter
	1	Change the buffer low parameter with the value specified in bits 8 to11
6..0	0..127	New integrity counter value

CPR PVT mode is relative

CPA PVT mode is absolute

MODE PVT Set PVT motion mode.

TUM1 Target Update Mode 1 (**TUM1**). Generates new trajectory starting from the actual values of position and speed reference (i.e. don't update the reference values with load/motor position and speed)

TUM0 Target Update Mode 0 (**TUM0**). Generates new trajectory starting from the actual values of load/motor position and speed (i.e. update the reference values with load/motor position and speed)

PVTP Position, Velocity, Time, Counter Defines a PVT point, where:

Position – is the PVT point position, measured in position units. In absolute mode, it is the position to reach. In relative mode, it is the position increment from the previous PVT point. The position value is a signed long integer limited to 24 bits, i.e. in the range – 8388608 to + 8388607. Values outside this range are truncated causing unpredictable results.

Velocity – is the PVT point velocity, measured in speed units. The velocity is a fixed value like command speed CSPD and target speed TSPD

Time – is the PVT point time measured in **time units** The time value is a 9-bit unsigned integer having values between 1 and 511.

Counter – is the PVT point integrity counter. It is a 7-bit unsigned integer with values between 0 and 127.

UPD Update motion parameters and start new motion mode

STOP Stop the motion

Remarks:

- When a PVT sequence of points is executed from a MPL program, the first **PVTP** commands are processed one after the other, until the PVT buffer fills up. At this point the MPL program stops until the PVT buffer starts to empty. Therefore, the next **PVTP** commands are processed in the cadence of the PVT points execution. At the end of the sequence, the PVT buffer starts to empty and next MPL instructions start to execute. This may lead to incorrect operation if for example a new motion mode is set while there are still points in the PVT buffer waiting to be executed. In order to avoid this situation, it is mandatory to end the PVT sequence with an [event on motion complete](#) and wait until this event occurs.
- In order to activate the **TUM1** mode, execute the MPL instruction **TUM1** AFTER the **MODE PVT** command and BEFORE the **UPD** command. When **MODE PVT** is executed, it automatically sets **TUM0** mode. However, as the new motion mode becomes effective only after the **UPD** command, a **TUM1** command will overwrite the **TUM0** mode
- Under **TUM0** mode, at the **UPD** command **TPOS=APOS_LD** and **TSPD=ASPD_LD**. In open loop control of steppers, **TUM0** is ignored as there is no position and/or speed feedback
- In setup configurations where there is no transmission ratio between the motor and the load, it is supposed that these are directly connected. In these cases: **APOS_MT=APOS_LD** and **ASPD_MT=ASPD_LD**

Programming Example

```
// PVT sequence. Position feedback: 500 lines incremental
// encoder (2000 counts/rev)
MASTERID = 4081; // Set host address to 255 (255<<4+1)
SETPVT 0xC000; //Clear PVT buffer, disable counter check
//Don't change counter & buffer low condition
MODE PVT; // Set PVT Mode
TUM1;//Start from actual value of position reference
CPR; // Relative mode
PVTP 400L, 60, 10U, 0;//PVT(0.2[rot], 1800[rpm], 0.01[s])
UPD; //Execute immediate
PVTP 400L, 0, 10U, 0;//PVT(0.4[rot], 0[rpm], 0.02[s])
!MC; WAIT!; //wait for completion
```

See also:

[PVT – MPL Programming Details](#)

[MPL Description](#)

6.2.2.2.13. External - MPL Programming Details

In the external modes, you program the [drives/motors](#) to work with an external reference provided by another device. There are 3 types of external references:

- Analogue – read by the drive/motor via a dedicated analogue input (10-bit resolution)
- Digital – computed by the drive/motor from:
 - Pulse & direction signals
 - Quadrature signals like A, B signals of an incremental encoder
 - Online – received online via a communication channel from a host and saved in a dedicated MPL variable

When the reference is analogue or online, you can set a:

- Position external mode, where the motor is controlled in position and the external reference is interpreted as a position reference
- Speed external mode, where the motor is controlled in speed and the external reference is interpreted as a speed reference
- Torque external mode, where the motor is controlled in torque and the external reference is interpreted as a current reference.
- Voltage external mode, where the motor is controlled in voltage and the external reference is interpreted as a voltage reference.

When the external reference is digital, the option for the input signals: pulse & direction or quadrature encoder is established during the drive/motor setup. The drive/motor performs only position control having as goal to follow the position reference computed from the input signals with a preset gear ratio. In this case, the drive/motor actually works in [electronic gearing mode](#), where you can find further details.

In position external mode with analogue or online reference, you can limit the maximum speed at sudden changes of the position reference and thus to reduce the mechanical shocks. This feature is activated by setting **UPGRADE.2=1** and the maximum speed value in **CSPD**.

In speed external mode with analogue or online reference, you can limit the maximum acceleration at sudden changes of the speed reference and thus to get a smoother transition. This feature is activated by setting **UPGRADE.2=1** and the maximum acceleration value in **CACC**.

In torque or voltage external mode with analogue reference, you can choose how often to read the analogue input: at each slow loop sampling period or at each fast loop sampling period.

When using the analogue reference, during the setup phase, you specify the reference values corresponding to the upper and lower limits of the analogue input. Depending on the control mode selected, these values may be position or speed or torque or voltage references. You may also select a dead-band symmetrical interval and its center point inside the analogue input range. While the analogue signal is inside the dead-band interval, the output reference is kept constant and equal with value corresponding to the dead-band center point. This option is especially useful when you need to set a precise reference, which doesn't change in the presence of some noise on the analogue input signal. If dead-band width is set to zero, the dead-band is disabled.

Remark: Setup tools like PROconfig, automatically compute the value of the MPL parameters needed to convert the analogue input range into the desired reference range.

See also:

[External – Related MPL Instructions and Data](#)

[MPL Description](#)

6.2.2.2.14. External - MPL Instructions and Data

Parameters

CADIN Half of the reference range expressed in internal units, divided by 2SFTADIN. The division result should lead to a value less than 32767. Depending on control mode selected, the reference range can be a:

- Position range expressed in [position units](#)
- Speed range expressed in [speed units](#)
- Torque range expressed in [current units](#)
- Voltage range expressed in [voltage command units](#)

SFTADIN The smallest power of 2 for which:

$$(\text{Half of the reference range in internal units}) / 2\text{SFTADIN} < 32767$$

AD5OFF Reference value expressed in internal units, corresponding to the lower limit of the analogue input. Depending on control mode selected, the reference value can be a:

- Position value expressed in [position units](#)
- Speed value expressed in [speed units](#)
- Torque value expressed in [current units](#)
- Voltage value expressed in [voltage command units](#)

FILTER1 Cutoff frequency for the low-pass filter on analogue input, computed with:

$$\text{FILTER1} = 32767 * (1 - \exp(-\text{fc} * \text{T})),$$

where **fc** is the cutoff frequency in radians/s

T is the slow loop sampling period in seconds.

*Remark: For the external torque mode with analogue input read in fast loop, **T** is the fast loop sampling period in seconds.*

LEVEL_AD5 Dead-band point in internal units computed with:

$$\text{LEVEL_AD5} = (\text{DB_Point} - \text{InputLow}) * 65472 / \text{InputRange}$$

where **DB_Point** – is the dead band point expressed in V

InputLowLimit – is the low limit of the drive/motor analogue input expressed in V

InputRange – is the drive/motor analogue input range expressed in V.

E_LEVEL_AD5 Dead-band range in internal units computed with formula:

$$\text{E_LEVEL_AD5} = \text{DB_Range} * 65472 / \text{InputRange},$$

where **DB_Range** – is the desired dead-band range expressed in V

InputRange – is the drive/motor analogue input range expressed in V.

UPGRADE MPL register. When **UPGRADE.2=1**, a speed limitation may be set in position external mode and an acceleration limitation in speed external mode. When **UPGRADE.2=0**, speed or acceleration limitation is disabled

CSPD Maximum speed in position external when **UPGRADE.2=1**

CACC Maximum acceleration in speed external when **UPGRADE.2=1**

Variables

AD5 16-bit unsigned integer value representing the value read from the analogue input. The output of the 10-bit A/D converter is set in the 10 MSB (most significant bits) of the AD5

EREFP MPL variable where an external device writes the position reference in external mode on-line. Measured in [position units](#)

EREFS MPL variable where an external device writes the speed reference in external mode on-line. Measured in [speed units](#)

EREFT MPL variable where an external device writes the torque reference in external mode on-line. Measured in [current units](#)

EREFV MPL variable where an external device writes the voltage reference in external mode on-line. Measured in [voltage command units](#)

TPOS Target load position – position reference computed by the reference generator at each slow loop sampling period, when position external mode is performed. **TPOS** is set function of the analogue input value, with analogue reference or with the **EREFP** value with online reference. Measured in [position units](#)

TSPD Target load speed – speed reference computed by the reference generator at each slow loop sampling period, when position or speed external mode is performed. In speed control, **TSPD** is set function of the analogue input value, with analogue reference or with the **EREFS** value with online reference. Measured in [speed units](#) In position control, **TSPD** is computed as the position variation over a slow loop sampling period. Measured in [speed units](#)

TACC Target load acceleration – acceleration/deceleration reference computed by the reference generator at each slow loop sampling period, when position or speed external mode is performed. Measured in [acceleration units](#)

IQREF Current reference – updated at each fast or slow loop function of the analogue input value or set with **EREFT** value, when torque external mode is performed. Measured in [current units](#)

UQREF Voltage reference – updated at each fast or slow loop function of the analogue input value or set with **EREFV** value, when voltage external mode is performed. Measured in [voltage command units](#)

APOS_LD Actual load position. Measured in [position units](#). Alternate name: **APOS**

ASPD_LD Actual load speed – measured in [speed units](#)

APOS_MT Actual motor position. Measured in [motor position units](#).

ASPD_MT Actual motor speed. Measured in [motor speed units](#). Alternate name: **ASPD**

IQ Motor current – measured in [current units](#)

Instructions

MODE PE Set position external mode

<u>MODE GS</u>	Set position external mode with digital reference
<u>MODE SE</u>	Set speed external mode
<u>MODE TES</u>	Set torque external mode with reference read in slow loop
<u>MODE TEF</u>	Set torque external mode with reference read in fast loop
<u>MODE VES</u>	Set voltage external mode with reference read in slow loop
<u>EXTREF 0</u>	Set external reference type on-line
<u>EXTREF 1</u>	Set external reference type analogue
<u>EXTREF 2</u>	Set external reference type digital
<u>UPD</u>	Update motion parameters and start new motion mode
<u>STOP</u>	Stop motion

Remarks:

- *In the absence of an external device, EREFP, EREFS, EREFT, EREFV may also be used as MPL parameters through which you can set a position, speed, torque or voltage reference in the external mode online. This is a simple way to impose step references*
- *The MPL variables EREFP, EREFS, EREFT, EREFV are alternate ways to address the MPL variable EREF in which the external devices must place the reference. The new mnemonics have been added to clearly differentiate how EREF is interpreted function of control mode selected:*
 - *Position control: EREFP = EREF. EREFP is a 32-bit long integer*
 - *Speed control: EREFS=EREF. EREF is a 32-bit fixed*
 - *Torque control: EREFT = EREF(H). EREFT is a 16-bit integer*
 - *Voltage control: EREFV = EREF(H). EREFV is a 16-bit integer*
- **CSPD** and **CACC** must be positive
- *The sum between **CSPD** and **CACC** values must be maximum 32767.99998 (0x7FFF.FFFF) i.e. the maximum value for fixed number*
- *In setup configurations where there is no transmission ratio between the motor and the load, it is supposed that these are directly connected. In these cases: **APOS_MT=APOS_LD** and **ASPD_MT=ASPD_LD***

Programming Example 1

```
// External mode. Read position reference from the analogue input
// Position feedback: 500 lines incremental
// encoder (2000 counts/rev)
MODE PE; //External position
CSPD = 100; // Limit = 3000[rpm]
SRB UPGRADE, 0xFFFF, 0x0004; //UPGRADE.2 = 1
UPD; //execute immediate
```

Programming Example 2

```
// External mode online. Read speed reference from variable EREFS
// Position feedback: 500 lines incremental
// encoder (2000 counts/rev)
EREFs = 33.3333; // EREFs initial = 1000[rpm]
EXTREF 0;
MODE SE; //External speed
CACC = 0.3183; // Limit = 1000[rad/s^2]
SRB UPGRADE, 0xFFFF, 0x0004; //UPGRADE.2 = 1
UPD; //execute immediate
```

See also:

[External – MPL Programming Details](#)

[MPL Description](#)

6.2.3. Electronic Gearing - MPL Programming Details

In the electronic gearing a drive/motor may operate as **master** or as **slave**.

When set as **master**, the drive/motor sends its position via a multi-axis communication channel, like the CANbus. When set as **slave**, the drive/motor follows the master position with a programmable gear ratio.

Master operation

The master operation can be enabled with the MPL command **SGM** followed by an **UPD** (update) and can be disabled by the MPL command **RGM** followed by an **UPD**. In both cases, these operations have no effect on the motion executed by the master.

Once at each slow loop sampling time interval, the master sends either its load position **APOS** (if **OSR.15** = 0) or its position reference **TPOS** (if **OSR.15** = 1) to the axis or the group of axes specified in the MPL parameter **SLAVEID**. The **SLAVEID** contains either the axis ID of one slave or the value of a group ID+256 i.e. the group of slaves to which the master should send its data.

***Remark:** The group ID is an 8-bit unsigned value. Each bit set to 1 represents a group: bit 0 – group 1, bit 1 – group 2, etc. In total there are 8 groups. For example, if the master sends its position to group 3, the group ID = 4 (00000100b) and the SLAVEID is 4+256 = 260.*

The master operation can be synchronized with that of the slaves. The synchronization process is performed in two steps. First, the master sends a synchronization message to all axes, including to itself. When this message is received, all the axes read their own internal time. Next, the master sends its internal time to all the slaves, which compare it with their own internal time. If there are differences, the slaves correct slightly their sampling periods in order to keep them synchronized with those of the master. As effect, when synchronization procedure is active, the execution of the control loops on the slaves is synchronized with those of the master within a 10µs time interval. Due to this powerful feature, drifts between master and slave axes are eliminated. The synchronization procedure is activated with the MPL command **SETSYNC value** where **value** represents the time interval in internal units between the synchronization messages sent by the master. Recommended value is 20ms.

If the master is going to be activated with slaves already set in electronic gearing, an initialization is necessary before enabling the master operation: the master must set the MPL parameter MPOS0 on all the slaves with a non-zero value, for example 1.

Slave operation

The slaves can get the master position in two ways:

1. Via a communication channel, from a drive/motor set as master
2. Via an external digital reference of type pulse & direction (if **ACR.2**=1) or quadrature encoder (if **ACR.2** = 0). Both options have dedicated inputs. The pulse & direction signals are usually provided by an indexer and must be connected to the pulse & direction inputs of the drive/motor. The quadrature encoder signals are usually provided by an encoder on the master and must be connected to the 2nd encoder inputs.

You can activate the first option with the MPL command: **EXTREF 0** and the second option with the MPL command **EXTREF 2**. Both become effective at the next **UPD** command.

In slave mode the drive/motor performs a position control. At each slow loop sampling period, the slave computes the master position increment and multiplies it with its programmed gear ratio. The result is the slave position reference increment, which added to the previous slave position reference gives the new slave position reference.

***Remark:** The slave executes a relative move, which starts from its actual position*

The gear ratio is specified via 3 MPL parameters: **GEAR**, **GEARSLAVE** and **GEARMASTER**. **GEARSLAVE** and **GEARMASTER** represent the numerator and denominator of the Slave / Master ratio. **GEARSLAVE** is a signed integer, while **GEARMASTER** is an unsigned integer. **GEARSLAVE** sign indicates the direction of movement: positive – same as the master, negative – reversed to the master. **GEAR** is a fixed value containing the result of the ratio i.e. the result of the division **GEARSLAVE / GEARMASTER**. **GEAR** is used to compute the slave reference increment, while **GEARSLAVE** and **GEARMASTER** are used by an automatic compensation procedure which eliminates the round off errors which occur when the gear ratio is an irrational number like: 1/3 (Slave = 1, Master = 3).

The MPL parameter **MASTERRES** provides the master resolution which is needed to compute correctly the master position and speed (i.e. the position increment). **MASTERRES** is a 32-bit long integer value, expressed in the master position units. If master position is not cyclic (i.e. the resolution is equal with the whole 32-bit range of position), set master resolution to 0x80000001.

When master position is provided via the external digital interface, the slave computes the master position by counting the pulse & direction or quadrature encoder signals. The initial value of the master position is set by default to 0. It may be changed to a different value by writing the desired value in the MPL variable **APOS2**.

MPL commands **REG_ON/REG_OFF** enable/disable the superposition of the electronic gearing mode with a second motion mode. When this *superposed mode* activated, the position reference is computed as the sum of the position references for each of the 2 superposed motions.

You may enable the *superposed mode* at any moment, independently of the activation/deactivation of the electronic gearing slave. If the *superposed mode* is activated during an electronic gearing motion, any subsequent motion mode change is treated as a second move to be superposed over the basic electronic gearing move, instead of replacing it. If the *superposed mode* is activated during another motion mode, a second electronic gearing mode will start using the motion parameters previously set. This move is superposed over the first one. After the first move ends, any other subsequent motion will be added to the electronic gearing.

When you disable the *superposed mode*, the electronic gearing slave move is stopped and the drive/motor executes only the other motion. If you want to remain in the electronic gearing slave mode, set first the electronic gearing slave move and then disable the *superposed mode*.

You can smooth the slave coupling with the master, by limiting the maximum acceleration on the slave. This is particularly useful when the slave is must couple with a master running at high speed. The feature is activated by setting **UPGRADE.2=1** and the maximum acceleration value in **CACC**.

Remark: When slave coupling with the master is complete **SRH.12 = 1**. The same bit is reset to zero if the slave is decoupled from the master. The bit has no significance in other motion modes.

See also:

[Electronic Gearing – Related MPL Instructions and Data](#)

[MPL Description](#)

6.2.3.1.1. Electronic Gearing - Related MPL Instructions and Data

Parameters

<u>CACC</u>	Maximum acceleration in slave mode when UPGRADE.2=1
<u>SLAVEID</u>	The axis or group ID to which the master sends its position. When group ID is used, the SLAVEID is set with group ID value + 256.
<u>GEAR</u>	Slave(s) gear ratio value. Negative values means opposite direction compared with the master
<u>GEARMASTER</u>	Denominator of gear ratio
<u>GEARSLAVE</u>	Numerator of gear ratio. Negative values means opposite direction compared with the master
<u>MASTERRES</u>	Master resolution used by slave(s) Measured in <u>master position units</u>
<u>MPOS0</u>	Initialization parameter. Must be set by the master with a non-zero value before enabling the master mode, if the slaves are already set in electronic gearing.
<u>OSR</u>	MPL register. When OSR.15=1 , the master sends the position reference. When OSR.15=0 , the master sends the actual load position
<u>UPGRADE</u>	MPL register. When UPGRADE.2=1 , an acceleration limitation may be set on slave. When UPGRADE.2=0 , the acceleration limitation is disabled
<u>ACR</u>	Auxiliary Control Register – includes several MPL Programming options. When ACR.2 = 0 , the external reference is quadrature encoder. When ACR.2 = 1 , the external reference is pulse & direction

Variables

<u>MREF</u>	Master position received or computed by the slave(s). Measured in <u>master position units</u>
<u>MSPD</u>	Master speed computed by the slaves. Measured in <u>master speed units</u>
<u>APOS2</u>	Master position computed by the slaves from pulse & direction or quadrature encoder inputs. At power-on it is set to 0. May be set to a different value, before starting the master. Measured in <u>master position units</u>
<u>TPOS</u>	Target position – position reference computed by the reference generator at each slow loop sampling period. Measured in <u>position units</u>
<u>TSPD</u>	Target speed – speed reference computed by the reference generator at each slow loop sampling period. Measured in <u>speed units</u>
<u>TACC</u>	Target acceleration – acceleration/deceleration reference computed by the reference generator at each slow loop sampling period. Measured in <u>acceleration units</u>
<u>APOS_LD</u>	Actual load position. Measured in <u>position units</u> . Alternate name: <u>APOS</u>
<u>ASPD_LD</u>	Actual load speed – measured in <u>speed units</u>
<u>APOS_MT</u>	Actual motor position. Measured in <u>motor position units</u> .
<u>ASPD_MT</u>	Actual motor speed. Measured in <u>motor speed units</u> . Alternate name: <u>ASPD</u>

Instructions

<u>EXTREF 0</u>	Get master position via a communication channel
<u>EXTREF 2</u>	Compute master position from pulse & direction or quadrature encoder signals
<u>MODE GS</u>	Set electronic gear slave mode
<u>SGM</u>	Set electronic gear master mode
<u>RGM</u>	Reset electronic gear master mode
<u>REG_ON</u>	Enable superposed mode
<u>REG_OFF</u>	Disable superposed mode
<u>SETSINC</u> value	Send synchronization messages at the time interval indicated by the 16-bit value . Measured in <u>time units</u>
<u>TUM1</u>	Target Update Mode 1 (TUM1). Generates new trajectory starting from the actual values of position and speed reference (i.e. don't update the reference values with load/motor position and speed)
<u>TUM0</u>	Target Update Mode 0 (TUM0). Generates new trajectory starting from the actual values of load/motor position and speed (i.e. updates the reference values with load/motor position and speed)
<u>UPD</u>	Update motion parameters and start new motion mode
<u>STOP</u>	Stop the motion
<u>SRB</u>	Set/reset bits from a MPL data

Remarks:

- Do not change **GEAR**, **GEARSLAVE** and **GEARMASTER** during slave operation
- **CACC** must be positive
- In order to activate the **TUM1** mode, execute the MPL instruction **TUM1** AFTER the **MODE GS** command and BEFORE the **UPD** command. When **MODE GS** is executed, it automatically sets **TUM0** mode. However, as the new motion mode becomes effective only after the **UPD** command, a **TUM1** command will overwrite the **TUM0** mode
- Under **TUM0** mode, at the **UPD** command **TPOS=APOS_LD** and **TSPD=ASPD_LD**. In open loop control of steppers, **TUM0** is ignored as there is no position and/or speed feedback
- In setup configurations where there is no transmission ratio between the motor and the load, it is supposed that these are directly connected. In these cases: **APOS_MT=APOS_LD** and **ASPD_MT=ASPD_LD**

Programming Example

```
//Electronic gearing. Master position is received via //communication
channel inputs.

//Master resolution: 2000 counts/rev

// On slave axis (Axis ID = 1):
GEAR = 0.3333; // gear ratio
GEARMASTER = 3; //gear ratio denominator
GEARSLAVE = 1; //gear ratio numerator
EXTREF 0; // master position got via communication channel
MASTERRES = 2000; // master resolution
REG_ON; //Enable superposition
MODE GS; //Set as slave, position mode
TUM1; //Set Target Update Mode 1
SRB UPGRADE, 0xFFFF, 0x0004; //UPGRADE.2 = 1
CACC = 0.9549; //Limit maximum acceleration at 3000[rad/s^2]
UPD; //execute immediate

// On master axis:
SLAVEID = 1;
SGM; //Enable Master in Electronic Gearing mode
SRB OSR, 0xFFFF, 0x8000; // OSR.15=1 -> Send Position Reference
[1]MPOS0 = TPOS;
UPD; //execute immediate
SETSYNC 20; //Send synchronization messages every 20[ms]
```

See also:

[Electronic Gearing – MPL Programming Details](#)

[MPL Description](#)

6.2.3.1.2. Electronic Camming - MPL Programming Details

In the electronic camming a drive/motor may operate as **master** or as **slave**.

When set as **master**, the drive/motor sends its position via a multi-axis communication channel, like the CAN bus. When set as **slave**, the drive/motor executes a cam profile function of the master position. The cam profile is defined by a cam table – a set of (X, Y) points, where X is cam table input i.e. the master position and Y is the cam table output i.e. the corresponding slave position. Between the points the drive/motor performs a linear interpolation.

Master operation

The master operation can be enabled with the MPL command **SGM** followed by an **UPD** (update) and can be disabled by the MPL command **RGM** followed by an **UPD**. In both cases, these operations have no effect on the motion executed by the master.

Once at each slow loop sampling time interval, the master sends either its load position **APOS** (if **OSR.15** = 0) or its position reference **TPOS** (if **OSR.15** = 1) to the axis or the group of axes specified in the MPL parameter **SLAVEID**. The **SLAVEID** contains the axis ID of one slave or the value of a group ID+256 i.e. the group of slaves to which the master should send its data.

***Remark:** The group ID is an 8-bit unsigned value. Each bit set to 1 represents a group: bit 0 – group 1, bit 1 – group 2, etc. In total there are 8 groups. For example, if the master sends its position to group 3, the group ID = 4 (00000100b) and the SLAVEID is 4+256 = 260.*

The master operation can be synchronized with that of the slaves. The synchronization process is performed in two steps. First, the master sends a synchronization message to all axes, including to itself. When this message is received, all the axes read their own internal time. Next, the master sends its internal time to all the slaves, which compare it with their own internal time. If there are differences, the slaves correct slightly their sampling periods in order to keep them synchronized with those of the master. As effect, when synchronization procedure is active, the execution of the control loops on the slaves is synchronized with those of the master within a 10µs time interval. Due to this powerful feature, drifts between master and slave axes are eliminated. The synchronization procedure is activated with the MPL command **SETSYNC value** where **value** represents the time interval in internal units between the synchronization messages sent by the master. Recommended value is 20ms.

Slave operation

The slaves can get the master position in two ways:

1. Via a communication channel, from a drive/motor set as master
2. Via an external digital reference of type pulse & direction (if **ACR.2**=1) or quadrature encoder (if **ACR.2** = 0). Both options have dedicated inputs. The pulse & direction signals are usually provided by an indexer and must be connected to the pulse & direction inputs of the drive/motor. The quadrature encoder signals are usually provided by an encoder on the master and must be connected to the 2nd encoder inputs.

You can activate the first option with the MPL command: **EXTREF 0** and the second option with the MPL command **EXTREF 2**. Both become effective at the next **UPD** command.

The MPL parameter **MASTERRES** provides the master resolution which is needed to compute correctly the master position and speed (i.e. the position increment). **MASTERRES** is a 32-bit long integer value, expressed in the master position units. If master position is not cyclic (i.e. the resolution is equal with the whole 32-bit range of position), set master resolution to 0x80000001.

When master position is provided via the external digital interface, the slave computes the master position by counting the pulse & direction or quadrature encoder signals. The initial value of the master

position is set by default to 0. It may be changed to a different value by writing the desired value in the MPL variable **APOS2**.

Through the MPL parameter **CAMOFF** you can shift the cam profile versus the master position, by setting an offset for each slave. The cam table input is computed as the master position minus the cam offset. For example, if a cam table is defined between angles 100 to 250 degrees, a cam offset of 50 degrees will make the cam table to execute between master angles 150 and 300 degrees.

In slave mode the drive/motor performs a position control. Based on the master position **X**, it calculates the cam table output **Y = f(X)**. It is not mandatory to define the cam table for 360 degrees of the master. You may also define shorter cam tables, with a start angle **Xmin** > 0 and an end angle **Xmax** < 360 degrees. In this case, the cam table output remains unchanged outside the active area of the cam, being computed as follows:

- **Y = Ymin = f(Xmin), if 0 < X < Xmin**
- **Y = f(X), if Xmin ≤ X ≤ Xmax**
- **Y = Ymax = f(Xmax), if Xmax < X < 360**

The electronic camming can be: **relative** (if **ACR.12 = 0**) or **absolute** (if **ACR.12 = 1**).

In the **relative** mode, the output of the cam table is added to the slave actual position. At each slow loop sampling period the slave computes a position increment **dY = Y – Yold**. This is the difference between the actual cam table output **Y** and the previous one **Yold**. The position increment **dY** is added to the old target position to get a new target position: **TPOS = TPOS + dY**. The slave detects when the master position rolls over, from 360 degrees to 0 or vice-versa and automatically compensates in **dY** the difference between **Ymax** and **Ymin**. Therefore, in relative mode, you can continuously run the master in one direction and the slaves will execute the cam profile once at each 360 degrees with a glitch free transition when the cam profile is restarted.

When electronic camming is activated in relative mode, the slave initializes **Yold** with the first cam output computed: **Yold = Y = f(X)**. The slave will keep its position until the master starts to move and then it will execute the remaining part of the cam. For example if the master moves from **X** to **Xmax**, the slave moves with **Ymax – Y**.

In the **absolute** mode, the output of the cam table **Y** is the target position to reach: **TPOS = Y**.

Remark: *The absolute mode must be used with great care because it may generate abrupt variations on the slave target position if:*

- *Slave position is different from **Y** at entry in the camming model*
- *Master rolls over and **Ymax ≠ Ymin***

In the absolute mode, you can introduce a maximum speed limit to protect against accidental sudden changes of the positions to reach. The feature is activated by setting **UPGRADE.2=1** and the maximum speed value in **CSPD**.

Remark: *When the slave can't reach the target position corresponding to the cam profile due to the speed limitation, **SRH.14 = 1**. The same bit is reset to zero when the slave returns to normal operation following the cam profile with a speed below the maximum limit. The bit has no significance in other motion modes.*

One way to avoid abrupt variations at activation of absolute mode is to move the slave(s) in the position corresponding to the master actual value, before enabling the camming slave mode. This approach requires finding the cam table output before entering in the camming mode. You can get this information in the following way:

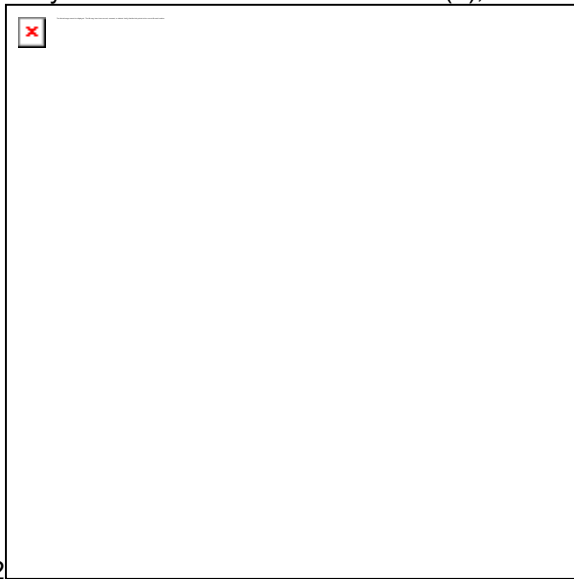
1. Set the slave(s) in trapezoidal position profile mode, for example to keep its actual position
2. Set MPL parameter **GEAR** (also used as gear ratio in electronic gearing) at 0.
3. Introduce an wait of 1ms (more exactly one slow-loop sampling period)
4. Read the cam table output for the actual master position from MPL variable **EREF**

Remark: Before executing point 2, make sure that the cam table is present in the RAM memory and **CAMSTART** is initialized accordingly (see below for details).

The cam tables are arrays of X, Y points, where X is the cam input i.e. the master position and Y is the cam output i.e. the slave position. The X points are expressed in the master internal position units, while the Y points are expressed in the slave internal [position units](#). Both X and Y points 32-bit long integer values. The X points must be positive (including 0) and equally spaced at: 1, 2, 4, 8, 16, 32, 64 or 128 i.e. having the interpolation step a power of 2 between 0 and 7. The maximum number of points for one cam table is 8192.

As cam table X points are equally spaced, they are completely defined by two data: the **Master start value** or the first X point and the **Interpolation step** providing the distance between the X points. This offers the possibility to minimize the cam size, which is saved in the drive/motor in the following format:

- 1st word (1 word = 16-bit data):
 - **Bits 15-13** – the power of 2 of the interpolation step. For example, if these bits have the binary value 010 (2), the interpolation step is



2 = 4, hence the master X values are spaced from 4 to 4: 0, 4, 8, 12, etc.

- **Bits 12-0** – the length -1 of the table. The length represents the number of points
- 2nd and 3rd words: the **Master start value** (long), expressed in **master position units**. 2nd word contains the low part, 3rd word the high part
- 4th and 5th words: Reserved. Must be set to 0
- Next pairs of 2 words: the slave Y positions (long), expressed in [position units](#). The 1st word from the pair contains the low part and the 2nd word from the pair the high part
- Last word: the cam table checksum, representing the sum modulo 65536 of all the cam table data except the checksum word itself

Before enabling electronic camming slave mode, the cam table must be downloaded into the drive/motor RAM memory and the MPL variable **CAMSTART** must be set with the value of the cam start address. It is possible to download more than one cam table in the drive/motor RAM memory and through **CAMSTART** to select which one to use at one moment.

Typically, the cam tables are first downloaded into the EEPROM memory of the drive, together with the rest of the MPL program. Then using the MPL command (included in the MPL program):

```
INITCAM LoadAddress, RunAddress
```

the cam tables are copied from the EEPROM memory into the drive/motor RAM memory. The `LoadAddress` is the EEPROM memory address where the cam table was loaded and `RunAddress` is the RAM memory address where to copy the cam table. After the execution of this command the MPL variable **CAMSTART** takes the value of the `RunAddress`.

Remarks:

- *Motion programming tool MotionPRO Developer automatically computes the start addresses in RAM and EEPROM of the selected cam tables and for each cam generates an **INITCAM** command. The **INITCAM** commands are included in the MPL application before ENDINIT. Therefore when this command is executed, all the selected cams are already copied from the EEPROM into the RAM.*
- *During electronic camming slave mode, only one cam table can be active at time*

You can compress/extend the cam table input. Specify through MPL parameter **CAMX**, an input correction factor by which the cam table input is multiplied. For example, an input correction factor of 2, combined with a cam offset of 180 degrees, will make possible to execute a cam table defined for 360 degrees of the master in the last 180 degrees.

You can also compress/extend the cam table output. Specify through the MPL parameter **CAMY**, an output correction factor by which the cam table output is multiplied. This feature addresses the applications where the slaves must execute different position commands at each master cycle, all having the same profile defined through a cam table. In this case, the drive/motor is programmed with a unique normalized cam profile and the cam table output is multiplied with the relative position command updated at each master cycle.

See also:

[Electronic Camming – Related MPL Instructions and Data](#)

[MPL Description](#)

6.2.3.1.3. Electronic Camming - Related MPL Instructions and Data

Parameters

- CSPD** Maximum speed in slave mode when **UPGRADE.2=1**
- CAMOFF** Cam offset. The input in the cam table before applying the scaling **MPOS0** is computed by subtracting cam offset from the master position. Measured in **master position units**
- CAMSTART** Pointer to cam table start address in RAM memory. When several cam tables are loaded in RAM, **CAMSTART** indicates which one is used. You can switch between cam tables by setting **CAMSTART** to the start address of another cam table. **CAMSTART** is automatically set by the INITCAM command, which copies the cam table from the EEPROM to the RAM memory
- CAMX** Cam input correction factor. Cam input **X** (MPL variable **CAMINPUT**) is:

$$X = \text{CAMINPUT} = \text{MPOS0} * \text{CAMX}$$

where **MPOS0 = MREF - CAMOFF**

- CAMY** Cam output correction factor. Cam table output **Y** is:

$$Y = f(X) * \text{CAMY}$$

- MASTERRES** Master resolution used by slave(s) (long) Measured in **master position units**.
- SLAVEID** The axis or group ID to which the master sends its position. When group ID is used, the **SLAVEID** is set with group ID value + 256.
- OSR** MPL register. When **OSR.15=1**, the master sends the position reference. When **OSR.15=0**, the master sends the actual load position
- UPGRADE** MPL register. When **UPGRADE.2=1**, a speed limitation may be set on slave. When **UPGRADE.2=0**, the speed limitation is disabled
- ACR** Auxiliary Control Register – includes several MPL Programming options. When **ACR.12 = 0**, the camming is relative. When **ACR.12 = 1**, the camming is absolute. When **ACR.2 = 0**, the external reference is quadrature encoder. When **ACR.2 = 1**, the external reference is pulse & direction

Variables

- MREF** Master position received or computed by the slave(s). Measured in [master position units](#)
- MSPD** Master speed computed by the slaves. Measured in [master speed units](#)
- MPOS0** Master position on the slave(s) after subtracting cam offset **CAMOFF**. Measured in [master position units](#)
- CAMINPUT** Cam table input
- APOS2** Master position computed by the slaves from pulse & direction or quadrature encoder inputs. At power-on it is set to 0. May be set to a different value, before starting the master. Measured in [master position units](#)
- TPOS** Target position – position reference computed by the reference generator at each slow loop sampling period. Measured in [position units](#)

<u>TSPD</u>	Target speed – speed reference computed by the reference generator at each slow loop sampling period. Measured in speed units
<u>TACC</u>	Target acceleration – acceleration/deceleration reference computed by the reference generator at each slow loop sampling period. Measured in acceleration units
<u>APOS_LD</u>	Actual load position. Measured in position units . Alternate name: <u>APOS</u>
<u>ASPD_LD</u>	Actual load speed – measured in speed units
<u>APOS_MT</u>	Actual motor position. Measured in motor position units .
<u>ASPD_MT</u>	Actual motor speed. Measured in motor speed units . Alternate name: <u>ASPD</u>

Instructions

<u>EXTREF 0</u>	Get master position via a communication channel
<u>EXTREF 2</u>	Compute master position from pulse & direction or quadrature encoder signals
<u>MODE CS</u>	Set electronic camming slave mode
<u>SGM</u>	Set electronic gearing/camming master mode
<u>RGM</u>	Reset electronic gearing/camming master mode
<u>SETSYNC value</u>	Send synchronization messages at the time interval indicated by the 16-bit value . Measured in time units
<u>INITCAM LoadAddress, RunAddress</u>	Copy a cam table from EEPROM starting with LoadAddress to RAM starting with RunAddress . Both values are unsigned integers
<u>TUM1</u>	Target Update Mode 1 (TUM1). Generates new trajectory starting from the actual values of position and speed reference (i.e. don't update the reference values with load/motor position and speed)
<u>TUM0</u>	Target Update Mode 0 (TUM0). Generates new trajectory starting from the actual values of load/motor position and speed (i.e. updates the reference values with load/motor position and speed)
<u>UPD</u>	Update motion parameters and start new motion mode
<u>STOP</u>	Stop the motion
<u>SRB</u>	Set/reset bits from a MPL data

Remarks:

- **CSPD** must be positive
- In order to activate the **TUM1** mode, execute the MPL instruction **TUM1** AFTER the **MODE CS** command and BEFORE the **UPD** command. When **MODE CS** is executed, it automatically sets **TUM0** mode. However, as the new motion mode becomes effective only after the **UPD** command, a **TUM1** command will overwrite the **TUM0** mode
- Under **TUM0** mode, at the **UPD** command **TPOS=APOS_LD** and **TSPD=ASPD_LD**. In open loop control of steppers, **TUM0** is ignored as there is no position and/or speed feedback
- In setup configurations where there is no transmission ratio between the motor and the load, it is supposed that these are directly connected. In these cases: **APOS_MT=APOS_LD** and **ASPD_MT=ASPD_LD**

Programming Example

```
// Electronic camming slave. Master position is read from 2nd
// encoder inputs. Master resolution : 2000 counts/rev

CAMSTART = 0xF000; //Initialize CAM table start address
EXTREF 2; // master position read from P&D or 2nd encoder
CAMOFF = 200; //Cam offset from master
CAMX = 0.5; //Cam input correction factor
CAMY = 1.5; //Cam output correction factor
MASTERRES = 2000; // master resolution
MODE CS; //Set electronic camming slave mode
TUM1; //Set Target Update Mode 1
SRB ACR, 0xEFFF, 0x0000; //Camming mode: Relative
UPD; //execute immediate
```

See also:

[Electronic Camming – MPL Programming Details](#)

[MPL Description](#)

6.2.3.1.4. Homing and Function Calls

ElectroCraft Motion Controller is able to start the execution of homing routines and MPL functions stored in the slaves' non-volatile memory. A maximum of 10 homing/functions can be called access from Motion Controller

Once the homing/function execution starts the Motion Controller application can be halted by using an event on function complete. The Motion complete resumes the application execution when the event occurs or it time outs.

See also:

[MPL Description](#)

6.2.3.1.5. Homing - MPL Programming Details

The *homing* is a sequence of motions, usually executed after power-on, through which the load is positioned into a well-defined point – the home position. Typically, the home position is the starting point for normal operation.

The search for the home position can be done in numerous ways. In order to offer maximum flexibility, the MPL does not impose the homing procedures but lets you define your own, according with your application needs.

Basically a homing procedure is a MPL function and by calling it you start executing the homing procedure. The call must be done using the MPL command **CALLS** – a cancelable call. This command offers the possibility to abort at any moment the homing sequence execution (with MPL command **ABORT**) and return to the point where the call was initiated. Therefore, if the homing procedure can't find the home position, you have the option to cancel it.

During the execution of a homing procedure **SRL.8** = 1. Hence you can find when a homing sequence ends, either by monitoring bit 8 from SRL or by programming the drive/motor to send a message to your host when **SRL.8** changes. As long as a homing sequence is in execution, you should not start another one. If this happens, the last homing is aborted and a warning is generated by setting **SRL.7** = 1.

***Remark:** In motion programming tools like MotionPRO Developer, ElectroCraft provides for each programmable drive/motor a collection of up to 32 homing procedures. These are predefined MPL functions, which you may call after setting the homing parameters. You may use any of these homing procedures as they are, or use them as a starting point for your own homing routines.*

Typically a homing function requires setting the following parameters before calling it:

- **CACC** – acceleration/deceleration rate for the position / speed profiles during homing
- **CDEC** – deceleration rate for quick stop when a limit switch is reached
- **CSPD** – High/normal speed for the position / speed profiles done during homing
- **HOMESPD** – Low speed for the final approach towards the home position
- **HOMEPOS** – New home position set at the end of the homing procedure

See also:

[Homing – Related MPL Instructions and Data](#)

[MPL Description](#)

6.2.3.1.6. Homing - Related MPL Instructions and Data

Parameters

CACC Acceleration/deceleration command for the position / speed profiles during homing. Measured in [acceleration units](#)

CDEC Deceleration rate during **quick stop**. Measured in [acceleration units](#)

CSPD High/normal speed command for the position / speed profiles during homing. Measured in [speed units](#)

HOMEPOS New home position set at the end of the homing procedure. Measured in [position units](#)

HOMESPD Low speed command for the final approach towards the home position. Measured in [speed units](#)

Instructions

CALLS Cancelable call of a MPL function

ABORT Abort execution of a function called with **CALLS**

SAP V32 Set actual position equal with the value of a 32-bit long variable V32. The value is measured in [position units](#)

Programming Example

```
// Position feedback: 500 lines incremental
// encoder (2000 counts/rev)
//Select homing parameters
CACC = 0.3183;//Acceleration rate = 1000[rad/s^2]
CDEC = 0.3183;//Deceleration rate = 1000[rad/s^2]
CSPD = 33.3333;//High speed = 1000[rpm]
HOMESPD = 3;//Low speed= 90[rpm]
HOMEPOS = 0;//Home position = 0[rot]
//Execute homing mode 1
CALLS HomeModel; // call HomeModel function
WaitHomingEnd:
    user_var = SRL;
    SRB user_var, 0x100, 0; // isolate SRL.8
    GOTO WaitHomingEnd, user_var, NEQ; // wait as long as SRL.8=1
HomingEnded:
    ...
```

HomeModel: // this function implements the homing procedure

...

SAP HOMEPOS; // Set home position = HOMEPOS value

RET;

See also:

[Homing – MPL Programming Details](#)

[MPL Description](#)

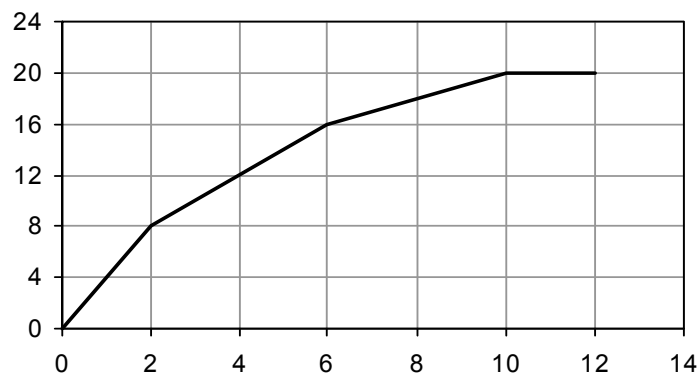
6.2.3.1.7. Contouring

In the contouring mode, you can program an arbitrary path via a series of points. Between the points, linear interpolation is performed, leading to a contour described by a succession of linear segments. The contouring mode may be executed only from a MPL program. You can't send contouring points from a host via a communication channel, like in the case of the PT mode. Depending on the control mode chosen, four options are available:

- *Position contouring* – the load/motor is controlled in position. The path represents a position reference
- *Speed contouring* – the load/motor is controlled in speed. The path represents a speed reference.
- *Torque contouring* – the motor is controlled in torque. The path represents a current reference.
- *Voltage contouring* – the motor is controlled in voltage. The path represents a voltage reference.

A contouring segment is described via the MPL command **SEG**, which has 2 parameters: **time** and reference **increment**. The **time** represents the segment duration expressed in **time units** i.e. in number of slow loop sampling periods. The reference **increment** represents the amount of reference variation per time unit i.e. per slow loop sampling period.

The contouring mode has been foreseen mainly for setup tests. However, you can also use the *position contouring* and the *speed contouring* for normal operation, as part of your motion application. You can switch at any moment to and from these 2 modes. The *torque contouring* and the *voltage contouring* have been foreseen only for setup tests. The *torque contouring* may be used, for example, to check the response of the current controllers to different input signals. Similarly, the *voltage contouring* may be used, for example, to check the motors behavior under a constant voltage or any other voltage shape.



Reference generation in contouring modes

In *position contouring* or *speed contouring*, the starting point is either the current value of the target position/speed (if **TUM1** command is set between the motion mode setting and the **UPD** command), or the actual value of the load position/speed (if **TUM1** is omitted). Therefore the contour is relative to the starting point.

In torque/voltage contouring, the starting point may be set by the user in **REF0(H)**. After reset, the default value of **REF0(H)** is zero.

In the MPL program, first the contouring mode must be set, followed by the first point. Then the contouring mode can be activated with the **UPD** command, followed by the next points. The sequence of points must end with a final point having the time interval 0.

Remarks:

- *When the last segment execution ends, the reference is kept constant at the last computed value.*
- *When a contouring sequence ends without having time value set to 0 on the last segment, the drive/motor remains in contouring mode waiting for new points. When the last segment has time value set to 0, the drive gets out from contouring mode. In order to execute other segments, the contouring mode must be set again.*

When a sequence of contour points is executed, the MPL instruction pointer IP advances as the segments described by the points are executed. When the reference generator starts working with a new segment, at MPL program level the IP advances to the execution of the **SEG** instruction. The execution of a MPL instruction for a contour segment means to copy the segment data into a local buffer and then wait (i.e. loop on the same instruction) until the previous segment, currently under execution at reference generator level will end. This procedure permits to immediately start the execution of the next contour segment when the current one ends because the next segment data are already available in a local buffer. Each time the reference generator starts to execute a new segment, the IP advances to the next contour segment and its data are transferred into the local buffer.

See also:

[Contouring – Related MPL Instructions and Data](#)

[MPL Description](#)

6.2.3.1.8. Contouring - MPL Instructions and Data

Parameters

REF0(H) Starting value for torque or voltage contouring. Measured in [current units](#) or [voltage command units](#)

Variables

TPOS Target load position – position reference computed by the reference generator at each slow loop sampling period in position or speed contouring. In speed contouring, TPOS is computed by integrating the target speed TSPD. Measured in [position units](#)

TSPD Target load speed – speed reference computed by the reference generator at each slow loop sampling period in position or speed contouring. Measured in [speed units](#)

TACC Target load acceleration – acceleration/deceleration reference computed by the reference generator at each slow loop sampling period in position and speed contouring. Measured in

[acceleration units](#)

IQREF Current reference – computed by the reference generator at each slow loop sampling period in torque contouring. Measured in [current units](#)

UQREF Voltage reference – computed by the reference generator at each slow loop sampling period in voltage contouring. Measured in [voltage command units](#)

APOS_LD Actual load position. Measured in [position units](#). Alternate name: **APOS**.

ASPD_LD Actual load speed – measured in [speed units](#)

APOS_MT Actual motor position. Measured in [motor position units](#).

ASPD_MT Actual motor speed. Measured in [motor speed units](#). Alternate name: **ASPD**

IQ Motor current. Measured in [current units](#)

Instructions

MODE PC Set position contouring mode

MODE SC Set speed contouring mode

MODE TC Set torque contouring mode

MODE VC Set voltage contouring.

SEG Time, Increment Set a contour segment where:

Time – is the segment time. It is an unsigned integer measured in [time units](#)

Increment – is the segment reference increment per time unit. It is 32-bit fixed value measured in:

- **speed units** for position contouring
- **acceleration units** for speed contouring
- **current units / time units** for torque contouring
- **voltage units / time units** for voltage contouring

TUM1 Target Update Mode 1 (TUM1). Generates new trajectory starting from the actual values of position and speed reference (i.e. don't update the reference values with load/motor position and speed)

TUM0 Target Update Mode 0 (TUM0). Generates new trajectory starting from the actual values of load/motor position and speed (i.e. updates the reference values with load/motor position and speed)

UPD Update motion parameters and start new motion mode

STOP Stop the motion.

Remarks:

- In order to activate the **TUM1** mode, execute the MPL instruction **TUM1** AFTER setting one of the contouring modes and BEFORE the **UPD** command. When the MPL command setting a contouring mode is executed, it automatically sets **TUM0** mode. However, as the new motion mode becomes effective only after the **UPD** command, a **TUM1** command will overwrite the **TUM0** mode
- Under **TUM0** mode, at the **UPD** command **TPOS=APOS_LD** and **TSPD=ASPD_LD**. In open loop control of steppers, **TUM0** is ignored as there is no position and/or speed feedback
- In setup configurations where there is no transmission ratio between the motor and the load, it is supposed that these are directly connected. In these cases: **APOS_MT=APOS_LD** and **ASPD_MT=ASPD_LD**

Programming Example

```
// Position contouring with position feedback on motor: 500 lines
// incremental encoder (2000 counts/rev)
MODE PC;//Set Position Contouring
TUM1;//Start from actual value of position reference
SEG 100U, 20.00000;// 1st point
UPD; //Execute immediate
SEG 100U, 0.00000; // 2nd point
SEG 0, 0.0; //End of contouring
```

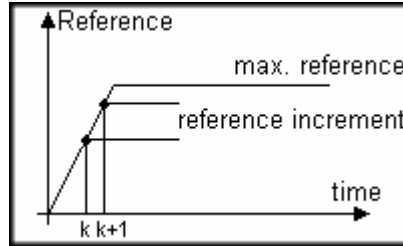
See also:

[Contouring – MPL Programming Details](#)

[MPL Description](#)

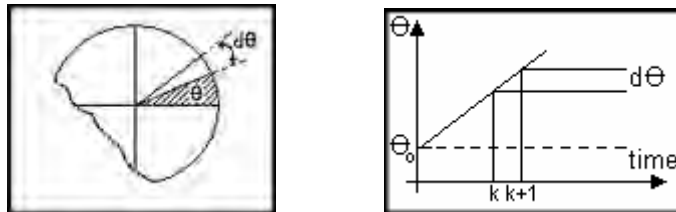
6.2.3.1.9. Test Mode - MPL Programming Details

The torque and voltage test modes have been designed to facilitate the testing during the setup phase. In these test modes, either a voltage or a torque (current) command can be set using a test reference consisting of a limited ramp (see figure below).



Reference profile in test modes

For AC motors (like for example the brushless motors), the test mode offers also the possibility to rotate a voltage or current reference vector with a programmable speed. As a result, these motors can be moved in an “open-loop” mode without using the position sensor. The main advantage of this test mode is the possibility to conduct in a safe way a series of tests, which can offer important information about the motor parameters, drive status and the integrity of the its connections.



Electrical angle setup in test modes with brushless AC motors

Remark: The Motion test is a special test mode to be used only in some special cases for drives setup. The Motion Test mode is not supposed to be used during normal operation

See also:

[Test Mode – Related MPL Instructions and Data](#)

[MPL Description](#)

6.2.3.1.10. Test Mode - MPL Instructions and Data

Parameters

[REFTST_V](#) Maximum voltage reference. Measured in [voltage units](#)

[REFTST_A](#) Maximum current reference. Measured in [current units](#)

[RINCTST_V](#) Voltage reference increment at each slow-loop sampling period. Measured in [voltage units / time units](#)

[RINCTST_A](#) Current reference increment at each slow-loop sampling period. Measured in [current units / time units](#)

[THTST](#) Initial value for the electrical angle. Measured in [electrical angle units](#)

[TINCTST](#) Electrical angle increment at each fast-loop sampling period. Measured in [electrical angle increments units](#).

Instructions

[MODE VT](#) Set voltage test mode

[MODE TT](#) Set torque test mode

[UPD](#) Update motion mode and parameters. Start motion

Programming Example

```
//Torque test mode, brushless AC motor. Drive IDM640-8EI
//with peak current 16.5A -> 32736 internal current units
//360° electric angle -> 65536 internal units
// fast loop sampling period = 0.1ms. Motor has 2 pole pairs
MODE TT; //Torque Test Mode
REFTST_A = 1984; //Reference saturation = 1[A]
RINCTST_A = 20; //Reference increment = 10[A/s]
THTST = 0; //Electric angle = 0[deg]
TINCTST = 7; //Electric angle increment ~ = 2e+002[deg/s]
UPD; //update immediate
```

See also:

[Test Mode – MPL Programming Details](#)

[MPL Description](#)

6.2.3.1.11. Motor Commands

You can apply one of following commands to the motor:

- Activate/deactivate the control loops and the power stage PWM output commands (**AXISON** / **AXISOFF**)
- Stop the motor with deceleration set in MPL parameter **CACC**
- Issue an update command, immediate (**UPD**) or when a previously programmed event occurs (**UPD!**)
- Change the value of the motor position and position reference

The **AXISON** command activates the control loops and the PWM output commands. After power on, the **AXISON** command has to be executed after the **ENDINIT** (end of initialization) command.

***Remark:** You may set the first motion mode either before or after the **AXISON** command. If the first **AXISON** is executed before setting the motion mode, the drive/motor enters in the default motion mode: voltage external online with voltage reference zero. Therefore, the drive gets zero voltage commands, until you'll set a new motion mode. If you first set a motion mode, followed by update **UPD** and then activate control with **AXISON**, the drive/motor enters directly in the desired motion mode.*

The **AXISON** command may be used to restore the normal drive operation following an **AXISOFF** command. Typically, this situation occurs at recovery from an error, following the fault reset command **FAULTR**, or after the drive/motor ENABLE input goes from status disabled to status enabled.

When **AXISON** is set after an **AXISOFF** command, the reference generator resumes its calculations from the same conditions left when the **AXISOFF** command was executed. As consequence, the values of the target position and speed provided by the reference generator may differ quite a lot from the actual values of the load position and speed which continue to be measured during the **AXISOFF** condition. In order to eliminate these differences:

- Set the motion mode, even if it is the same. The motion mode commands, automatically set the target update mode zero (**TUM0**), which updates the target position and speed with the actual measured values of the load position and speed
- Execute update command **UPD**
- Execute **AXISON** command

Example: A motor controlled in speed with a trapezoidal profile, was stopped with an **AXISOFF** command. In order to resume the normal operation, with the same parameters, the MPL program can be:

```
// Resume speed profile operation from AXISOFF
MODE SP;    // set speed profile mode
UPD;        // update immediate
AXISON;     // motion starts.
            //The initial value for target speed is 0 because was
            //updated with the actual motor speed which is 0
            //because the motor is still
```

The **AXISOFF** command deactivates the control loops, the reference generator and the PWM output commands (all the switching devices are off). However, all the measurements remain active and therefore the motor currents, speed, position as well as the supply voltage continue to be updated and monitored. If the **AXISOFF** command is applied during motion, it leaves the motor free running. Typically,

the **AXISOFF** command is used when a fault condition is detected, for example when a protection is triggered.

Fault conditions trigger MPL interrupts. Each drive/motor has a built-in set of MPL interrupt service routines (ISR), which are automatically activated after power-on. In these routines, the default action for fault conditions is an **AXISOFF** command. If needed, you may replace any built-in ISR with your own ISR and thus, adapt the fault treatment to your needs.

***Remark:** The **AXISOFF** command is automatically generated when the Enable input goes from enabled to disabled status. If the Enable input returns to the enabled status, the **AXISON** command is automatically generated if*

- **ACR.3** = 1, or
- **ACR.1** = 1 i.e. the drive/motor is set to start automatically after power-on with an external

***Remark:** **SRL.15** shows the **AXISON/AXISOFF** condition and **SRH.15** shows a fault condition*

The **STOP** command stops the motor with the deceleration rate set in MPL parameter **CACC**. The drive/motor decelerates following a trapezoidal position or speed profile. If the **STOP** command is issued during the execution of an S-curve profile, the deceleration profile may be chosen between a trapezoidal or an S-curve profile (see S-curve dialogue settings). You can detect when the motor has stopped by setting a motion complete event (**IMC**)and waiting until the event occurs (**WAIT!**). The **STOP** command can be used only when the drive/motor is controlled in position or speed.

Remarks:

- *In order to restart after a **STOP** command, you need to set again the motion mode. This operation disables the stop mode and allows the motor to move*
- *When **STOP** command is sent via a communication channel, it will automatically stop any MPL program execution, to avoid overwriting the **STOP** command from the MPL program*

If an error requiring the immediate stop of the motion occurs (like triggering a limit switch or following a command error), the drive/motor enters automatically in the **quick stop mode**. This mode stops the motor with a trapezoidal profile, using the deceleration rate set in the MPL parameter **CDEC**.

When an immediate update command **UPD** is executed, the last motion mode programmed together with the latest motion parameters are taken into consideration. During motion execution, you can freely change the motion mode and/or its parameters. These changes will have no effect until an update command is executed.

If you intend to perform an update when a specific condition occurs, you can set an event which monitors the condition, followed by an update on event command **UPD!**. When the monitored condition occurs, the update will be automatically performed. Once you have set an update on event **UPD!**, you can either wait for the monitored event to occur, or perform other operations.

The MPL command **SAP** offers you the possibility to set / change the referential for position measurement by changing simultaneously the load position **APOS** and the target position **TPOS** values, while keeping the same position error.

You can specify the new position either as an immediate value or via a 32-bit long variable. **SAP** command can be executed at any moment during motion. When **SAP** command is executed, the following operations are performed:

- Under **TUM1**, i.e. if **TUM1** command has been executed after the last motion mode setting and before the last **UPD**, the target/reference position **TPOS** is set equal with the new position value and the actual motor position **APOS** is set equal with the new position reference minus the position error (**POSERR**)

```
TPOS = new_value;
```

```
APOS = TPOS - POSERR;
```

- Under **TUM0**, i.e. if **TUM1** command has not been executed after the last motion mode setting and before the last **UPD**, the actual load position **APOS** is set equal with the new position value and the target/reference position **TPOS** is set equal with the new position plus the position error (**POSERR**)

```
APOS = new_value;
```

```
TPOS = APOS + POSERR;
```

The MPL command **STA** sets the target position equal with the actual position: $TPOS = APOS$.

See also:

[Motor Commands – Related MPL Instructions and Data](#)

[MPL Description](#)

6.2.3.1.12. Motor Commands - Related MPL Instructions and Data

Parameters

CACC Deceleration rate following a **STOP** command

CDEC Deceleration rate during **quick stop**

MPL Variables

APOS_LD Actual load position. Measured in [position units](#). Alternate name: **APOS**

TPOS Target position – position reference computed by the reference generator at each slow loop sampling period. Measured in [position units](#)

POSERR Represents the value of load position error, computed as the difference between the target position and the measured position of the load

Instructions

AXISON Set axis ON. Activate control loops and PWM commands

AXISOFF Set axis OFF. Deactivate control loops and PWM commands

STOP Stop motion with the acceleration/deceleration set in **CACC**

<u>UPD</u>	Update immediate motion mode and parameters. Start motion
<u>UPD!</u>	Update the motion mode and parameters when the programmed event occurs
<u>SAP V32</u>	Set V32 in the actual or target position. V32 is either a 32-bit immediate value or a long MPL data (user variable) containing the value to set
<u>STA</u>	Set target position TPOS equal with the actual position APOS

Programming Example

```

// Position profile. Position feedback: 500 lines incremental
// encoder (2000 counts/rev)
CACC = 0.3183;//acceleration rate = 1000[rad/s^2]
CSPD = 33.3333;//slew speed = 1000[rpm]
CPOS = 6000;//position command = 3[rot]
CPR; //position command is relative
MODE PP;
TUM1; //set Target Update Mode 1
UPD; //execute immediate
...
STOP; // stop motion before reaching the final position
//Define event: When actual position remains inside
//a settle band around the position to reach
SRB UPGRADE, 0xFFFF, 0x0800;
!MC;
WAIT!;//Wait until the event occurs i.e. motor stops
SAP 0; // Set actual position value to 0[rot]
...
MODE PP;
UPD; //execute immediate - restart motion after a STOP command

```

See also:

[Motor Commands – MPL Programming Details](#)

[MPL Description](#)

6.2.3.2. Program flow control

6.2.3.2.1. Events on drives with built-in Motion Controller

6.2.3.2.1.1. List of Events

An *event* is a programmable condition, which once set, is monitored for occurrence. You can do the following actions in relation with an event:

- A) Change the motion mode and/or the motion parameters, when the event occurs
- B) Stop the motion when the event occurs
- C) Wait for the programmed event to occur

Remark: *The programmed event is automatically erased if the event is reached, if the timeout for the wait is reached or if a new event is programmed.*

Only a single event can be programmed at a time. The Table below presents all the events with their mnemonic and a short description.

No.	Mnemonic	Event Description
1	!MC	When the actual motion is completed
2	!AMPU value32 !AMPU var32	When the motor absolute position is equal or under a 32-bit long value or the value of a long variable
3	!AMPO value32 !AMPO var32	When the motor absolute position is equal or over a 32-bit long value or the value of a long variable
4	!ALPU value32 !ALPU var32	When the load absolute position is equal or under a 32-bit long value or the value of a long variable
5	!ALPO value32 !ALPO var32	When the load absolute position is equal or over a 32-bit long value or the value of a long variable
6	!RPU value32 !RPU var32	When the load/motor relative position is equal or under a 32-bit long value or the value of a long variable
7	!RPO value32 !RPO var32	When the load/motor relative position is equal or over a 32-bit long value or the value of a long variable;
8	!MSU value32 !MSU var32	When the motor speed is equal or under a 32-bit fixed value or the value of a fixed variable
9	!MSO value32 !MSO var32	When the motor speed is equal or over a 32-bit fixed value or the value of a fixed variable
10	!LSU value32 !LSU var32	When the load speed is equal or under a 32-bit fixed value or the value of a fixed variable
11	!LSO value32 !LSO var32	When the load speed is equal or over a 32-bit fixed value or the value of a fixed variable
12	!RT value32 !RT var32	After a wait time (measured from the event setting) equal with a 32-bit long value or the value of a long variable
13	!PRU value32 !PRU var32	When the position reference is equal or under a 32-bit value or the value of a long variable
14	!PRO value32 !PRO var32	When the position reference is equal or over a 32-bit value or the value of a long variable
15	!SRU value32 !SRU var32	When the speed reference is equal or under a 32-bit value or the value of a fixed variable
16	!SRO value32 !SRO var32	When the speed reference is equal or over a 32-bit value or the value of a fixed variable
17	!TRU value32 !TRU var32	When the torque reference is equal or under a 32-bit value or the value of a fixed variable
18	!TRO value32 !TRO var32	When the torque reference is equal or over a 32-bit value or the value of a fixed variable
19	!CAP	When the selected capture input goes low or high
20	!LSP	When the positive limit switch input (LSP) goes low or high
21	!LSN	When the negative limit switch input (LSN) goes low or high
22	!IN#n 0	When digital input #n goes low
23	!IN#n 1	When digital input #n goes high
24	!VU var32a, value32 !VU var32a, var32b	When the long/fixed variable var32a is equal or under the 32-bit long/fixed value32 or the value of long/fixed variable var32b
25	!VO var32a, value32 !VO var32a, var32b	When the long/fixed variable var32a is equal or under the 32-bit long/fixed value32 or the value of long/fixed variable var32b

If you want to change the motion mode and/or the motion parameters when an event occurs, you must do the following:

- Program/define one of the above events
- Set the new motion mode and/or the motion parameters

-
- Set one of the MPL commands: **UPD!** (Update on event) or **STOP!** (Stop on event), which will become effective when the programmed event occurs

Remark: After you have programmed a new motion mode and/or new motion parameters with update on event, you need to wait until the programmed event occurs, using the MPL command **WAIT!**. Otherwise, the program will continue with the next instructions that may override the event monitoring.

The instruction **WAIT!**, stops the MPL program further execution, until the programmed event occurs. During this period, only the MPL commands received via a communication channel are processed. You may also specify the time limit for the wait, by adding a time value after the **WAIT!** command: **WAIT! time_limit**. If the monitored event doesn't occur in the time limit set, the wait loop is interrupted, the event checking is reset and the MPL program passes to the next instruction.

See also:

[Events – When the actual motion is completed. Related MPL Instructions and Data](#)

[Events – Function of motor or load position Related MPL Instructions and Data](#)

[Events – Function of motor or load speed Related MPL Instructions and Data](#)

[Events – After a wait time Related MPL Instructions and Data](#)

[Events – Function of reference Related MPL Instructions and Data](#)

[Events – Function of inputs status Related MPL Instructions and Data](#)

[Events – Function of 32-bit variable value Related MPL Instructions and Data](#)

6.2.3.2.1.2. When the actual motion is completed

Setting this event allows you to detect when a motion is completed. You can use, for example, this event to start your next move only after the actual move is finalized.

The motion complete condition is set in the following conditions:

- During position control:
 - If `UPGRADE.11=1`, when the position reference arrives at the position to reach (commanded position) and the position error remains inside a *settle band* for a preset *stabilize time* interval
 - If `UPGRADE.11=0`, when the position reference arrives at the position to reach (commanded position)
- During speed control, when the speed reference arrives at the commanded speed

The motion complete condition is reset when a new motion is started i.e. when the update command – `UPD` is executed.

Remark: In case of steppers controlled open-loop, the motion complete condition for positioning is always set when the position reference arrives at the position to reach independently of the `UPGRADE.11` status.

Parameters

[POSOKLIM](#) Specifies the settle band when `UPGRADE.11=1`. Measured in [position units](#)

[TONPOSOK](#) Specifies the stabilize time `UPGRADE.11=1`. Measured in [time units](#)

UPGRADE MPL register. When **UPGRADE.11=1**, the motion complete is set when commanded/target position is reached and the position error is inside a settle band for a preset stabilize time. When **UPGRADE.11=0**, the motion complete is set when commanded/target position speed is reached

Instructions

IMC Set event when the actual position is completed

UPD! Update the motion mode and/or the motion parameters when the programmed event occurs

STOP! Stop motion with the acceleration/deceleration set in **CACC**, when the programmed event occurs

WAIT! **value16** Wait until the programmed event occurs. If the command is followed by **value16**, the wait ends after the time interval specified in this 16-bit integer value. Value16 is measured in **time units**

Programming Example 1

```
//Execute successive position profiles
//Position feedback: 500 lines encoder (2000 counts/rev)
CACC = 0.3183;//acceleration rate = 1000[rad/s^2]
CSPD = 100;//slew speed = 3000[rpm]
CPOS = 20000;//position command = 10[rot]
CPR; //position command is relative
MODE PP;
TUM1; //set Target Update Mode 1
UPD; //execute immediate
!MC; WAIT!; // set event and wait for motion complete
// start here next move
```

Programming Example 2

```
//Execute successive position profiles
// Position feedback: 500 lines encoder (2000 counts/rev)
// First move
CACC = 0.3183;//acceleration rate = 1000[rad/s^2]
CSPD = 100;//slew speed = 3000[rpm]
CPOS = 20000;//position command = 10[rot]
CPR; //position command is relative
MODE PP;
TUM1; //set Target Update Mode 1
UPD; //start first move
```

```
// set motion complete parameters
POSOKLIM = 20; //Set settle band to 0.01[rot]
TONPOSOK = 10; //Set stabilize time to 10[ms]
SRB UPGRADE, 0xFFFF, 0x0800;

!MC; // set event when motion is complete
// Prepare data for second move
CPOS = 10000;//new position command = 5[rot]
CPR; //position command is relative
MODE PP;
TUM1; //set Target Update Mode 1
UPD!; //execute on event
WAIT!;//Wait until the event occurs. When the event
// occurs the second move will start
```

See also:

[Events – Function of motor or load position. Related MPL Instructions and Data](#)

[Events – Function of motor or load speed. Related MPL Instructions and Data](#)

[Events – After a wait time. Related MPL Instructions and Data](#)

[Events – Function of reference. Related MPL Instructions and Data](#)

[Events – Function of inputs status. Related MPL Instructions and Data](#)

[Events – Function of 32-bit variable value. Related MPL Instructions and Data](#)

[Events – MPL Programming Details](#)

6.2.3.2.1.3. Function of motor or load position

Setting any of these events allows you to detect when the load or motor absolute or the relative position is equal or over/under a value or the value of a variable.

The absolute load or motor position is the measured position of the load or motor. The relative position is the load displacement from the beginning of the actual movement. For example if a position profile was started with the absolute load position 50 revolutions, when the absolute load position reaches 60 revolutions, the relative motor position is 10 revolutions.

Remark: The origin for the relative position measurement (MPL variable **POS0**) is set function of the target update mode. Under **TUM1**, **POS0 = TPOS**. Under **TUM0**, **POS0=APOS_LD**.

Variables

<u>POS0</u>	Origin for the relative position measurement for the position events. Measured in <u>position units</u>
<u>RPOS</u>	Relative load position for the position events. It is computed with formula: RPOS = APOS_LD – POS0 . Measured in <u>position units</u>
<u>TPOS</u>	Target position – position reference computed by the reference generator at each slow loop sampling period. Measured in <u>position units</u>
<u>APOS_LD</u>	Actual load position. Measured in <u>position units</u> . Alternate name: <u>APOS</u>
<u>APOS_MT</u>	Actual motor position. Measured in <u>motor position units</u> .

Instructions

<u>!AMPOvalue32</u>	Set event when motor absolute position is equal or over value32. Value32 is a long integer. Measured in <u>motor position units</u>
<u>!AMPOvar32</u>	Set event when motor absolute position is equal or over var32. Var32 is a long integer MPL parameter or variable. Measured in <u>motor position units</u>
<u>!ALPOvalue32</u>	Set event when load absolute position is equal or over value32. Value32 is a long integer. Measured in <u>position units</u>
<u>!ALPOvar32</u>	Set event when load absolute position is equal or over var32. Var32 is a long integer MPL parameter or variable. Measured in <u>position units</u>
<u>!AMPUvalue32</u>	Set event when motor absolute position is equal or under value32. Value32 is a long integer. Measured in <u>motor position units</u>
<u>!AMPUvar32</u>	Set event when motor absolute position is equal or under var32. Var32 is a long integer MPL parameter or variable. Measured in <u>motor position units</u>
<u>!ALPUvalue32</u>	Set event when load absolute position is equal or under value32. Value32 is a long integer. Measured in <u>position units</u>
<u>!ALPUvar32</u>	Set event when load absolute position is equal or under var32. Var32 is a long integer MPL parameter or variable. Measured in <u>position units</u>

<u>!RPO</u>value32	Set event when load relative position is equal or over value32. Value32 is a long integer. Measured in position units
<u>!RPO</u>var32	Set event when load relative position is equal or over var32. Var32 is a long integer MPL parameter or variable. Measured in position units
<u>!RPU</u>value32	Set event when load relative position is equal or under value32. Value32 is a long integer. Measured in position units
<u>!RPU</u>var32	Set event when load relative position is equal or under var32. Var32 is a long integer MPL parameter or variable. Measured in position units
<u>UPD!</u>	Update the motion mode and/or the motion parameters when the programmed event occurs
<u>STOP!</u>	Stop motion with the acceleration/deceleration set in CACC , when the programmed event occurs
<u>WAIT!</u> value16	Wait until the programmed event occurs. If the command is followed by value16 , the wait ends after the time interval specified in this 16-bit integer value. Value16 is measured in time units

Programming Example

```
//Stop motion when motor position > 3 rev
//Position feedback: 500 lines encoder (2000 counts/rev)
!AMPO 6000; //Set event: when motor absolute position is >= 3 rev
STOP!;//Stop the motion when the event occurs
WAIT!;//Wait until the event occurs
```

See also:

- [Events – When the actual motion is completed. Related MPL Instructions and Data](#)
- [Events – Function of motor or load speed Related MPL Instructions and Data](#)
- [Events – After a wait time Related MPL Instructions and Data](#)
- [Events – Function of reference Related MPL Instructions and Data](#)
- [Events – Function of inputs status Related MPL Instructions and Data](#)
- [Events – Function of 32-bit variable value Related MPL Instructions and Data](#)
- [Events – MPL Programming Details](#)

6.2.3.2.1.4. Function of motor or load speed

Setting any of these events allows you to detect when the load or motor speed is equal or over/under a value or the value of a variable.

Variables

- ASPD_LD** Actual load speed – measured in [speed units](#)
- APOS_MT** Actual motor position. Measured in [motor position units](#).
- ASPD_MT** Actual motor speed. Measured in [motor speed units](#). Alternate name: [ASPD](#)

Instructions

- !MSOvalue32** Set event when motor speed is equal or over value32. Value32 is a fixed value. Measured in [motor speed units](#)
- !MSOvar32** Set event when motor speed is equal or over var2. Var32 is a fixed MPL parameter or variable. Measured in [motor speed units](#)
- !LSOvalue32** Set event when load speed is equal or over value32. Value32 is a fixed value. Measured in [speed units](#)
- !LSOvar32** Set event when load speed is equal or over var2. Var32 is a fixed MPL parameter or variable. Measured in [speed units](#)
- !MSUvalue32** Set event when motor speed is equal or under value32. Value32 is a fixed value. Measured in [motor speed units](#)
- !MSUvar32** Set event when motor speed is equal or under var2. Var32 is a fixed MPL parameter or variable. Measured in [motor speed units](#)
- !LSUvalue32** Set event when load speed is equal or under value32. Value32 is a fixed value. Measured in [speed units](#)
- !LSUvar32** Set event when load speed is equal or under var2. Var32 is a fixed MPL parameter or variable. Measured in [speed units](#)
- UPD!** Update the motion mode and/or the motion parameters when the programmed event occurs
- STOP!** Stop motion with the acceleration/deceleration set in **CACC**, when the programmed event occurs
- WAIT! value16** Wait until the programmed event occurs. If the command is followed by **value16**, the wait ends after the time interval specified in this 16-bit integer value. Value16 is measured in [time units](#)

Programming Example

```
//Motor is decelerating. Start a position profile when motor
//speed <= 600 rpm
//Position feedback: 500 lines encoder (2000 counts/rev)
!MSU 20; //Set event: when motor speed is <= 600 rpm
// prepare new motion mode
CACC = 0.3183;//acceleration rate = 1000[rad/s^2]
CSPD = 100;//slew speed = 3000[rpm]
CPOS = 20000;//position command = 10[rot]
CPR; //position command is relative
MODE PP;
TUM1; //set Target Update Mode 1
UPD!; //execute on event
WAIT!;//Wait until the event occurs
```

See also:

[Events – When the actual motion is completed. Related MPL Instructions and Data](#)

[Events – Function of motor or load position Related MPL Instructions and Data](#)

[Events – After a wait time Related MPL Instructions and Data](#)

[Events – Function of reference Related MPL Instructions and Data](#)

[Events – Function of inputs status Related MPL Instructions and Data](#)

[Events – Function of 32-bit variable value Related MPL Instructions and Data](#)

[Events – MPL Programming Details](#)

6.2.3.2.1.5. After a wait time

Setting this event allows you to introduce a delay in the execution of the MPL program.

The monitored event is: when relative time (MPL variable **RTIME**) is equal or over a value or the value of a variable. The relative time **RTIME** is computed with formula:

$$\mathbf{RTIME} = \mathbf{ATIME} - \mathbf{TIME0},$$

where **ATIME** is a 32-bit absolute time counter, incremented by 1 at each slow loop sampling period and **TIME0** is the **ATIME** value when the wait event was set. After power on, **TIME0** is set to 0. **RTIME** is updated together with **ATIME**, at each slow loop sampling period.

Remark:

- **ATIME** and **RTIME** start *ONLY* after the execution of the **ENDINIT** (end of initialization) command. Therefore you should not set wait events before executing this command
- After setting a wait time event, in order to effectively execute the time delay, you need to wait for the event to occur, using **WAIT!**

Variables

ATIME Absolute time counter. Incremented at each slow loop sampling period. Starts after execution of **ENDINIT** command. Measured in [time units](#)

RTIME Relative time. **RTIME = ATIME – TIME0**. Measured in [time units](#)

TIME0 Absolute time when last wait event was set. Measured in [time units](#)

Instructions

IRT value32 Introduce a time delay equal with value32. Value32 is a 32-bit long integer number. Measured in [time units](#)

IRT var32 Introduce a time delay equal with value of var32. Var32 is a 32-bit long integer MPL variable or parameter. Measured in [time units](#)

UPD! Update the motion mode and/or the motion parameters when the programmed event occurs

STOP! Stop motion with the acceleration/deceleration set in **CACC**, when the programmed event occurs

WAIT! value16 Wait until the programmed event occurs. If the command is followed by **value16**, the wait ends after the time interval specified in this 16-bit integer value. Value16 is measured in [time units](#)

Programming Example

```
//Introduce a 100 ms delay
!RT 100;    // set event: After a wait of 100 slow-loop periods
            // 1 slow-loop period = 1ms
WAIT!;     // wait the event to occur
```

See also:

[Events – When the actual motion is completed. Related MPL Instructions and Data](#)

[Events – Function of motor or load position Related MPL Instructions and Data](#)

[Events – Function of motor or load speed Related MPL Instructions and Data](#)

[Events – Function of reference Related MPL Instructions and Data](#)

[Events – Function of inputs status Related MPL Instructions and Data](#)

[Events – Function of 32-bit variable value Related MPL Instructions and Data](#)

[Events – MPL Programming Details](#)

6.2.3.2.1.6. Function of reference

Setting any of these events allows you to detect when the position or speed or torque reference is equal or over/under a value or the value of a variable. Use:

- Position reference events, only when position control is performed
- Speed reference events, only when speed control is performed
- Torque reference events, only when torque control is performed

Remark: *Setting an event based on the position or speed reference is particularly useful for open loop operation where motor position and speed is not available*

Variables

TPOS Target load position – position reference computed by the reference generator at each slow loop sampling period, when position or speed control is performed. Measured in [position units](#)

TSPD Target load speed – speed reference computed by the reference generator at each slow loop sampling period, when position or speed control is performed. Measured in [speed units](#)

IQREF Current reference – Measured in [current units](#)

TREF Target reference. It is a:

- Position reference, when position control is performed
- Speed reference, when speed control is performed
- Current/torque reference, when torque control is performed
- Voltage reference, when voltage control is performed

Function of the control mode, it is measured in [position units](#) or [speed units](#) or [current units](#) or [voltage command units](#)

Instructions

!PROvalue32 Set event if position reference is equal or over value32. Value32 is a long integer value. Measured in [position units](#)

!PROvar32 Set event if position reference is equal or over var32. Var32 is a long integer MPL parameter or variable. Measured in [position units](#)

!PRUvalue32 Set event if position reference is equal or under value32. Value32 is a long integer value. Measured in [position units](#)

!PRUvar32 Set event if position reference is equal or under var32. Var32 is a long integer MPL parameter or variable. Measured in [position units](#)

!SROvalue32 Set event when speed reference is equal or over value32. Value32 is a fixed value. Measured in [speed units](#)

!SROvar32 Set event when speed reference is equal or over var32. Var32 is a fixed MPL parameter or variable. Measured in [speed units](#)

-
- !SRUvalue32** Set event when speed reference is equal or under value32. Value32 is a fixed value. Measured in [speed units](#)
- !SRUvar32** Set event when torque reference is equal or under var32. Var32 is a fixed MPL parameter or variable. Measured in [speed units](#)
- !TROvalue32** Set event when torque reference is equal or over value32. Value32 is a fixed value. Measured in [current units](#)
- !TROvar32** Set event when speed reference is equal or over var32. Var32 is a fixed MPL parameter or variable. Measured in [current units](#)
- !TRUvalue32** Set event when torque reference is equal or under value32. Value32 is a fixed value. Measured in [current units](#)
- !TRUvar32** Set event when speed reference is equal or under var32. Var32 is a fixed MPL parameter or variable. Measured in [current units](#)
- UPD!** Update the motion mode and/or the motion parameters when the programmed event occurs
- STOP!** Stop motion with the acceleration/deceleration set in **CACC**, when the programmed event occurs
- WAIT! value16** Wait until the programmed event occurs. If the command is followed by **value16**, the wait ends after the time interval specified in this 16-bit integer value. Value16 is measured in [time units](#)

Programming Example

```
// Motor will reach a hard stop. Disable control when torque
// reference > 1 A = 1984 internal current units
!TRO 1984.0; // set event when torque reference > 1 A
WAIT!;//Wait until the event occurs
AXISOFF; // disable control
```

See also:

[Events – When the actual motion is completed. Related MPL Instructions and Data](#)

[Events – Function of motor or load position Related MPL Instructions and Data](#)

[Events – Function of motor or load speed Related MPL Instructions and Data](#)

[Events – After a wait time Related MPL Instructions and Data](#)

[Events – Function of inputs status Related MPL Instructions and Data](#)

[Events – Function of 32-bit variable value Related MPL Instructions and Data](#)

[Events – MPL Programming Details](#)

6.2.3.2.1.7. Function of inputs status

Setting any of these events allows you to detect when:

- A transition occurs on one of the 2 capture inputs. On these inputs, are usually connected the 1st and 2nd encoder index signals
- A transition occurs on one of the 2 limit switch inputs
- A general purpose digital input changes its status

Capture and limit switch inputs events

The capture inputs and the limit switch inputs can be programmed to sense either a low to high or high to low transition. When the programmed transition occurs on either of these inputs, the following happens:

- Motor position **APOS_MT** is captured and memorized in the MPL variable **CAPPOS**, except the case of open-loop systems, where reference position **TPOS** is captured instead
- Master position **APOS2** or load position **APOS_LD** is captured and memorized in the MPL variable **CAPPOS2**, except the case of steppers controlled open loop with an encoder on the load, when load position is captured in **CAPPOS**.

The selection between master and load position is done as follows: load position is saved in **CAPPOS2** only for the setup configurations which use different sensors for load and motor and foresee a transmission ratio between them. For all the other setup configurations, the master position is saved in **CAPPOS2**. The master position is automatically computed when pulse and direction signals or quadrature encoder signals are connected to their dedicated inputs. More details about the capture process are presented at [Special I/O – MPL Programming Details](#)

Remarks:

- *If both capture inputs are activated in the same time, the capture event is set by the capture input that is triggered first. The capture event makes no difference between the two capture inputs.*
- *If the drive/motor accepts CANopen protocol, the home input is the same with the 2nd encoder index. Therefore, the home input can be programmed like a capture input to sense transitions and to memorize the load and master position when the transition occurs.*

In order to set an event on a capture input, you need to:

- 1) Enable the capture input for the detection of a low->high or a high-> low transition, using one of the MPL instructions: **ENCAPI0, ENCAPI1, EN2CAPI0, EN2CAPI1**
- 2) Set a capture event, with the MPL instruction: **!CAP**
- 3) Wait for the event to occur, with the MPL instruction: **WAIT!**

Remarks:

- *When the programmed transition is detected, the capture input is automatically disabled. In order to use it again, you need to enable it again for the desired transition*
- *You may also disable a capture input (i.e. its capability to detect a programmed transition) previously enabled, using the MPL commands: **DISCAPI, DIS2CAPI***

In order to set an event on a limit switch input, you need to:

- 1) Enable the limit switch input for the detection of a low->high or a high-> low transition, using one of the MPL instructions: **ENLSP0, ENLSP1, ENLSN0, ENLSN1**

2) Set a limit switch event with one of the MPL instructions: **!LSP, !LSN**

3) Wait for the event to occur, with the MPL instruction: **WAIT!**

Remarks:

- *When the programmed transition is detected, the limit switch input is automatically disabled (for sensing transitions). In order to use it again, you need to enable it again for the desired transition*
- *You may also disable a limit switch input (i.e. its capability to detect a programmed transition) previously enabled, using the MPL commands: **DISLSP, DISLSN***

Variables

CAPPOS Position captured when programmed transition occurs on 1st capture/encoder index input. Measured in [motor position units](#), except the case of stepper motors, when it is measured in [position units](#)

CAPPOS2 Position captured when programmed transition occurs on 2nd capture/encoder index input. Measured in [position units](#) when load position is captured, or in [master position units](#) when master position is captured

APOS2 Master position computed by the slaves from pulse & direction or quadrature encoder inputs. Measured in [master position units](#)

TPOS Target position – position reference computed by the reference generator at each slow loop sampling period. Measured in [position units](#)

APOS LD Actual load position. Measured in [position units](#). Alternate name: [APOS](#)

APOS MT Actual motor position. Measured in [motor position units](#).

Instructions:

ICAP Set event on capture inputs

ENCAP10 Enable 1st capture/encoder index input to detect a high to low transition

EN2CAP10 Enable 2nd capture/encoder index input to detect a high to low transition

ENCAP11 Enable 1st capture/encoder index input to detect a low to high transition

EN2CAP11 Enable 2nd capture/encoder index input to detect a low to high transition

!LSN Set event on negative limit switch input

!LSP Set event on positive limit switch input

ENLSP0 Enable positive limit switch input to detect a high to low transition

ENLSN0 Enable negative limit switch input to detect a high to low transition

ENLSP1 Enable positive limit switch input to detect a low to high transition

ENLSN1 Enable negative limit switch input to detect a low to high transition

DISCAP1 Disable 1st capture/encoder index input to detect transitions

DIS2CAP1 Disable 2nd capture/encoder index input to detect transitions

-
- DISLSP** Disable positive limit switch input to detect transitions
- DISLSN** Disable negative limit switch input to detect transitions
- UPD!** Update the motion mode and/or the motion parameters when the programmed event occurs
- STOP!** Stop motion with the acceleration/deceleration set in **CACC**, when the programmed event occurs
- WAIT! value16** Wait until the programmed event occurs. If the command is followed by **value16**, the wait ends after the time interval specified in this 16-bit integer value. Value16 is measured in **time units**

Programming Example

```
//Stop motion on next encoder index
ENCAP11; //Set event: When the encoder index goes low->high
!CAP;
STOP!;//Stop the motion when event occurs
WAIT!;//Wait until the event occurs
// now load/motor is in deceleration
CPOS = CAPPOS; // new command position = captured position
CPA; //position command is absolute
MODE PP;
TUM1; //set Target Update Mode 1
UPD; //execute immediate
!MC; WAIT!; //wait for completion
```

General purpose digital inputs events

You can program an event on any general-purpose digital input. The event can be set when the input is high (after a low to high transition) or low (after a high to low transition).

A general purpose input event is checked at each slow loop sampling period, when the status of the selected input is compared with the one set in the event. A match triggers the event.

Instructions

- !IN#n 1** Set event when the Input #n is high
- !IN#n 0** Set event when input #n is low
- UPD!** Update the motion mode and/or the motion parameters when the programmed event occurs
- STOP!** Stop motion with the acceleration/deceleration set in **CACC**, when the programmed event occurs

WAIT! value16 Wait until the programmed event occurs. If the command is followed by **value16**, the wait ends after the time interval specified in this 16-bit integer value. Value16 is measured in **time units**

Programming Example

```
// Start motion when digital input #36 is high
!IN#36 1; // set event when input #36 is high
//Position profile. Position feedback: 500-lines encoder
CACC = 0.3183;//acceleration rate = 1000[rad/s^2]
CSPD = 100;//slew speed = 3000[rpm]
CPOS = 20000;//position command = 10[rot]
CPA; //position command is absolute
MODE PP;
TUM1; //set Target Update Mode 1
UPD!; //execute on event
WAIT!;//Wait until the event occurs
```

See also:

[Events – When the actual motion is completed. Related MPL Instructions and Data](#)

[Events – Function of motor or load position Related MPL Instructions and Data](#)

[Events – Function of motor or load speed Related MPL Instructions and Data](#)

[Events – After a wait time Related MPL Instructions and Data](#)

[Events – Function of reference Related MPL Instructions and Data](#)

[Events – Function of 32-bit variable value Related MPL Instructions and Data](#)

[Events – MPL Programming Details](#)

[Special I/O – MPL Programming Details](#)

6.2.3.2.1.8. Function of variable value

Setting any of these events allows you to detect when a *selected variable* is equal or over/under a value or the value of a variable. The *selected variable* can be any 32-bit MPL variable, long or fixed.

Instructions

IVO var32, value32 Set event when 32-bit MPL parameter or variable var32 is equal or over value32. Value32 is either a long or a fixed, depending on var32 type.

IVO var32, var32c Set event when 32-bit MPL parameter or variable var32 is equal or over var32c. Var32c is a 32-bit MPL parameter of variable of the same type like var32.

IVU var32, value32 Set event when 32-bit MPL parameter or variable var32 is equal or under value32. Value32 is either a long or a fixed, depending on var32 type.

IVU var32, var32c Set event when 32-bit MPL parameter or variable var32 is equal or under var32c. Var32c is a 32-bit MPL parameter of variable of the same type like var32.

UPD! Update the motion mode and/or the motion parameters when the programmed event occurs

STOP! Stop motion with the acceleration/deceleration set in **CACC**, when the programmed event occurs

WAIT! value16 Wait until the programmed event occurs. If the command is followed by **value16**, the wait ends after the time interval specified in this 16-bit integer value. Value16 is measured in [time units](#)

Programming Example

```
//Wait until master position MREF > 500 counts, then activate
//electronic gearing slave mode
!VO MREF, 500; //Set event when variable MREF is <= 500
GEAR = 1; // gear ratio
GEARMASTER = 1; // Gear ratio denominator
GEARSLAVE = 1; // Gear ratio numerator
EXTREF 2; // read master from 2nd encoder or pulse & dir
MASTERRES = 2000; // master resolution
MODE GS; //Set as slave, position mode
TUM1; //Set Target Update Mode 1
SRB UPGRADE, 0xFFFF, 0x0004;//UPGRADE.2=1 enables CACC limitation
CACC = 0.3183; //Limit maximum acceleration at 1000[rad/s^2]
UPD!; //execute on event
```

See also:

[Events – When the actual motion is completed. Related MPL Instructions and Data](#)

[Events – Function of motor or load position Related MPL Instructions and Data](#)

[Events – Function of motor or load speed Related MPL Instructions and Data](#)

[Events – After a wait time Related MPL Instructions and Data](#)

[Events – Function of reference Related MPL Instructions and Data](#)

[Events – Function of inputs status Related MPL Instructions and Data](#)

[Events – MPL Programming Details](#)

6.2.3.2.2. Jumps and Function Calls

The MPL offers the possibility to make unconditional or conditional jumps and calls of functions.

The jumps are executed with MPL command **GOTO**, followed by a *jump address*. The *jump address* may be specified with an immediate value, through a label or via 16-bit MPL variable containing it. A label can be any user-defined string of up to 32 characters starting with an alphanumeric character or with underscore. A label starts from the first column of a text line and ends with a colon (:). It contains the MPL program address of the next MPL instruction. Using an assignment instruction of type: `user_var = label;` you can set a *jump address* in an integer MPL variable.

In a conditional jump, a condition is tested. If the condition is true the jump is executed, else the next MPL command is carried out. The condition is specified by a *test variable* and a *test condition* both added after the *jump address*. The *test variable* is always compared with zero. The possible *test conditions* are: < 0 , ≤ 0 , > 0 , ≥ 0 , $= 0$, $\neq 0$.

The calls are executed with MPL command **CALL**, followed by a *MPL function address*. A MPL function is a set of MPL commands which starts with a label and ends with the **RET** instruction. The label gives the *MPL function address* and name. Like the *jump address*, the *MPL function address* may be specified with an immediate value, through a label or via 16-bit MPL variable containing it.

In a conditional call, a condition is tested. If the condition is true the MPL function is executed, else the next MPL command is carried out. The condition is specified by a *test variable* and a *test condition* added after the *MPL function address*. The *test variable* is always compared with zero. The possible *test conditions* are: < 0 , ≤ 0 , > 0 , ≥ 0 , $= 0$, $\neq 0$.

Using MPL command **CALLS**, you can do a cancelable call. Use this command if the exit from the called function depends on conditions that may not be reached. In this case, using MPL command **ABORT** you can terminate the function execution and return to the next instruction after the call.

See also:

[Jumps and Function Calls – Related MPL Instructions and Data](#)

[MPL Description](#)

6.2.3.2.3. Jumps and Function Calls - Related MPL Instructions and Data

Instructions

GOTO label	Unconditional jump to the address indicated by the label.
GOTO value16	Unconditional jump to the address set in value16. Value16 is a 16-bit unsigned integer.
GOTO var16	Unconditional jump to the address indicated by var16. Var16 is a 16-bit MPL variable whose value is the jump address
GOTO label, var, cond	Conditional jump to the address indicated by the label. Var is a 16 or 32-bit MPL variable compared with 0. Test condition is: EQ, NEQ, GT, GEQ, LT, LEQ
GOTO value16, var, cond	Conditional jump to the address set in value16. Var is a 16 or 32-bit MPL variable compared with 0. Test condition is: EQ, NEQ, GT, GEQ, LT, LEQ
GOTO var16, var, cond	Conditional jump to the address indicated by var16. Var is a 16 or 32-bit MPL variable compared with 0. Test condition is: EQ, NEQ, GT, GEQ, LT, LEQ
CALL label	Unconditional call from the address indicated by the function starting label (i.e. function name)
CALL value16	Unconditional call from the address set in value16. Value16 is a 16-bit unsigned integer.
CALL var16	Unconditional call from the address indicated by var16. Var16 is a 16-bit MPL variable whose value is the MPL function address
CALL label, var, cond	Conditional call from the address indicated by the function starting label. Var is a 16 or 32-bit MPL variable compared with 0. Test condition is: EQ, NEQ, GT, GEQ, LT, LEQ
CALL value16, var, cond	Conditional call from the address set in value16. Var is a 16 or 32-bit MPL variable compared with 0. Test condition is: EQ, NEQ, GT, GEQ, LT, LEQ
CALL var16, var, cond	Conditional call from the address indicated by var16. Var is a 16 or 32-bit MPL variable compared with 0. Test condition is: EQ, NEQ, GT, GEQ, LT, LEQ
CALLS label	Cancelable call from the address indicated by the function starting label.
CALLS value16	Cancelable call from the address set in value16. Value16 is a 16-bit unsigned integer.
CALLS var16	Cancelable call from the address indicated by var16. Var16 is a 16-bit MPL variable whose value is the MPL function address
ABORT	Abort the execution of a MPL function called with CALLS
RET	Return from a MPL function

Remarks:

- All labels mentioned in the **GOTO** or **CALL** instructions must exist i.e. must be defined somewhere in the MPL program.
- The label values are assigned after MPL program compilation

- When you call a MPL function, the return address pointed by the IP (instruction pointer) is saved into the MPL stack. When **RET** is executed, the IP is set with the last value from the MPL stack, hence the MPL program execution continues with the next instruction after the call. The MPL stack dimension is 12 words. Each function call and MPL interrupt service routine call uses one word of the MPL stack
- The body of the MPL subroutines must be placed outside the main MPL program, for example, after the **END** instruction.

Programming Examples

```

GOTO label1, var1, LT; // jump to label1 if var1 < 0
GOTO label2, var1, LEQ; // jump to label2 if var1 <= 0
GOTO label3, var1, GT; // jump to label3 if var1 > 0
GOTO label4; // unconditional jump to label4
CALL fct1, var2, GEQ; // call function fct1, if var2 >= 0
CALL fct1, var2, EQ; // call function fct1, if var2 = 0
CALL fct1, var2, NEQ; // call function fct1, if var2 != 0
CALL fct1; // unconditional call of function fct1
CALLS fct2; // unconditional cancelable call of fct1
...
END; // end of main program

fct1:
...
...
RET;

fct2:
...
ABORT; // abort function, return to next MPL
// command after the CALLS

RET;

```

See also:

[Jumps and Function Calls - MPL Programming Details](#)

[MPL Description](#)

6.2.3.2.4. MPL Interrupts

In MPL, you can monitor simultaneously up to 13 conditions. Each condition triggers a MPL interrupt. When a MPL interrupt occurs, the normal MPL program execution is suspended to execute a MPL function associated with the interrupt, called the **interrupt service routine** (in short ISR). The MPL interrupt mechanism is the following:

- The programmable drive continuously monitors 12 conditions that may generate MPL interrupts. The motion controller has an additional condition that triggers the MPL interrupt **Int12** when an error on the slaves occurs
- When an interrupt condition occurs, a flag (bit) is set in the **ISR** (Interrupt Status Register)
- If the interrupt is enabled e.g. the same bit (as position) is set in the **ICR** (Interrupt Control Register) and also if the interrupts are globally enabled (**EINT** instruction was executed), the interrupt condition is qualified and it generates a MPL interrupt
- The interrupt causes a jump to the associated interrupt service routine. On entry in this routine, the MPL interrupts are globally disabled (**DINT**) and the interrupt flag is reset
- The interrupt service routine must end with the MPL instruction **RETI**, which returns to normal program execution and in the same time globally enables the MPL interrupts

The 13 monitored conditions are:

1. **Int0 – Enable input has changed:** either transition of the Enable input sets the interrupt flag
2. **Int1 – Short-circuit:** when the drive/motor hardware protection for short-circuit is triggered
3. **Int2 – Software protections:** when any of the following protections is triggered:
 - a. Over current
 - b. I2t motor
 - c. I2t drive
 - d. Over temperature motor
 - e. Over temperature drive
 - f. Over voltage
 - g. Under voltage
4. **Int3 – Control error:** when the control error protection is triggered
5. **Int4 – Communication error:** when a communication error occurs
6. **Int5 – Wrap around:** when the target position computed by the reference generator wraps around because it bypasses the limit of the 32-bit long integer representation
7. **Int6 – LSP programmed transition detected:** when the programmed transition (low to high or high to low) is detected on the limit switch input for positive direction (LSP)
8. **Int7 – LSN programmed transition detected:** when the programmed transition (low to high or high to low) is detected on the limit switch input for negative direction (LSN)
9. **Int8 – Capture input transition detected:** when the programmed transition (low to high or high to low) is detected on the 1st capture / encoder index input or on the 2nd capture / encoder index input

-
10. **Int9 – Motion is completed:** in position control, when *motion complete* condition is set and in speed control when target speed reaches zero.
 11. **Int10 – Time period has elapsed:** periodic time interrupt with a programmable time period set in the MPL parameter **TMLINTPER**
 12. **Int11 – Event set has occurred:** when last defined event has been occurred
 13. **Int12 – Error on slave occurred:** when a slave reports an error.

The interrupt service routines (ISR) of the MPL interrupts are similar with the MPL functions: the starting point is a label and the ending point is the MPL instruction **RETI** (return from interrupt). When a MPL interrupt occurs, the MPL instruction pointer (IP) jumps to the start address of the associated ISR. This information is read from an *interrupt table*, which contains the values of the starting labels for all the ISR. The beginning of the interrupt table is pointed by the MPL parameter **INTTABLE**. Like the MPL functions, the interrupt table and the interrupt service routines must be positioned outside the main section of the MPL program (see the programming example below).

At power-on, each drive/motor starts with a built-in interrupt table and a set of default ISR. The MPL interrupts are globally enabled together with the first 4 interrupts: **Int 0** to **Int 3**. For **Int 2**, all the protections are activated, except over temperature motor, which depends on the presence or not of a temperature sensor on the motor; hence this protection may or may not be activated. For each of these 4 interrupts there is a default ISR which is executed when the corresponding interrupt occurs.

***Remark:** A basic description of these defaults ISR is presented below. Their exact content is product dependent and can be seen using MPL development platforms like MotionPRO Developer which include the possibility to view and/or modify the contents of the default ISR for each type of drive/motor.*

If you intend to enable other MPL interrupts or to modify the default ISR for the first 4 MPL interrupts, you need to create another MPL interrupt table which will point towards your own ISR. In this new interrupt table, put the starting labels for your ISR and use the global symbols: **default_intx (x=0 to 11)** as labels for those ISR you don't want to change. These global symbols contain the start addresses of the default ISR.

***Remark:** Some of the drive/motor protections may not work properly if the MPL Interrupts are handled incorrectly. In order to avoid this situation keep in mind the following rules:*

- *The MPL interrupts must be kept globally enabled to allow execution of the ISR for those MPL interrupts triggered by protections. As during a MPL interrupt execution, the MPL interrupts are globally disabled, you should keep the ISR as short as possible, without waiting loops. If this is not possible, you must globally enable the interrupts with **EINT** command during your ISR execution.*
- *If you modify the interrupt service routines for **Int 0** to **Int 4**, make sure that you keep the original MPL commands from the default ISR. Put in other words, you may add your own commands, but these should not interfere with the original MPL commands. Moreover, the original MPL commands must be present in all the ISR execution paths.*

The interrupt flags are set independently of the activation or not of the MPL interrupts. Therefore, as a general rule, before enabling an interrupt, reset the corresponding flag. This operation will avoid triggering an interrupt immediately after activation, due to an interrupt flag set in the past.

To summarize, in order to work with a MPL interrupt, you need to:

- Edit your own ISR or decide to use the default ISR.
- Create your own interrupt table, and set the MPL parameter **INTTABLE** equal with your interrupt table start address. Exception: if you use only default ISR
- Reset the interrupt flag to avoid entering in an interrupt due to a flag set in the past

- Enable the MPL interrupt. As the MPL interrupts must be globally enabled, the MPL interrupt is now activated and your ISR will execute when the interrupt flag will be set.

Default ISR Description

ISR for Int0 – Enable input has changed: When enable input goes from disable to enable status, executes AXISON if ACR.1 = 1 (i.e. the drive/motor is set to start automatically after power-on with an external reference) or if ACR.3 = 1 (i.e. specific request to execute AXISON at recover from disable status). Before executing AXISON, if the drive/motor is set in electronic gearing slave mode, the motion mode is set again (followed by an update command – UPD) to force a re-initialization for smooth recoupling with the master.

ISR for Int1 – Short-circuit: Set Ready output (if present) to **not ready** status and turn off the green led (if present). Set Error output (if present) to **error** status and turn on the red led (if present). Execute AXISOFF and set SRL.3 =1 to set the drive/motor into the FAULT condition.

ISR for Int2 – Software protections: Same as ISR for Int1

ISR for Int3 – Control error: Same as ISR for Int1

See also:

[MPL Interrupts – Related MPL Instructions and Data](#)

[MPL Description](#)

6.2.3.2.5. MPL Interrupts - Related MPL Instructions and Data

Parameters

[INTTABLE](#) Pointer to the start address of the interrupt table

[TMLINTPER](#) Time period for the periodic time interrupt Int10. Measured in [time units](#)

Instructions

[EINT](#) Globally enables the MPL interrupts. Sets ICR.15 = 1

[DINT](#) Globally disables the MPL interrupts. Sets ICR.15 = 0

[SRB ICR, ANDm, ORm](#) Individually enable/disable MPL interrupts, by setting/resetting bits from ICR register according with AND mask **ANDm** and OR mask **ORm**

[SRB ISR, ANDm, 0;](#) Reset interrupt flags in the ISR register according with AND mask **ANDm**

[RETI](#) Return from a MPL interrupt service routine

Programming Example

Set MPL Int10 to generate a time interrupt at each 0.5s. In the ISR, switch the status of output #25/Ready to signal that the drive/motor is in standby. Leave the other MPL interrupts with their default ISR.

```
BEGIN;           // MPL program start

INTTABLE = InterruptTable; // set start address for the new interrupt table

ENDINIT;        // end of initialization

...

```

```

    int Ready_status; // Define integer variable Ready_status
    Ready_status = 0; // initialize Ready_status
    TMLINTPER = 500; //Set a time interrupt at every 0.5[s]
    SRB ICR, 0x8FFF, 0x0400; // Set ICR.10 to enable Int10
    ...
    END;                // end of the main section

InterruptTable:        // start of the interrupt table
    @default_int0;
    @default_int1;
    @default_int2;
    @default_int3;
    @default_int4;
    @default_int5;
    @default_int6;
    @default_int7;
    @default_int8;
    @default_int9;
    @int10;
    @default_int11;

int10:
    GOTO Turn_on, Ready_status, EQ; // Branch to Turn_on if Ready_status == 0
    SOUT#25;                        //Set Low to I/O line #25
    Ready_status = 0;               // set Ready_status = 0
    RETI;                            // return from interrupt

Turn_on:                          //Define label Turn_on
    ROUT#25;                        //Set High to I/O line #25
    Ready_status = 1;               // set Ready_status = 1
    RETI;

```

See also:

[MPL Interrupts – Related MPL Instructions and Data](#)
[MPL Description](#)

6.2.3.3. I/O Programming

6.2.3.3.1. General I/O (Firmware FAXx)

In MPL you can access up to 40 digital I/O lines, numbered: #0 to #39. Each programmable drive/motor has a specific number of inputs and outputs, therefore only a part of the 40 I/Os is used. The I/O numbering is common for all the products; hence each product has its own list of available I/Os. This is not an ordered list. For example, a product with 4 inputs and 4 outputs can use the inputs: #36, #37, #38 and #39 and the outputs #28, #29, #30 and #31.

Remark: *Read carefully the drive/motor user manual to find which I/O lines are available. Do not attempt to use the other I/Os. This may harm your drive/motor.*

Some [drives/motors](#) include I/O lines that may be used either as inputs or as outputs. In these cases, the same I/O number occurs both in the list of available inputs and in the list of available outputs. Before using these lines, you need to specify how you want to use them, with the MPL commands:

```
SETIO#n OUT;      //Set the I/O line #n as an output
SETIO#n IN;       //Set the I/O line #n as an input
```

Remarks:

- *Check the drive/motor user manual to find how are set, after power-on, the I/O lines that may be used either as inputs or as outputs*
- *You need to set an I/O line as input or output, only once, after power on*

You can read and save the status of an input with the MPL command:

```
user_var = IN#n; // read input #n in the user variable user_var
```

where `user_var` is a 16-bit integer variable and `n` is the input number. If the input line is low (0 logic), `user_var` is set to 0, else `user_var` is set to a non-zero value.

You can set an output high (1 logic) or low (0 logic) with the following commands:

```
ROUT#n;          // Set low the output line #n
SOUT#n;          // Set high the output line #n
```

Remark: *Check the drive/motor user manual to find if the I/O lines you are using are passed directly or are inverted inside the drive/motor. If an I/O line is inverted, you need to switch high with low in the examples above, which refer to the I/O line status at the drive/motor connector level.*

Using MPL command:

```
user_var = INPORT, 0xE00F; // read inputs in variable user_var
```

you can read simultaneously and save in a 16-bit integer variable the status of the following inputs:

- Enable input (#16/ENABLE) – saved in bit 15
- Limit switch input for negative direction (#24/LSN) - saved in bit 14
- Limit switch input for positive direction (#2/LSP) - saved in bit 13
- General-purpose inputs #39, #38, #37 and #36 – save din bits 3, 2, 1 and 0

The bits corresponding to these inputs are set as follows: 0 if the input is low and 1 if the input is high. The other bits of the variable are set to 0.

Remark: Each drive/motor contains in the MPL parameter **DIGIN_INVERSION_MASK** an inversion mask for these inputs. A bit set to 1 in this mask, means that the corresponding input is inverted. The value set in **user_var** is obtained after a logical XOR between the inputs status and the inversion mask. As result, the bits in **user_var** always show correctly the inputs status at connectors level (0 if the input is low and 1 if the input is high) even when the inputs are inverted.

Using MPL command:

```
    OUTPUT user_var;           // Send variable user_var to external output port
```

you can set simultaneously with the command value specified by a 16-bit integer variable, the following outputs:

- Ready output (#25/READY) – set by bit 15
- Error output (#12/ERROR) – set by bit 14
- General-purpose outputs: #31, #30, #29, #28 – set by bits 3, 2, 1, and 0

The outputs are set as follows: low if the corresponding bit in the variable is 0 and high if the corresponding bit in the variable is 1. The other bits of the variable are not used.

Remark: Each drive/motor contains in the MPL parameter **DIGOUT_INVERSION_MASK** an inversion mask for these outputs. A bit set to 1 in this mask, means that the corresponding output is inverted. The commands effectively sent to the outputs are obtained after a logical XOR between the **user_var** value and the inversion mask. As result, the outputs at connectors level always correspond to the **user_var** command values (low if the bit is 0 and high if the bit is 1), even when the outputs are inverted.

[General-purpose I/O – Related MPL Instructions and Data](#)

[MPL Description](#)

6.2.3.3.2. General I/O (Firmware F_{AXX}) - Related MPL Instructions and Data

Parameters

DIGIN INVERSION MASK Inversion mask for the following digital inputs:

- Enable input (#16/ENABLE) – bit 15
- Limit switch input for negative direction (#24/LSN) - bit 14
- Limit switch input for positive direction (#2/LSP) - bit 13
- General-purpose inputs #39, #38, #37 and #36 – bits 3, 2, 1 and 0

A bit set signals that the corresponding input is inverted. The MPL variable **INSTATUS** as well as the MPL command **INPORT** are considering this inversion mask to switch the status of inverted inputs. As result, in **INSTATUS** and in the MPL variable set by **INPORT**, the above bits always show correctly the inputs status at connectors level (0 if the input is low and 1 if the input is high) even when the inputs are inverted

DIGOUT INVERSION MASK Inversion mask for the following digital outputs:

- Ready output (#25/READY) – set by bit 15
- Error output (#12/ERROR) – set by bit 14
- General-purpose outputs: #31, #30, #29, #28 – set by bits 3, 2, 1, and 0

A bit set signals that the corresponding output is inverted. The MPL command **OUTPORT** uses this inversion mask to switch the command for the inverted outputs. As result, the outputs at connectors level are always set as specified by the above bits in the MPL variable of the **OUTPORT** command (low if the bit is 0 and high if the bit is 1), even when the outputs are inverted.

Variables

INSTATUS Provides status of the following digital inputs:

- Enable input (#16/ENABLE) – in bit 15
- Limit switch input for negative direction (#24/LSN) - in bit 14
- Limit switch input for positive direction (#2/LSP) - in bit 13
- General-purpose inputs #39, #38, #37 and #36 – in bits 3, 2, 1 and 0

The above bits are set to 0 if the input is low (at connectors level) and 1 if the input is high (at connectors level). The information is automatically corrected in the case of inverted inputs. The other bits **INSTATUS** have no significance.

Instructions

user_var = IN#n Read input #n in the user variable user_var

OUTPORTvalue16 Set simultaneously the output lines as specified by value16

ROUT#n Set low the output line #n

SOUT#n Set high the output line #n

SETIO#n OUT; Set the I/O line #n as an input

SETIO #n IN; Set the I/O line #n as an output

Programming Example

```
user_var = IN#36;            // read input #36 in user_var
GOTO label1, user_var, NEQ; // go to label1 if input #36 is 1
// input #36 is 0
user_var = IN#39;            // read input #39 in user_var
GOTO label2, user_var, EQ;  // go to label2 if input #39 is 0
// input #39 is 1
...
Label1:                      // input #36 is 1
...
Label2:                      // input #39 is 0
...
```

See also:

[General-purpose I/O – MPL Programming Details](#)

[MPL Description](#)

6.2.3.3.3. Special I/O (Firmware F_{Axx})

In MPL, there are 5 inputs and 2 outputs that have dedicated functions. These are:

- Enable input: #16/ENABLE
- 2 limit switch inputs: #2/LSP and #24/LSN
- 2 capture inputs: #5/CAPI and #34/2CAPI
- Ready output: #25/READY
- Error output: #12/ERROR

Remark: On some [drives/motors](#) only a part of these special I/O is available. When present, the capture and limit switch inputs are always connected to the same I/O numbers. However, the Enable input as well as the Ready and Error outputs may be assigned to other I/O lines. Their I/O number allocation is specific for each product.

The **enable** input is a safety input, and can be: active or inactive. On the active level, it enables normal operation. On the inactive level it disables the drive/motor similarly with the **AXISOFF** command. When the enable input goes from inactive to active level and **AXISON** command is automatically performed if **ACR.1 = 1** or **ACR.3 = 1**.

The active level is programmable: low or high via MPL parameter **DIGIN_ACTIVE_LEVEL** as follows:

- If **DIGIN_ACTIVE_LEVEL.15 = 1**, #16/ENABLE is active when the input is high
- If **DIGIN_ACTIVE_LEVEL.15 = 0**, #16/ENABLE is active when the input is low

Remark: The enable input high/low refers to the input level at drive/motor connector. After power on, the active level is set to enable normal operation with nothing connected on the input

The **limit switch inputs** main goal is to protect against accidental moves outside a defined working area. The protection involves connecting limit switches to:

- #2/LSP to stop movement in positive direction
- #24/LSN to stop movement in negative direction

A limit switch input can be: active or inactive. The active level is programmable: low or high via MPL parameter **DIGIN_ACTIVE_LEVEL** as follows:

- If **DIGIN_ACTIVE_LEVEL.14 = 1**, #24/LSN is active when the input is high
- If **DIGIN_ACTIVE_LEVEL.14 = 0**, #24/LSN is active when the input is low
- If **DIGIN_ACTIVE_LEVEL.13 = 1**, #2/LSP is active when the input is high
- If **DIGIN_ACTIVE_LEVEL.13 = 0**, #2/LSP is active when the input is low

Remark: The limit switch inputs high/low refers to the inputs level at drive/motor connector. After power on, the active level is set to have both limit switches inactive with nothing connected on these inputs

When positive limit switch #2/LSP input is active, movement is possible only in the negative direction. Any attempt to move in the positive direction will set the drive/motor in quick stop mode, and this will stop the move with the deceleration rate set in MPL parameters **CDEC**.

When negative limit switch #24/LSN input is active, movement is possible only in the positive direction. Any attempt to move in the negative direction will set the drive/motor in quick stop mode, and this will stop the move with the deceleration rate set in MPL parameters **CDEC**.

Remark: The drive/motor exits from quick stop mode only by setting a new motion mode.

The limit switch inputs may also be used as capture inputs due to their capability to sense low to high or high to low transitions and to capture the motor, load or master position when these transitions occur. The limit switch inputs capturing behavior is identical with that of the capture inputs #5/CAPI and #34/2CAPI and therefore is presented below together with the capture inputs.

You can set either an event or a [MPL interrupt](#), for each limit switch input, to detect when a programmed transition has occurred. In both cases you need to:

1. Enable limit switch input capability to detect a low->high or a high-> low transition with one of the following MPL instructions:

```
ENLSP0;      //Enable #2/LSP to detect a high->low transition
ENLSP1;      //Enable #2/LSP to detect a low->high transition
ENLSN0;      //Enable #24/LSN to detect a high->low transition
ENLSN1;      //Enable #24/LSN to detect a low->high transition
```

2. Set:

- A limit switch event with !LSP or !LSN, then wait until the event occurs with WAIT! ;, or
- Enable the LSP or LSN MPL interrupt with the MPL commands:

```
SRB ICR 0xFFFF,0x0040; // Set ICR.6 = 1 to enable LSP interrupt
SRB ICR 0xFFFF,0x0080; // Set ICR.7 = 1 to enable LSN interrupt
```

Remarks:

- The main task of the limit switches i.e. to protect against accidental moves outside the working area is performed independently of the fact that limit switches may be enabled or not to detect transitions
- A limit switch input capability to detect transitions is automatically disabled, after the programmed transition was detected. In order to reuse it, you need to enable it again.
- You may also disable a limit switch input capability to detect transitions, using the MPL commands: **DISLSP, DISLSN**

You can also use the limit switch inputs as general-purpose inputs by disabling their capability to protect against accidental moves outside a defined working area. For this you need to set MPL parameter **LSACTIVE** = 1. This command, doesn't affect the limit switch inputs capability to detect transitions.

Remark: After power on, **LSACTIVE** = 0 and the limit switches are active.

You can read the limit switches inputs, at any moment, independently of **LSACTIVE** value, like any other inputs using the MPL instructions:

```
var = IN#2;      // read status of the positive limit switch input
var = IN#24;     // read status of the negative limit switch input
```

The **capture inputs** are special inputs that can be programmed to sense either a low to high or high to low transition and capture the motor, load or master position with very high accuracy when these transitions occur.

Typically, the 1st encoder index is connected to the 1st capture input – #5/CAPI, and the 2nd encoder index is connected to the 2nd capture input – #34/2CAPI.

When an incremental encoder provides the motor position, its signals are always connected to the 1st encoder interface. When an incremental encoder provides the master position, its signals are always connected to the 2nd encoder interface. When an incremental encoder provides the load position, its signals are connected to:

- 2nd encoder interface, if there is another sensor on the motor (for example DC motor with encoder on load and tachometer on the motor)
- 1st encoder interface, if there is no other sensor on the motor (for example steppers controlled open-loop with an encoder on the load)

When the programmed transition occurs on any capture or limit switch input, the following happens:

- Motor position **APOS_MT** is captured and memorized in the MPL variable **CAPPOS**, except the case of open-loop systems, where reference position **TPOS** is captured instead
- Master position **APOS2** or load position **APOS_LD** is captured and memorized in the MPL variable **CAPPOS2**, except the case of steppers controlled open loop with an encoder on the load, when load position is captured in **CAPPOS**.

The selection between master and load position is done as follows: load position is saved in **CAPPOS2** only for the setup configurations which use different sensors for load and motor and foresee a transmission ratio between them. For all the other setup configurations, the master position is saved in **CAPPOS2**. The master position is automatically computed when pulse and direction signals or quadrature encoder signals are connected to their dedicated inputs.

When an incremental encoder is connected to the 1st encoder interface and 1st capture/encoder index detects the programmed transition, the position captured in **CAPPOS** is very accurate, being read in less than 200 ns after the input transition. The position captured in **CAPPOS2** is also accurate being read with a maximum delay of 5µs.

When an incremental encoder is connected to the 2nd encoder interface or when master position is set via pulse & direction signals and 2nd capture/encoder index detects the programmed transition, the position captured in **CAPPOS2** is very accurate, being read in less than 200 ns, after the input transition. The position captured in **CAPPOS2** is read with a maximum delay of 5µs.

When any of the 2 limit switch inputs detects the programmed transition, the positions captured in **CAPPOS** and **CAPPOS2** are accurate, both being read with a maximum delay of 5µs.

You can set either an event or a [MPL interrupt](#) on a capture input. In both cases you need to:

1. Enable the capture input for the detection of a low->high or a high-> low transition with one of the following MPL instructions:

```
ENCAP10; //Enable #5/CAP1 to detect a high->low transition
ENCAP11; //Enable #5/CAP1 to detect a low->high transition
EN2CAP10; //Enable #34/2CAP1 to detect a high->low transition
EN2CAP11; //Enable #34/2CAP1 to detect a low->high transition
```

2. Set:

- A capture event with `!CAP`, then wait until the event occurs with `WAIT!;`, or
- Enable the MPL capture interrupt with the MPL command:

```
SRB ICR 0xFFFF,0x0100; //Set ICR.8 = 1
```

Remarks:

- *If both capture inputs are activated in the same time, the capture event and the MPL capture interrupt flag is set by the capture input that is triggered first. The capture event or the MPL capture interrupt makes no difference between the two capture inputs.*
- *When the programmed transition is detected, the capture input is automatically disabled. In order to reuse it, you need to enable it again for the desired transition*
- *You may also disable a capture input (i.e. its capability to detect a programmed transition) previously enabled, using the MPL commands: **DISCAPI**, **DIS2CAPI***

See also:

[Special I/O – Related MPL Instructions and Data](#)

[MPL Description](#)

6.2.3.3.4. Special I/O (Firmware F_{Axx}) - Related MPL Instructions and Data

Parameters

[DIGIN ACTIVE LEVEL](#) Sets active levels for enable and limit switch inputs as follows:

- Enable input (#16/ENABLE) – bit 15: 0 – low, 1 – high
- Limit switch input for negative direction (#24/LSN) - on bit 14: 0 – low, 1 – high
- Limit switch input for positive direction (#2/LSP) - on bit 13: 0 – low, 1 – high

[LSACTIVE](#) When set to a non-zero value, disables limit switch inputs capability to protect against accidental moves outside a defined working area. In this case, the limit switch inputs are treated like 2 extra general-purpose inputs

[CDEC](#) Command deceleration for quick stop mode. Measured in [acceleration units](#)

[ICR](#) Interrupt Control Register. The MPL interrupts can be enabled or disabled by setting or resetting the corresponding bits from this register

[ACR](#) Auxiliary Control Register. If **ACR.1** = 1 the drive/motor is set to start automatically after power-on with an external reference. If **ACR.3** = 1 there is a specific request to execute **AXISON** at recover from disable status. In both cases, an **AXISON** is executed when **enable** input goes from inactive to active status.

Variables

[CAPPOS](#) Position captured when programmed transition occurs on 1st capture/encoder index input. Measured in [motor position units](#), except the case of stepper motors, when it is measured in [position units](#)

[CAPPOS2](#) Position captured when programmed transition occurs on 2nd capture/encoder index input. Measured in [position units](#) when load position is captured, or in [master position units](#) when master position is captured

[APOS2](#) Master position computed by the slaves from pulse & direction or quadrature encoder inputs. Measured in [master position units](#)

<u>TPOS</u>	Target position – position reference computed by the reference generator at each slow loop sampling period. Measured in <u>position units</u>
<u>APOS_LD</u>	Actual load position. Measured in <u>position units</u> . Alternate name: <u>APOS</u>
<u>APOS_MT</u>	Actual motor position. Measured in <u>motor position units</u> .
Instructions	
<u>ICAP</u>	Set event on capture inputs
<u>ENCAP10</u>	Enable 1st capture/encoder index input to detect a high to low transition
<u>EN2CAP10</u>	Enable 2nd capture/encoder index input to detect a high to low transition
<u>ENCAP11</u>	Enable 1st capture/encoder index input to detect a low to high transition
<u>EN2CAP11</u>	Enable 2nd capture/encoder index input to detect a low to high transition
<u>ILSN</u>	Set event on negative limit switch input
<u>ILSP</u>	Set event on positive limit switch input
<u>ENLSP0</u>	Enable positive limit switch input to detect a high to low transition
<u>ENLSN0</u>	Enable negative limit switch input to detect a high to low transition
<u>ENLSP1</u>	Enable positive limit switch input to detect a low to high transition
<u>ENLSN1</u>	Enable negative limit switch input to detect a low to high transition
<u>DISCAP1</u>	Disable 1st capture/encoder index input to detect transitions
<u>DIS2CAP1</u>	Disable 2nd capture/encoder index input to detect transitions
<u>DISLSP</u>	Disable positive limit switch input to detect transitions
<u>DISLSN</u>	Disable negative limit switch input to detect transitions
<u>UPD!</u>	Update the motion mode and/or the motion parameters when the programmed event occurs
<u>STOP!</u>	Stop motion with the acceleration/deceleration set in CACC , when the programmed event occurs
<u>WAIT!</u> value16	Wait until the programmed event occurs. If the command is followed by value16 , the wait ends after the time interval specified in this 16-bit integer value. Value16 is measured in <u>time units</u>
<u>SRB</u>	Set/reset bits from a MPL data

Programming Example

```
//Stop motion on next encoder index
ENCAP11; //Set event: When the encoder index goes low->high
!CAP;
STOP!;//Stop the motion when event occurs
WAIT!;//Wait until the event occurs
CPOS = CAPPOS; // new command position = captured position
CPA; //position command is absolute
MODE PP;
TUM1; //set Target Update Mode 1
UPD; //execute immediate
!MC; WAIT!; //wait for completion
```

See also:

[Special I/O – MPL Programming Details](#)

[MPL Description](#)

6.2.3.3.5. General-purpose I/O (Firmware FBxx)

In MPL you can access up to 16 digital input and 16 digital output lines, numbered: 0 to 15. Each programmable drive/motor has a specific number of inputs and outputs, therefore only a part of the 16 inputs or 16 outputs is used. The I/O numbering is common for all the products; hence each product has its own list of available I/Os. This is an ordered list. For example, a product with 4 inputs and 4 outputs can use the inputs: 0, 1, 2 and 3 and the outputs 0, 1, 2 and 3.

Remark: *Read carefully the drive/motor user manual to find which I/O lines are available. Do not attempt to use the other I/Os. This may harm your drive/motor.*

Some [drives/motors](#) include I/O lines that may be used either as inputs or as outputs. Before using these lines, you need to specify how you want to use them, with the MPL commands:

```
SetAsInput(n); //Set the IO line n as an input
SetAsOutput(n); //Set the IO line n as an output
```

Remarks:

- *Check the drive/motor user manual to find how are set, after power-on, the I/O lines that may be used either as inputs or as outputs*
- *You need to set an I/O line as input or output, only once, after power on*

You can read and save the status of an input with the MPL command:

```
user_var = IN(n); //Read IO line n data into variable user_var
```

where `user_var` is a 16-bit integer variable and `n` is the input number. If the input line is low (0 logic), `user_var` is set to 0, else `user_var` is set to a non-zero value.

You can set an output high (1 logic) or low (0 logic) with the following commands:

```
OUT(n)=value16; // Set IO line n according with its
corresponding bit from value16
```

Remark: Check the drive/motor user manual to find if the I/O lines you are using are passed directly or are inverted inside the drive/motor. If an I/O line is inverted, you need to switch high with low in the examples above, which refer to the I/O line status at the drive/motor connector level.

Using MPL command:

```
user_var = IN(n1,n2,n3,...); // Set corresponding bits from a according
with selected inputs status
```

you can read simultaneously and save in a 16-bit integer variable the status of the selected inputs.

The bits corresponding to these inputs are set as follows: 0 if the input is low and 1 if the input is high. The other bits of the variable are set to 0.

Remark: Each drive/motor contains in the MPL parameter **DIGIN_INVERSION_MASK** an inversion mask for these inputs. A bit set to 1 in this mask, means that the corresponding input is inverted. The value set in **user_var** is obtained after a logical XOR between the inputs status and the inversion mask. As result, the bits in **user_var** always show correctly the inputs status at connectors level (0 if the input is low and 1 if the input is high) even when the inputs are inverted.

Using MPL command:

```
OUT(n1, n2, n3,...) = value16; // Set outputs n1, n2, n3, ... according
with corresponding bits from value16
```

you can set simultaneously with the command value specified by a 16-bit integer variable, the selected outputs.

The outputs are set as follows: low if the corresponding bit in the variable is 0 and high if the corresponding bit in the variable is 1. The other bits of the variable are not used.

Remark: Each drive/motor contains in the MPL parameter **DIGOUT_INVERSION_MASK** an inversion mask for these outputs. A bit set to 1 in this mask, means that the corresponding output is inverted. The commands effectively sent to the outputs are obtained after a logical XOR between the immediate or **user_var** value and the inversion mask. As result, the outputs at connectors level always correspond to the immediate or **user_var** command values (low if the bit is 0 and high if the bit is 1), even when the outputs are inverted.

[General-purpose I/O – Related MPL Instructions and Data](#)

[MPL Description](#)

6.2.3.3.6. General-purpose I/O – Related MPL Instructions and Data (Firmware FBxx)

Variables

INSTATUS Provides status of the following digital inputs:

The above bits are set to 0 if the input is low (at connectors level) and 1 if the input is high (at connectors level). The information is automatically corrected in the case of inverted inputs. The other bits **INSTATUS** have no significance.

Instructions

user_var = IN(n) Read input **n** in the user variable `user_var`

user_var = IN(n1, n2, n3, ...) Read inputs **n1, n2, n3,...** in the user variable `user_var`

OUT(n) =value16 Set the output line as specified by `value16`

OUT(n1, n2, n3, ...) =value16 Set the output lines `n1 n2, n3` as specified by `value16`

SetAsInput(n); Set the I/O line `#n` as an input

SetAsOutput(n); Set the I/O line `#n` as an output

Programming Example

```
user_var = IN#36;           // read input #36 in user_var
GOTO label1, user_var, NEQ; // go to label1 if input #36 is 1
// input #36 is 0
user_var = IN#39;           // read input #39 in user_var
GOTO label2, user_var, EQ;  // go to label2 if input #39 is 0
// input #39 is 1
...
Label1:                     // input #36 is 1
...
Label2:                     // input #39 is 0
...
```

See also:

[General-purpose I/O – MPL Programming Details](#)

[MPL Description](#)

6.2.3.3.7. Special I/O - MPL Programming Details (Firmware FBxx)

In MPL, there are 5 inputs and 2 outputs that have dedicated functions. These are:

- Enable input
- 2 limit switch inputs
- 2 capture inputs
- Ready output
- Error output

Remark: On some [drives/motors](#) only a part of these special I/O is available. When present, the capture and limit switch inputs are always connected to the same I/O numbers. However, the Enable input as well as the Ready and Error outputs may be assigned to other I/O lines. Their I/O number allocation is specific for each product.

The **enable** input is a safety input, and can be: active or inactive. On the active level, it enables normal operation. On the inactive level it disables the drive/motor similarly with the **AXISOFF** command. When the enable input goes from inactive to active level and **AXISON** command is automatically performed if **ACR.1 = 1** or **ACR.3 = 1**.

The active level is programmable: low or high via MPL parameter **DIGIN_ACTIVE_LEVEL** as follows:

- If **DIGIN_ACTIVE_LEVEL.15 = 1**, Enable is active when the input is high
- If **DIGIN_ACTIVE_LEVEL.15 = 0**, Enable is active when the input is low

Remark: The enable input high/low refers to the input level at drive/motor connector. After power on, the active level is set to enable normal operation with nothing connected on the input

The **limit switch inputs** main goal is to protect against accidental moves outside a defined working area. The protection involves connecting limit switches to:

- LSP input to stop movement in positive direction
- LSN input to stop movement in negative direction

A limit switch input can be: active or inactive. The active level is programmable: low or high via MPL parameter **DIGIN_ACTIVE_LEVEL** as follows:

- If **DIGIN_ACTIVE_LEVEL.14 = 1**, Limit Switch Negative is active when the input is high
- If **DIGIN_ACTIVE_LEVEL.14 = 0**, Limit Switch Negative is active when the input is low
- If **DIGIN_ACTIVE_LEVEL.13 = 1**, Limit Switch Positive is active when the input is high
- If **DIGIN_ACTIVE_LEVEL.13 = 0**, Limit Switch Positive is active when the input is low

Remark: The limit switch inputs high/low refers to the inputs level at drive/motor connector. After power on, the active level is set to have both limit switches inactive with nothing connected on these inputs

When positive limit switch input is active, movement is possible only in the negative direction. Any attempt to move in the positive direction will set the drive/motor in quick stop mode, and this will stop the move with the deceleration rate set in MPL parameters **CDEC**.

When negative limit switch input is active, movement is possible only in the positive direction. Any attempt to move in the negative direction will set the drive/motor in quick stop mode, and this will stop the move with the deceleration rate set in MPL parameters **CDEC**.

Remark: The drive/motor exits from quick stop mode only by setting a new motion mode.

The limit switch inputs may also be used as capture inputs due to their capability to sense low to high or high to low transitions and to capture the motor, load or master position when these transitions occur. The limit switch inputs capturing behavior is identical with that of the capture inputs and therefore is presented below together with the capture inputs.

You can set either an event or a [MPL interrupt](#), for each limit switch input, to detect when a programmed transition has occurred. In both cases you need to:

1. Enable limit switch input capability to detect a low->high or a high-> low transition with one of the following MPL instructions:

```
ENLSP0; //Enable Positive Limit Switch to detect a high->low
transition
ENLSP1; //Enable Positive Limit Switch to detect a low->high
transition
ENLSN0; //Enable Negative Limit Switch to detect a high->low
transition
ENLSN1; //Enable Negative Limit Switch to detect a low->high
transition
```

2. Set:

- A limit switch event with !LSP or !LSN, then wait until the event occurs with WAIT! ;, or
- Enable the LSP or LSN MPL interrupt with the MPL commands:

```
SRB ICR 0xFFFF,0x0040; //Set/Reset Bits of Interrupt Control Register
SRB ICR 0xFFFF,0x0080; //Set/Reset Bits of Interrupt Control Register
```

Remarks:

- The main task of the limit switches i.e. to protect against accidental moves outside the working area is performed independently of the fact that limit switches may be enabled or not to detect transitions
- A limit switch input capability to detect transitions is automatically disabled, after the programmed transition was detected. In order to reuse it, you need to enable it again.
- You may also disable a limit switch input capability to detect transitions, using the MPL commands: **DISLSP, DISLSN**

You can also use the limit switch inputs as general-purpose inputs by disabling their capability to protect against accidental moves outside a defined working area. For this you need to set MPL parameter **LSACTIVE** = 1. This command, doesn't affect the limit switch inputs capability to detect transitions.

Remark: After power on, **LSACTIVE** = 0 and the limit switches are active.

You can read the limit switches inputs, at any moment, independently of **LSACTIVE** value, like any other inputs using the MPL instructions:

```
var = IN#2; // read status of the positive limit switch input
var = IN#24; // read status of the negative limit switch input
```

The **capture inputs** are special inputs that can be programmed to sense either a low to high or high to low transition and capture the motor, load or master position with very high accuracy when these transitions occur.

Typically, the 1st encoder index is connected to the 1st capture input – #5/CAP1, and the 2nd encoder index is connected to the 2nd capture input – #34/2CAP1.

When an incremental encoder provides the motor position, its signals are always connected to the 1st encoder interface. When an incremental encoder provides the master position, its signals are always connected to the 2nd encoder interface. When an incremental encoder provides the load position, its signals are connected to:

- 2nd encoder interface, if there is another sensor on the motor (for example DC motor with encoder on load and tachometer on the motor)
- 1st encoder interface, if there is no other sensor on the motor (for example steppers controlled open-loop with an encoder on the load)

When the programmed transition occurs on any capture or limit switch input, the following happens:

- Motor position **APOS_MT** is captured and memorized in the MPL variable **CAPPOS**, except the case of open-loop systems, where reference position **TPOS** is captured instead
- Master position **APOS2** or load position **APOS_LD** is captured and memorized in the MPL variable **CAPPOS2**, except the case of steppers controlled open loop with an encoder on the load, when load position is captured in **CAPPOS**.

The selection between master and load position is done as follows: load position is saved in **CAPPOS2** only for the setup configurations which use different sensors for load and motor and foresee a transmission ratio between them. For all the other setup configurations, the master position is saved in **CAPPOS2**. The master position is automatically computed when pulse and direction signals or quadrature encoder signals are connected to their dedicated inputs.

When an incremental encoder is connected to the 1st encoder interface and 1st capture/encoder index detects the programmed transition, the position captured in **CAPPOS** is very accurate, being read in less than 200 ns after the input transition. The position captured in **CAPPOS2** is also accurate being read with a maximum delay of 5µs.

When an incremental encoder is connected to the 2nd encoder interface or when master position is set via pulse & direction signals and 2nd capture/encoder index detects the programmed transition, the position captured in **CAPPOS2** is very accurate, being read in less than 200 ns, after the input transition. The position captured in **CAPPOS2** is read with a maximum delay of 5µs.

When any of the 2 limit switch inputs detects the programmed transition, the positions captured in **CAPPOS** and **CAPPOS2** are accurate, both being read with a maximum delay of 5µs.

You can set either an event or a [MPL interrupt](#) on a capture input. In both cases you need to:

1. Enable the capture input for the detection of a low->high or a high-> low transition with one of the following MPL instructions:

```
ENCAP10;    //Activate CAPI input to trigger a rising transitions
ENCAP11;    //Activate CAPI input to trigger a falling transitions
EN2CAP10;   //Activate CAPI input to trigger a rising transitions
EN2CAP11;   //Activate CAPI input to trigger a falling transitions
```

2. Set:

- A capture event with !CAP, then wait until the event occurs with WAIT! ;, or
- Enable the MPL capture interrupt with the MPL command:

```
SRB ICR 0xFFFF,0x0100; //Set/Reset Bits of Interrupt Control Register
```

Remarks:

- *If both capture inputs are activated in the same time, the capture event and the MPL capture interrupt flag is set by the capture input that is triggered first. The capture event or the MPL capture interrupt makes no difference between the two capture inputs.*
- *When the programmed transition is detected, the capture input is automatically disabled. In order to reuse it, you need to enable it again for the desired transition*
- *You may also disable a capture input (i.e. its capability to detect a programmed transition) previously enabled, using the MPL commands: **DISCAPI**, **DIS2CAPI***

See also:

[Special I/O – Related MPL Instructions and Data](#)

[MPL Description](#)

6.2.3.3.8. Special I/O - Related MPL Instructions and Data (Firmware FBxx)

Parameters

DIGIN ACTIVE LEVEL Sets active levels for enable and limit switch inputs as follows:

- Enable input – on bit 15: 0 – low, 1 – high
- Limit switch input for negative direction – on bit 14: 0 – low, 1 – high
- Limit switch input for positive direction – on bit 13: 0 – low, 1 – high

LSACTIVE When set to a non-zero value, disables limit switch inputs capability to protect against accidental moves outside a defined working area. In this case, the limit switch inputs are treated like 2 extra general-purpose inputs

CDEC Command deceleration for quick stop mode. Measured in [acceleration units](#)

ICR Interrupt Control Register. The MPL interrupts can be enabled or disabled by setting or resetting the corresponding bits from this register

ACR Auxiliary Control Register. If **ACR.1** = 1 the drive/motor is set to start automatically after power-on with an external reference. If **ACR.3** = 1 there is a specific request to execute **AXISON** at recover from disable status. In both cases, an **AXISON** is executed when **enable** input goes from inactive to active status.

Variables

CAPPOS Position captured when programmed transition occurs on 1st capture/encoder index input. Measured in [motor position units](#), except the case of stepper motors, when it is measured in [position units](#)

CAPPOS2 Position captured when programmed transition occurs on 2nd capture/encoder index input. Measured in [position units](#) when load position is captured, or in [master position units](#) when master position is captured

APOS2 Master position computed by the slaves from pulse & direction or quadrature encoder inputs. Measured in [master position units](#)

TPOS Target position – position reference computed by the reference generator at each slow loop sampling period. Measured in [position units](#)

APOS_LD Actual load position. Measured in [position units](#). Alternate name: **APOS**

APOS_MT Actual motor position. Measured in [motor position units](#).

Instructions

ICAP Set event on capture inputs

ENCAP10 Enable 1st capture/encoder index input to detect a high to low transition

EN2CAP10 Enable 2nd capture/encoder index input to detect a high to low transition

ENCAP11 Enable 1st capture/encoder index input to detect a low to high transition

EN2CAP11 Enable 2nd capture/encoder index input to detect a low to high transition

ILSN Set event on negative limit switch input

ILSP Set event on positive limit switch input

ENLSP0 Enable positive limit switch input to detect a high to low transition

ENLSN0 Enable negative limit switch input to detect a high to low transition

ENLSP1 Enable positive limit switch input to detect a low to high transition

ENLSN1 Enable negative limit switch input to detect a low to high transition

DISCAP1 Disable 1st capture/encoder index input to detect transitions

DIS2CAP1 Disable 2nd capture/encoder index input to detect transitions

DISLSP Disable positive limit switch input to detect transitions

DISLSN Disable negative limit switch input to detect transitions

UPD! Update the motion mode and/or the motion parameters when the programmed event occurs

STOP! Stop motion with the acceleration/deceleration set in **CACC**, when the programmed event occurs

WAIT! value16 Wait until the programmed event occurs. If the command is followed by **value16**, the wait ends after the time interval specified in this 16-bit integer value. Value16 is measured in [time units](#)

SRB Set/reset bits from a MPL data

Programming Example

```
//Stop motion on next encoder index
ENCAP11; //Set event: When the encoder index goes low->high
!CAP;
STOP!;//Stop the motion when event occurs
WAIT!;//Wait until the event occurs
CPOS = CAPPOS; // new command position = captured position
CPA; //position command is absolute
MODE PP;
TUM1; //set Target Update Mode 1
UPD; //execute immediate
!MC; WAIT!; //wait for completion
```

See also:

[Special I/O – MPL Programming Details](#)

[MPL Description](#)

6.2.3.4. Assignment and Data Transfer

6.2.3.4.1. Setup 16-bit variable

The MPL instructions presented in this paragraph show you the options you have to:

1. Assign a value to a [16-bit integer MPL data](#)
2. Transfer in a memory location, a 16-bit value or the value of a 16-bit integer MPL data

In the first case, the destination is a 16-bit MPL data: MPL register, MPL parameter or user variable and the source can be:

- A 16-bit immediate value or a label
- A 16-bit MPL data: MPL register, parameter, variable or user variable (direct or negated)
- The high or low part of a 32-bit MPL data: MPL parameter, variable or user variable
- A memory location indicated through a pointer variable
- The result of the checksum performed with all locations situated between 2 memory addresses specified either as immediate values or via 2 pointer variables.

In the second case and the destination is a memory location indicated through a pointer variable and the source can be:

- A 16-bit immediate value
- A 16-bit MPL data: MPL register, parameter, variable or user variable

Programming Examples

1) Source: 16-bit immediate value, Destination: 16-bit MPL data. The immediate value can be decimal or hexadecimal

```
user_var = 100; // set user variable user_var with value 100
user_var = 0x100; // set user variable user_var with value 0x100 (256)
label1:
user_var = label1; // set user variable user_var with label1 value
```

2) Source: 16-bit MPL data, Destination: 16-bit MPL data.

```
var_dest = var_source; // copy value of var_source in var_dest
var_dest = -var_source; // copy negate value of var_source in var_dest
```

3) Source: high or low part of a 32-bit MPL data, Destination: 16-bit MPL data. The 32-bit MPL data can be either long or fixed

```
int_var = long_var(L); // copy low part of long_var in int_var
int_var = fixed_var(H); // copy high part of fixed_var in int_var
```

4) Source: a memory location indicated through a pointer variable, Destination: 16-bit MPL data. The memory location can be of 3 types: RAM for data (dm), RAM for MPL programs (pm), EEPROM SPI-connected for MPL programs (spi). If the pointer variable is followed by a + sign, after the assignment, the pointer variable is incremented by 1

```
p_var = 0x4500; // set 0x4500 in pointer variable p_var
var1 = (p_var),spi; // var1 = value of the EEPROM memory location 0x4500
var1 = (p_var+),spi; // var1 = value of the EEPROM memory location 0x4500
// p_var = 0x4501
p_var = 0x8200; // set 0x8200 in pointer variable p_var
var1 = (p_var),pm; // var1 = value of the RAM memory location 0x8200 for
//MPL programs
var1 = (p_var+),pm; // var1 = value of the RAM memory location 0x8200 for
//MPL programs, then set p_var = 0x8201
p_var = 0xA00; // set 0xA00 in pointer variable p_var
var1 = (p_var),dm; // var1 = value of the RAM memory location 0xA00 for
//MPL data
var1 = (p_var+),dm; // var1 = value of the RAM memory location 0xA00 for
//MPL data, then set p_var = 0xA01
```

5) Source: the result of the checksum. Destination: 16-bit MPL data. The checksum is performed with all locations situated between 2 memory addresses. These are specified either as immediate values or via 2 pointer variables. The memory can be of 3 types: RAM for data (dm), RAM for MPL programs (pm), EEPROM SPI-connected for MPL programs (spi).

```
checksum, spi 0x4000, 0x4500, var1; // var1=checksum value computed
// between EEPROM memory addresses 0x4000 and 0x4500
```

```

start = 0x9000;          // set start address = 0x9000
end = 0x9100;           // set end address = 0x9100
checksum, pm start, stop, var1;    // var1=checksum value computed
// between RAM (for MPL programs) addresses 0x9000 and 0x9100 pointed by the MPL
// variables start and stop

```

6) Source: 16-bit immediate value (decimal or hexadecimal) or 16-bit MPL data. Destination: a memory location indicated through a pointer variable. The memory location can be of 3 types: RAM for data (dm), RAM for MPL programs (pm), EEPROM SPI-connected for MPL programs (spi). If the pointer variable is followed by a + sign, after the assignment, the pointer variable is incremented by 1

```

p_var = 0x4500;          // set 0x4500 in pointer variable p_var
(p_var), spi = -5;      // write value -5 in the EEPROM memory location
                        // 0x4500
(p_var+), spi = var1;   // write var1 value in the EEPROM memory location
                        // 0x4500, then set p_var = 0x4501
p_var = 0x8200;          // set 0x8200 in pointer variable p_var
(p_var), pm = 0x10;     // write value 0x10 in RAM memory location 0x8200 for
// MPL programs
(p_var+), pm = var1;    // write var1 value in RAM memory location 0x8200 for
                        // MPL programs, then set p_var = 0x8201
p_var = 0xA00;           // set 0xA00 in pointer variable p_var
(p_var), dm = 50;       // write value 50 in the RAM memory location 0xA00 for
                        // MPL data
(p_var+), dm = var1;    // write var1 value in the RAM memory location 0xA00
                        // for MPL data, then set p_var = 0xA01

```

Remark: The MPL assignment instructions with source an immediate value or a MPL data and destination a MPL data, use a **short address** format for the destination. The short address format requires a destination address between 0x200 and 0x3FF or between 0x800 and 0x9FF. This restriction is respected now by all the predefined or user-defined MPL data, hence you can use the above assignment instructions without checking the variables addresses.

However, considering possible future developments, the MPL also includes assignment instructions using a **full address** format where the destination address can be any 16-bit value. The following commands support full addressing:

```

int_var, dm = 100;      // set int_var = 100 using full addressing
int_var, dm = 0x100;    // set int_var = 0x100(256) using full addressing
var_dest, dm = var_source; // copy value of var_source in var_dest using
                        // full addressing

```

See also:

[Assignment and Data Transfer. 32-bit data – MPL Programming Details](#)

[MPL Description](#)

6.2.3.4.2. Setup 32-bit variable

The MPL instructions presented in this paragraph show you the options you have to:

1. Assign a value to a [32-bit long or fixed MPL data](#)
2. Assign a value to the high (16MSB) or low (16LSB) part of a 32-bit long or fixed data
3. Transfer in 2 consecutive memory locations, a 32-bit value or the value of a 32-bit long or fixed MPL data

In the first case, the destination is a 32-bit MPL data: MPL parameter or user variable and the source can be:

- A 32-bit immediate value
- A 32-bit MPL data: MPL register, parameter, variable or user variable (direct or negated)
- A 16-bit MPL data left shifted by 0 to 16
- 2 consecutive memory locations, indicated through a pointer variable

In the second case, the destination is the high or low part of a 32-bit MPL data: MPL parameter or user variable and the source can be:

- A 16-bit immediate value
- A 16-bit MPL data: MPL register, parameter, variable or user variable

In the third case, the destination is 2 consecutive memory locations, indicated through a pointer variable and the source can be:

- A 32-bit immediate value
- A 32-bit MPL data: MPL parameter, variable or user variable

Programming Programmable Examples

1) Source: 32-bit immediate value, Destination: 32-bit MPL data. The immediate value can be decimal or hexadecimal. The destination can be either a long or a fixed variable

```
long_var = 100000;      // set user variable long_var with value 100000
long_var = 0x100000;    // set user variable long_var with value 0x100000
fixed_var = 1.5;       // set user variable fixed_var with value 1.5 (0x18000)
fixed_var = 0x14000;   // set user variable fixed_var with value 1.25 (0x14000)
```

2) Source: 32-bit MPL data, Destination: 32-bit MPL data.

```
var_dest = var_source; // copy value of var_source in var_dest
var_dest = -var_source; // copy negate value of var_source in var_dest
```

Remark: source and destination must be of the same type i.e. both long or both fixed

3) Source: 16-bit immediate value (decimal or hexadecimal) or 16-bit MPL data, Destination: high or low part of a 32-bit MPL data. The 32-bit MPL data can be either long or fixed

```
long_var(L) = -1;      // write value -1 (0xFFFF) into low part of long_var
fixed_var(H) = 0x2000; // write value 0x2000 into high part of fixed_var
```

```
long_var(L) = int_var; // copy int_var into low part of long_var
fixed_var(H) = int_var; // copy int_var into high part of fixed_var
```

4) Source: 16-bit MPL data left shifted 0 to 16. Destination: 32-bit MPL data. The 32-bit MPL data can be either long or fixed

```
long_var = int_var << 0; // copy int_var left shifted by 0 into long_var
fixed_var = int_var << 16; // copy int_var left shifted by 16 fixed_var
```

Remarks:

- *The left shift operation is done with sign extension. If you intend to copy the value of an integer MPL data into a long MPL data preserving the sign use this operation with left shift 0*
- *If you intend to copy the value of a 16-bit unsigned data into a 32-bit long variable, assign the 16-bit data in low part of the long variable and set the high part with zero.*

Examples:

```
var = 0xFFFF; // As integer, var = 1, as unsigned integer var = 65535
lvar = var << 0; // lvar = -1 (0xFFFFFFFF), the 16MSB of lvar are all set to 1 the
// sign bit of var
lvar(L) = var; // lvar(L) = 0xFFFF
lvar(H) = 0; // lvar(H) = 0. lvar = 65535 (0x0000FFFF)
```

5) Source: 2 consecutive memory locations, indicated through a pointer variable, Destination: 32-bit MPL data. The memory locations can be of 3 types: RAM for data (dm), RAM for MPL programs (pm), EEPROM SPI-connected for MPL programs (spi). The pointer variable indicates first of the 2 memory locations. If the pointer variable is followed by a + sign, after the assignment, it is incremented by 2. The destination can be either a long or a fixed MPL data

```
p_var = 0x4500; // set 0x4500 in pointer variable p_var
var1 = (p_var), spi; // var1 = value of the EEPROM memory location 0x4500
var1 = (p_var+), spi; // var1 = value of the EEPROM memory location 0x4500,
// then set p_var = 0x4502
p_var = 0x8200; // set 0x8200 in pointer variable p_var
var1 = (p_var), pm; // var1 = value of the RAM memory location 0x8200 for MPL
// programs
var1 = (p_var+), pm; // var1 = value of the RAM memory location 0x8200 for MPL
// programs, then set p_var = 0x8202
p_var = 0xA00; // set 0xA00 in pointer variable p_var
var1 = (p_var), dm; // var1 = value of the RAM memory location 0xA00 for MPL
// data
var1 = (p_var+), dm; // var1 = value of the RAM memory location 0xA00 for MPL
// data, then set p_var = 0xA02
```

6) Source: 32-bit immediate value (decimal or hexadecimal) or a 32-bit MPL data. Destination: 2 consecutive memory locations indicated through a pointer variable. The memory locations can be of 3 types: RAM for data (dm), RAM for MPL programs (pm), EEPROM SPI-connected for MPL programs (spi). The pointer variable indicates first of the 2 memory locations. If the pointer variable is followed by a + sign, after the assignment, it is incremented by 2. The source can be either a long or a fixed MPL data.

```

p_var = 0x4500;           // set 0x4500 in pointer variable p_var
(p_var),spi = 200000;    // write 200000 in the EEPROM memory location 0x4500
(p_var+),spi = var1;     // write var1 value in the EEPROM memory location
                        // 0x4500, then set p_var = 0x4502
p_var = 0x8200;         // set 0x8200 in pointer variable p_var
(p_var),pm = 3.5;       // write value 3.5 in RAM memory location 0x8200 for
                        // MPL programs
(p_var+),pm = var1;     // write var1 value in RAM memory location 0x8200 for
                        // MPL programs, then set p_var = 0x8202
p_var = 0xA00;          // set 0xA00 in pointer variable p_var
(p_var),dm = -1L;       // write -1 (0xFFFFFFFF) in the RAM memory location
                        // 0xA00
(p_var+),dm = var1;     // write var1 value in the RAM data memory location
                        // 0xA00, then set p_var = 0xA02

```

When this operation is performed having as source an immediate value, the MPL compiler checks the type and the dimension of the immediate value and based on this generates the binary code for a 16-bit or a 32-bit data transfer. Therefore if the immediate value has a decimal point, it is automatically considered as a fixed value. If the immediate value is outside the 16-bit integer range (-32768 to +32767), it is automatically considered as a long value. However, if the immediate value is inside the integer range, in order to execute a 32-bit data transfer it is necessary to add the suffix `L` after the value, for example: `200L` or `-1L`.

Examples:

```

user_var = 0x29E;        // write CPOS address in pointer variable user_var
(user_var),dm = 1000000; // write 1000000 (0xF4240) in the CPOS parameter i.e
                        // 0x4240 at address 0x29E and 0xF at address 0x29F
(user_var+),dm = -1;    // write -1 (0xFFFF) in CPOS(L). CPOS(H) remains
                        // unchanged. CPOS is (0xFFFF) i.e. 1048575,
                        // and user_var is incremented by 2
user_var = 0x29E;        // write CPOS address in pointer variable user_var
(user_var+),dm = -1L;   // write -1L long value (0xFFFFFFFF) in CPOS i.e.
                        // CPOS(L) = 0xFFFF and CPOS(H) = 0xFFFF,
                        // user_var is incremented by 2
user_var = 0x2A0;        // write CSPD address in pointer variable user_var
(user_var),dm = 1.5;    // write 1.5 (0x18000) in the CSPD parameter i.e

```

// 0x8000 at address 0x2A0 and 0x1 at address 0x2A1

Remark: The MPL assignment instructions with source an immediate value or a MPL data and destination a MPL data, use a **short address** format for the destination. The short address format requires a destination address between 0x200 and 0x3FF or between 0x800 and 0x9FF. This restriction is respected now by all the predefined or user-defined MPL data, hence you can use the above assignment instructions without checking the variables addresses.

However, considering possible future developments, the MPL also includes assignment instructions using a **full address** format where the destination address can be any 32-bit value. The following commands support full addressing:

```
long_var,dm = 100000; // set long_var = 100000 in using full addressing
long_var,dm = 0x100000; // set long_var = 0x100000 using full addressing
var_dest,dm = var_source; // copy value of var_source in var_dest using
// full addressing
```

See also:

[Assignment and Data Transfer. 16-bit data – MPL Programming Details MPL Description](#)

6.2.3.5. Arithmetic and logic manipulation

The MPL offers the possibility to perform the following operations with the MPL data:

- Addition
- Subtraction
- Multiplication
- Division
- Left and right shift
- logic AND / OR

Except the multiplication, the result of these operations is saved in the left operand. For the multiplication, the result is saved in the dedicated product register. The operands are always treated as signed numbers and the right shift is performed with sign-extension.

Addition: The right-side operand is added to the left-side operand

The left side operand can be:

- A 16-bit MPL data: MPL parameter or user variable
- A 32-bit MPL data: MPL parameter or user variable

The right side operand can be:

- A 16-bit immediate value
- A 16-bit MPL data: MPL parameter, variable or user variable
- A 32-bit immediate value, if the left side operand is a 32-bit MPL data

-
- A 32-bit MPL data: MPL parameter, variable or user variable, if the left side operand is a 32-bit data too

Programming Examples

```
int_var += 10;           // int_var1 = int_var1 + 10
int_var += int_var2;    // int_var = int_var + int_var2
long_var += -100; // long_var = long_var + (-100) = long_var - 100
long_var += long_var2; // long_var = long_var + long_var2
fixed_var += 10.; // fixed_var = fixed_var + 10.0
fixed_var += fixed_var2; // fixed_var = fixed_var + fixed_var2
```

Subtraction: The right-side operand is subtracted from the left-side operand

The left side operand can be:

- A 16-bit MPL data: MPL parameter or user variable
- A 32-bit MPL data: MPL parameter or user variable

The right side operand can be:

- A 16-bit immediate value
- A 16-bit MPL data: MPL parameter, variable or user variable
- A 32-bit immediate value, if the left side operand is a 32-bit MPL data
- A 32-bit MPL data: MPL parameter, variable or user variable, if the left side operand is a 32-bit data too

Programming Examples

```
int_var -= 10;           // int_var1 = int_var1 - 10
int_var -= int_var2;    // int_var = int_var - int_var2
long_var -= -100; // long_var = long_var - (-100) = long_var + 100
long_var -= long_var2; // long_var = long_var - long_var2
fixed_var -= 10.; // fixed_var = fixed_var - 10.0
fixed_var -= fixed_var2; // fixed_var = fixed_var - fixed_var2
```

Remark: At addition and subtraction, when the left operand is a 32-bit long or fixed MPL data and the right operand is a 16-bit integer value, it is treated as follows:

- Sign extended to a 32-bit long value, if the left operand is a 32-bit long
- Set as the integer part of a fixed value, if the left operand is a 32-bit fixed

Multiplication: The 2 operands are multiplied and the result is saved in a dedicated 48-bit product register (PREG). This can be accessed via the MPL variables: **PRODH** – the 32 most significant bits, and **PROD** – the 32 least significant bits of the product register. The result of the multiplication can be left or right-shifted with 0 to 15 bits, before being stored in the product register. At right shifts, high order bits are sign-extended and the low order bits are lost. At left shifts, high order bits are lost and the low order bits are zeroed. The result is preserved in the product register until the next multiplication.

The first (left) operand can be:

- A 16-bit MPL data: MPL parameter, variable or user variable
- A 32-bit MPL data: MPL parameter, variable or user variable

The second (right) operand can be:

- A 16-bit immediate value
- A 16-bit MPL data: MPL parameter, variable or user variable

Remark: *The result is placed in the product register function of the left operand. When shift is 0:*

- *In the 32 least significant bits, when the left operand is a 16-bit integer. The result is a 32-bit long integer*
- *In all the 48 bits, when the left operand is a 32-bit fixed. The result has the integer part in the 32 most significant bits and the fractional part in the 16 least significant bits*
- *In all the 48 bits, when the left operand is a 32-bit long. The result is a 48-bit integer*

Programming Examples

```
long_var * -200 << 0;      // PROD = long_var * (-200)
fixed_var * 10 << 5;      // PROD = fixed_var * 10 * 25 i.e. fixed_var *320
int_var1 * int_var2 >> 1; // PROD = (int_var1 * int_var2) / 2
long_var * int_var >> 2;  // PROD = (long_var * int_var) / 4
long_var = PROD;         // save 32LSB of PROD in long_var
long_var = PROD(H);     // save 32MSB of PROD in long_var i.e. bits 47-15
```

Division: The left operand – the dividend, is divided by the right operand – the divisor, and the result is saved in the left operand..

The first (left) operand is a 32-bit MPL data: MPL parameter or user variable.

The second (right) operand is a 16-bit MPL data: MPL parameter, variable or user variable

The result, saved in the first operand is a fixed value with the integer part in the 16 most significant bits and the fractional part in the 16 least significant bits.

Programming Examples

```
long_var /= int_var;      // long_var = long_var / int_var
fixed_var /= int_var;    // fixed_var = fixed_var / int_var
```

Left and right shift: The operand is left or right shifted with 0 to 15. The result is saved in the same operand. At right shifts, high order bits are sign-extended and the low order bits are lost. At left shifts, high order bits are lost and the low order bits are zeroed.

The operand can be:

- A 16-bit MPL data: MPL parameter, variable or user variable
- A 32-bit MPL data: MPL parameter, variable or user variable

-
- The 48-bit product register with the result of the last multiplication

Programming Examples

```
long_var << 3;           // long_var = long_var * 8
int_var = -16;          // int_var = -16 (0xFFF0)
int_var >> 3;           // int_var = int_var / 8 = -2 (0xFFFE)
PROD << 1;              // PREG = PREG * 2
```

Remark: The shifts instructions having *PROD* as operand are performed on all the 48-bits of the product register.

Logic AND / OR: A logic AND is performed between the operand and a 16-bit data (the AND mask), followed by a logic OR between the result and another 16-bit data (the OR mask).

The operand is a 16-bit MPL data: MPL register, MPL parameter or user variable

The AND and OR masks are 16-bit immediate values, decimal or hexadecimal.

Programming Examples

```
int_var = 13;           // int_var = 13 (0xD)
SRB int_var, 0xFFFE, 0x2; // set int_var bit 0 = 0 and bit 1 = 1
                          // int_var = 12 (0xC)
```

The **SRB** instruction allows you to set/reset bits in a MPL data in a safe way avoiding the interference with the other concurrent processes wanting to change the same MPL data. This is particularly useful for the MPL registers, which have bits that can be manipulated by both the drive/motor and the user at MPL level.

Remark: The **SRB** instruction, use a **short address** format for the operand. The short address format requires an operand address between 0x200 and 0x3FF or between 0x800 and 0x9FF. This restriction is respected now by all the predefined or user-defined MPL data, hence you can use the above assignment instructions without checking the variables addresses.

However, considering possible future developments, the MPL also includes a similar instruction **SRBL** using a **full address** format where the operand address can be any 16-bit value. The **SRBL** command has the following mnemonic:

```
SRBL MPLvar, 0xFFFE, 0x2; // set bit 0 = 0 and bit 1 = 1 in MPLvar with
                          // using full addressing
```

See also:

[MPL Description](#)

6.2.3.6. Multi-axis control

6.2.3.6.1. Axis identification

In multiple-axis configurations, each axis (drive/motor) needs to be identified through a unique number – the **axis ID**. This is a value between 1 and 255. If the destination of a message is specified via an axis ID, the message is received only by the axis with the same axis ID. The axis ID is initially set at power on using the following algorithm:

- a. With the value read from the EEPROM setup table containing all the setup data.
- b. If the setup table is invalid, with the last axis ID value read from a valid setup table
- c. If there is no axis ID set by a valid setup table, with the value read from the hardware switches/jumpers for axis ID setting
- d. If the drive/motor has no hardware switches/jumpers for axis ID setting, with the default axis ID value which is 255.

Remark: *If the axis ID read from a valid setup table is 0, the axis ID is set with the value read from the hardware switches/jumpers or in their absence according with d)*

Typically, the axis ID is kept constant during operation at the value established during the setup phase. However, if needed, you can change the axis ID to any of the 255 possible values, using the MPL instruction **AXISID**, followed by an integer value between 1 and 255.

Apart from the axis ID, each drive has also a **group ID**. The group ID represents a filter for multicast messages. The destination of a multicast message is specified via a group ID. When a multicast message is received, each axis compares the group ID from the message with its own group ID. If the axis group ID has a bit in common with the group ID from the message, the message is accepted. The group ID is an 8-bit integer value. Each bit corresponds to one group: bit 0 – group 1, bit 1 – group 2... bit 7 – group 8. Hence a drive/motor can be programmed to be member of up to 8 groups. When a MPL command is sent to a group, all the axes members of this group will receive the command. For example, if a drive/motor has the group ID = 11 (1011b), it is member of groups 1, 2 and 4 and will receive the messages sent any of these groups.

For each drive/motor you can:

- Set its group ID using the MPL instruction **GROUPID**
- Add new groups to its group ID using the MPL instruction **ADDGRID**
- Remove groups from its group ID using the MPL instruction **REMGRID**.

Remarks:

- *You can read at any moment the actual values of the **axis ID** and **group ID** of a drive/motor from the Axis Address Register [AAR](#)*
- *By default all the drives are set as members of group 1.*
- *A broadcast to all the axes means to send a message with the destination group ID = 0*

Variables

[AAR](#)

MPL register (Axis Address Register). Contains the Group ID in the 8MSB and the Axis ID in the 8LSB

Instructions

- [AXISID](#) value** Set axis ID = value. Value is an 8-bit integer between 1 and 255
- [GROUPID](#) (1,3,5,..)** Set group ID = value. Value is an 8-bit integer, where:
- Bit 0 is set to 1, if (group) 1 occurs in the parenthesis, else it is set to 0
 - Bit 1 is set to 1, if (group) 2 occurs in the parenthesis, else it is set to 0
 - ...
 - Bit 7 is set to 1, if (group) 8 occurs in the parenthesis, else it is set to 0
- [ADDGRID](#) (2,4,6...)** Add the groups from parenthesis to the Group ID. The corresponding bits from Group ID will be set to 1
- [REMGRID](#) (2,5...)** Remove the groups from parenthesis from the Group ID. The corresponding bits from Group ID will be set to 0

Programming Example

```
AXISID 10;           // set axis ID = 10
GROUPID (2,3);      // set group ID = 6 (110b) i.e. bits 1, 2 = 1
ADDGRID (4);        // add group 4. Group ID = 14 (1110b) i.e. bits 1, 2, 3 = 1
REMGRID (2,4);      // remove groups 2 and 4. Group ID = 4 (100b) i.e. bit 2 = 1
                    // AAR = 40Ah i.e. group ID = 4 and axis ID = 10 (Ah)
```

See also:

[Communication Protocols – RS232 & RS485](#)

[Communication Protocols – CAN](#)

[MPL Description](#)

6.2.3.6.2. Data transfers between axes

There are 2 categories of data transfer operations between axes:

1. Read data from a remote axis. A variable or a memory location from the remote axis is saved into a local variable
2. Write data to a remote axis or group of axes. A variable or a memory location of a remote axis or group of axes is written with the value of a local variable

In a read data from a remote axis operation:

- The source is placed on a remote axis and can be:
 - A 16-bit MPL data: MPL register, parameter, variable or user variable
 - A memory location indicated through a pointer variable
- The destination is placed on the local axis and can be:
 - A 16-bit MPL data: MPL register, parameter or user variable

Programming Examples

1) Source: remote 16-bit MPL data, Destination: local 16-bit MPL data.

```
local_var = [2]remote_var; // set local_var with value of remote_var from axis 2
```

Remark: If *remote_var* is a user variable, it has to be declared in the local axis too. Moreover, for correct operation, *remote_var* must have the same address in both axes, which means that it must be declared on each axis on the same position. Typically, when working with data transfers between axes, it is advisable to establish a block of user variables that may be the source, destination or pointer of data transfers, and to declare these data on all the axes as the first user variables. This way you can be sure that these variables have the same address on all the axes.

2) Source: remote memory location pointed by a remote pointer variable, Destination: 16-bit MPL data. The remote memory location can be of 3 types: RAM memory for MPL data (*dm*), RAM memory for MPL programs (*pm*), EEPROM SPI-connected memory for MPL programs (*spi*). If the pointer variable is followed by a + sign, after the assignment, the pointer variable is incremented by 1 if the destination is a 16-bit integer or by 2 if the destination is a 32-bit long or fixed

```
local_var = [2](p_var),spi; // local_var = value of EEPROM program memory  
// location from axis 2, pointed by p_var from axis 2  
long_var = [3](p_var+),dm; // local long_var = value of RAM data memory  
// locations from axis 3, pointed by p_var from axis 3  
// p_var is incremented by 2  
int_var = [4](p_var+),pm; // local int_var = value of RAM program memory  
// location from axis 4, pointed by p_var from axis 4;  
// p_var is incremented by 1
```

Remark: The MPL instructions for data transfers between axes use a **short address** format for the remote source when this is a MPL data. The short address format requires a source address between 0x200 and 0x3FF or between 0x800 and 0x9FF. This restriction is respected now by all the predefined or user-defined MPL data, hence you can use the above assignment instructions without checking the variables addresses.

However, considering possible future developments, the MPL also includes data transfers using a **full address** format where the source address can be any 16-bit value. The following command supports full addressing:

```
local_var = [2]remote_var,dm; // set local_var with value of remote_var  
// from axis 2 using extended addressing
```

In a write data to a remote axis or group of axes operation:

- The source is placed on the local drive and can be:
 - A 16-bit MPL data: MPL register, parameter, variable or user variable
- The destination is placed on the remote axis or group of axes and can be:
 - A 16-bit MPL data: MPL register, parameter or user variable
 - A memory location indicated through a pointer variable

Programming Examples

1) Source: local 16-bit MPL data, Destination: remote 16-bit MPL data.

```
[2]remote_var = local_var; // set remote_var from axis 2 with local_var value
[G2]remote_var = local_var; // set remote_var from group 2 with local_var value
[B]remote_var = local_var; // set remote_var from all axes with local_var value
// broadcast with group ID = 0 -> got by everyone
```

2) Source: 16-bit MPL data, Destination: remote memory location pointed by a remote pointer variable. The remote memory location can be of 3 types: RAM memory (dm), RAM memory for MPL programs (pm), EEPROM SPI-connected memory for MPL programs (spi). If the pointer variable is followed by a + sign, after the assignment, the pointer variable is incremented by 1 if the source is a 16-bit integer or by 2 if the source is a 32-bit long or fixed

```
[2](p_var),spi = local_var; // set local_var value in EEPROM program memory
// location from axis 2, pointed by p_var from axis 2
[G3](p_var+),dm = long_var; // set local long_var value in RAM data memory
// location from group 3 of axes, each location being
// pointed its own p_var, which is incremented by 2
[4](p_var+),pm = int_var; // set local int_var value in RAM program memory
// location from axis 4, pointed by p_var from axis 4;
// p_var is incremented by 1
```

Remark: The MPL instructions for data transfers between axes use a **short address** format for the remote destination when this is a MPL data. The short address format requires a destination address between 0x200 and 0x3FF or between 0x800 and 0x9FF. This restriction is respected now by all the predefined or user-defined MPL data, hence you can use the above assignment instructions without checking the variables addresses.

However, considering possible future developments, the MPL also includes data transfers using a **full address** format where the destination address can be any 16-bit value. The following command supports full addressing:

```
[G2]remote_var,dm = local_var; // set remote_var from group 2 with
// local_var value, using extended addressing
```

See also:

[MPL Description](#)

6.2.3.6.3. Remote control

The MPL includes powerful instructions through which you can program a drive to issue MPL commands to another drive or group of drives. You can include these instructions in the MPL program of a drive, which can act like a host and can effectively control the operation of the other drives from the network. These MPL instructions are:

```
[axis]{MPL command1; MPL command2;...};  
[group]{MPL command1; MPL command2;...};  
[broadcast]{MPL command1; MPL command2;...};
```

where MPL command1, MPL command2, etc. can be any single axis MPL instructions. A single axis MPL instruction is defined as an instruction that does not transfer data or sends MPL commands to other axes. If you include multiple MPL commands separated by semicolon (;), these will be sent one by one in order from left to right i.e. first MPL command1, then MPL command2, etc.

Remark: Most of the MPL instructions enter in the category of those that can be sent by a drive/motor to another one using the above MPL commands.

Programming Examples

```
[G1]{CPOS=2000;}; // send a new CPOS command to all axes from group 1  
[G1]{UPD}; // send an update command to all the axes from group 1  
// all axes from group 1 will start to move simultaneously  
[B]{STOP;}; // broadcast a STOP command to all axes from the network
```

See also:

[MPL Description](#)

6.2.3.6.4. Axis Synchronization

The MPL provides a synchronization procedure between the ElectroCraft drives/motors connected in a CAN network. When the synchronization procedure is active, the execution of the control loops is synchronized within a 10 time interval. Due to this powerful feature, drifts between the drives/motors are eliminated.

The synchronization process is performed in two steps. First, the synchronization master sends a synchronization message to all axes, including to itself. When this message is received, all the axes read their own internal time. Next, the master sends its internal time to all the synchronization slaves, which compare it with their own internal time. If there are differences, the slaves correct slightly their sampling periods in order to keep them synchronized with those of the master.

A drive/motor becomes the synchronization master when it receives the MPL command **SETSYNC value** where **value** represents the time interval in internal units between the synchronization messages sent by the synchronization master. Recommended value is 20ms.

6.2.3.7. Monitoring

6.2.3.7.1. Position Triggers

A *position trigger* is a position value with which the actual position is continuously compared. The compare result is shown in the Status Register High (**SRH**). If the actual position is below a position trigger, the corresponding bit from SRH is set to 0, else it is set to 1.

In total there are 4 position triggers. Their status is shown in **SRH** bits 4 to 1. The position triggers are set in the following MPL parameters:

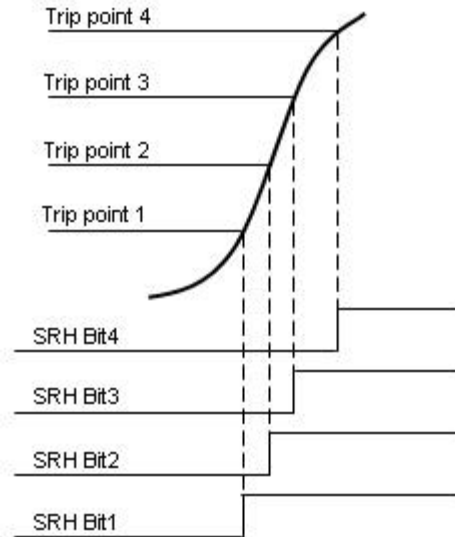
POSTRIGG1 – for Position Trigger 1

POSTRIGG2 – for Position Trigger 1

POSTRIGG3 – for Position Trigger 1

POSTRIGG4 – for Position Trigger 1

You can change at any moment the value of a position trigger.



The actual position that is compared with the position triggers is:

- The **Load position feedback** (MPL variable **APOS_LD**) for configurations with position sensor
- The **position reference** (MPL variable **TPOS** – Target position) in the case of steppers controlled in open-loop

Remark: The position triggers can be used to monitor the motion progress. If this operation is done from a host, you may program the drive/motor to automatically issue a message towards the host, each time when the status of a position trigger is changed.

See also:

[Position Triggers – Related MPL Instructions and Data](#)

[MPL Description](#)

6.2.3.7.2. Position Triggers - Related MPL Instructions and Data

Parameters

POSTRIGG1	Position trigger 1. Measured in position units .
POSTRIGG2	Position trigger 2. Measured in position units .
POSTRIGG3	Position trigger 3. Measured in position units .
POSTRIGG4	Position trigger 4. Measured in position units .

Variables

APOS LD	Actual load position. Measured in position units . Alternate name: APOS
TPOS	Target position – position reference computed by the reference generator at each slow loop sampling period. Measured in position units

Programming Example

```
// Position feedback: 500 lines incremental
// encoder (2000 counts/rev)
POSTRIGG1 = 2000;//Set First Position Trigger = 1[rot]
```

See also:

[Position Triggers – MPL Programming Details](#)

[MPL Description](#)

6.2.3.7.3. Status Register

The drive/motor status condition is described in registers [SRH](#) and [SRL](#).

See also:

[Status register low part – SRL](#)

[Status register high part – SRH](#)

[MPL Description](#)

6.2.3.7.4. FAULT Status

A drive/motor enters in the FAULT status, when an error occurs. In the FAULT status:

- The drive/motor is in AXISOFF with the control loops and the power stage deactivated
- The MPL program execution is stopped
- The error register **MER** shows the type of errors detected and the status register **SRH.15** signals the fault condition
- Ready and error outputs (if present) are set to the **not ready** level, respectively to the **error** active level. When available, ready green led is turned off and error red led is turned on

Remark: *The following conditions signaled in MER do not set the drive/motor in fault status:*

- *Drive /motor disabled due to the enable input set on the disable level*
- *Command error*
- *Negative limit switch input on active level*
- *Positive limit switch input on active level*
- *Position wraparound*
- *Serial and CAN bus communication errors*

You can modify this default behavior by changing the [MPL interrupt](#) service routines

The drive/motor can be got out from the FAULT status, with the MPL command **FAULTR** – fault reset. This command clears most of the error bits from MER, sets the ready output (if available) to the **ready** level, and sets the error output (if available) to the **no error** level.

Remarks:

- *The FAULTR command does not change the status of MER.15 (enable input on disabled level), MER.7 (negative limit switch input active), MER.6 (positive limit switch input active) and MER.2 (invalid setup table)*
- *The drive/motor will return to FAULT status if there are errors when the FAULTR command is executed*

See also:

[MPL Description](#)

6.2.3.7.5. Messages sent to the host

You can program a drive/motor to send messages to your host. The messages are all of type “**Take Data 2**” (see Communication Protocols: [RS232 & RS485](#) or [CAN](#)) i.e. return the value of a MPL data, like if the drive/motor would have had received a “**Give Me Data 2**” request from the host to return that MPL data.

The message transmission can be triggered by:

- Conditions which change the status registers **SRL**, **SRH** or the error register **MER**
- The execution of the MPL command **SEND** from your MPL program. Through this command you can send to your host the contents of any MPL data

In the first case, you can select the registers bits, which will trigger a message when are changed. The selection is done via 3 masks, one for each register, set in MPL parameters: **SRL_MASK**, **SRH_MASK**, **MER_MASK**. A bit set in a mask, enables a message transmission when the same bit from the corresponding register changes.

When the transmission is triggered by a bit change in **SRH** (high part) or **SRL** (low part), the message sent contains these 2 registers grouped together as a single 32-bit register/data, with **SRH** on bits 31-16. When the transmission is triggered by a bit change in **MER**, the message sent contains this register.

The host ID is specified via the MPL parameter **MASTERID**. This contains the host ID (an integer value between 1 and 255), multiplied by 16, plus 1. For example, if the host ID is 1, the value of **MASTERID** must be $1 * 16 + 1 = 17$.

***Remark:** By default, at power on, the **MASTERID** is set for a host ID equal with the drive/motor axis ID. Therefore, the messages will be sent via RS-232 serial communication. If the host ID is set different from the drive/motor axis ID, the messages are sent via the other communication channels: CAN bus, RS485, etc*

Parameters

MASTERID Provides the host ID (address), according with formula:

$$\mathbf{MASTERID = host\ ID * 16 + 1}$$

SRL_MASK Mask for **SRL** register. A bit set to 1, enables to send **SRH** and **SRL** when the same bit from **SRL** changes

SRH_MASK Mask for **SRH** register. A bit set to 1, enables to send **SRH** and **SRL** when the same bit from **SRH** changes

MER_MASK Mask for **MER** register. A bit set to 1, enables to send **MER** when the same bit from **MER** changes

Variables

SRL MPL register. Low part of the 32-bit status register grouping key information about the drive/motor status

SRH MPL register. High part of the 32-bit status register grouping key information concerning the drive/motor status

MER MPL register. Groups all the errors conditions

Instructions

SEND var Sends a “Take Data 2” message with var contents. Var can be any 16-bit or 32-bit MPL data: register, parameter or variable

Programming Examples

```
MASTERID = 33; // Set host ID / address = 2
//Send SRH & SRL if motion complete or pos. trigger 1 bits change
SRH_MASK = 0x0002;
SRL_MASK = 0x0400;
MER_MASK = 0xFFFF; // send MER on any bit change
SEND CAPPOS; // Send to host contents of variable CAPPOS
```

See also:

[Communication Protocols – RS232 & RS485](#)

[Communication Protocols – CAN](#)

[MPL Description](#)

6.2.3.8. Miscellaneous

This category includes the following MPL instructions:

FAULTR

Fault reset. Gets out the drive/motor from the [FAULT status](#) in which it enters when an error occurs. After a **FAULTR** command, most of the error bits from [MER](#) are cleared (set to 0), the Ready output (if present) is set to “ready” level, the Error output (if present) is set to “no error” level.

Remarks:

- *The FAULT reset command does not change the status of MER.15 (enable input on disabled level), MER.7 (negative limit switch input active), MER.6 (positive limit switch input active) and MER.2 (invalid setup table)*
- *The drive/motor will return to FAULT status if there are errors when the FAULTR command is executed*

SAVE

Saves the actual values of the MPL parameters from the RAM memory into the EEPROM memory, in the setup table. Through this command, you can save all the setup modifications done, after the power on initialization.

SCIBR value16

Changes the serial communication interface (SCI) baud rate. SCI is used in data

Value 16	SCI baud rate
0	9600
1	19200
2	38400
3	56500
4	115200

exchanges on RS232 or RS485

The serial baud rate is set at power on using the following algorithm:

- a. With the value read from the EEPROM setup table
- b. If the setup table is invalid, with the last baud rate read from a valid setup table
- c. If there is no baud rate set by a valid setup table, with 9600.

Remarks:

- *Use this command when a drive/motor operates in AUTORUN (after power on starts to execute the MPL program from the EEPROM) and it must communicate with a host at a baud rate different from the default value. In this case, the MPL program must start with a serial baud rate change.*
- *An alternate solution to the above case is to set via **SCIBR** command the desired baud rate and then to save it in the EEPROM, with command **SAVE**. After a reset, the drive/motor starts directly with the new baud rate, if the setup table was valid. Once set, the new default baud rate is preserved, even if the setup table is later on disabled, because the default serial baud rate is stored in a separate area of the EEPROM.*

CANBR value16

Changes the CAN bus baud rate as follows:

Value 16	CAN baud rate
125	0xF36C
250	0x736C
500	0x3273
800	0x412A
1000	0x1273

The CAN baud rate is set at power on using the following algorithm:

- d. With the value read from the EEPROM setup table
- e. If the setup table is invalid, with the last baud rate read from a valid setup table
- f. If there is no baud rate set by a valid setup table, with 500kb.

Remarks:

- *Use this command when a drive/motor operates in AUTORUN (after power on starts to execute the MPL program from the EEPROM) and it must communicate with a host at a baud rate different from the default value. In this case, the MPL program must start with a CAN baud rate change.*
- *An alternate solution to the above case is to set via **CANBR** command the desired baud rate and then to save it the EEPROM, with command **SAVE**. After a reset, the drive/motor starts directly with the new baud rate, if the setup table was valid. Once set, the new default baud rate is preserved, even if the setup table is later on disabled, because the default CAN baud rate is stored in a separate area of the EEPROM*

LOCKEEPROM value16 Locks or unlocks the EEPROM write protection. When the EEPROM is write-protected, it is not possible to write data into the EEPROM, with the exception of the MPL command SAVE. This command temporary unlocks the EEPROM, saves the setup data and then locks back the EEPROM. Value16 may have the following values:

- 0 – Disables EEPROM write protection
- 1 – Enables write protection for the last quarter of the EEPROM
- 2 – Enables write protection for the last half of the EEPROM
- 3 – Enables write protection for the entire EEPROM

Example: An EEPROM has 8Kwords. In the MPL program space occupies the address range: 4000-5FFFh. **LOCKEEPROM 1** protects the address range: 5800-5FFFh, **LOCKEEPROM 2** protects the address range: 5000-5FFFh and **LOCKEEPROM 3** protects the entire address range: 4000-5FFFh.

ENEEPROM Enables EEPROM usage after it was disabled by the initialization of feedback devices like SSI or EnDat encoders using the same SPI link as the EEPROM

NOP No operation

BEGIN First instruction of a MPL program.

END

Last instruction of the main section of a MPL program. When END instruction is executed, the MPL program execution is stopped.

Remark: *It is mandatory to end the main section of a MPL program with an END command. All the MPL functions and the MPL interrupt service routines must follow after the END command.*

ENDINIT

END of the INITialization part of the MPL program. This command uses the available setup data to perform key initializations, but does not activate the controllers or the PWM outputs. These are activated with the **AXISON** command

Remarks:

- *After power on, the **ENDINIT** command may be executed only once. Subsequent **ENDINIT** commands are ignored.*
- *The first **AXISON** command must be executed only after the **ENDINIT** command*
- *Typically, the **ENDINIT** command is executed at the beginning of a MPL program and may be followed by the **AXISON** command even if no motion mode was set. In the absence of any programmed motion, the drive applies zero voltage to the motor. Alternately, after **ENDINIT** you can set a first motion and then execute **AXISON***

See also:

[MPL Description](#)

6.2.4. MPL Instruction set

6.2.4.1. MPL Instructions

This section describes the complete set of MPL instructions, grouped by functionality. In each group, the instructions are ordered alphabetically. The groups are:

- Motion programming and control, including
 - [Motion configuration](#)
 - [Motor commands](#)
- Program flow (decision) group
 - [Events](#)
 - [Motion Controller Events](#)
 - [Jumps and function calls](#)
 - [MPL interrupts](#)
- [I/O handling](#) (firmware [FAxx](#))
- [I/O handling](#) (firmware [FBxx](#))
- [Assignment and data transfer](#)
- [Arithmetic and logic operations](#)
- [Multi axis control and monitoring](#)
- [Miscellaneous](#)
- [On-line commands](#)

The presentation also lists the [Obsolete instructions](#) together with their equivalents.

The description of each MPL instruction includes:

- Syntax
- Operands
- Binary code
- Description
- Example(s)

All the notational conventions used are grouped in the [symbols](#) section.

6.2.4.2. Symbols used in instructions descriptions

Symbol	Description
&Label	Value of a MPL program label i.e. a MPL program address
&V16	Address of a 16-bit integer variable
&V32	Address of a 32-bit long or fixed variable
(V16)	Contents of memory location from address equal with V16 value
(fa)	Full full addressing. Source/destination operand provided with 16-bit address. Some MPL instructions using 9-bit short addressing are doubled with their long addressing equivalent
9LSB(&V16)	The 9 LSB (less significant bits) of the address of a 16-bit integer
9LSB(&V32)	The 9 LSB (less significant bits) of the address of a 32-bit long or fixed
A	Message destination is an axis indicated via its Axis ID
A/G	Message destination can be an axis indicated via an Axis ID or a group of axes indicated by a Group ID
ANDdis	16-bit AND mask. See Table MCRx & AND/OR masks for DISIO#n and Table MCRx & PxDIR addresses
ANDen	16-bit AND mask. See Table MCRx & AND/OR masks for ENIO#n and Table MCRx & PxDIR addresses
ANDin	16-bit AND mask. See Table AND/OR masks for SETIO#n IN
ANDm	16-bit user-defined AND mask
ANDout	16-bit AND mask. See Table AND/OR masks for SETIO#n OUT
ANDrst	16-bit AND mask. See Table AND/OR masks for ROUT#n
ANDset	16-bit AND mask. See Table AND/OR masks for SOUT#n
Bit_mask	16-bit AND mask. See Tables PxDIR & Bit_mask for V16=IN#n and table MCRx & PxDIR addresses
D_ref	32-bit fixed value
D_time	16-bit value
Flag	Condition Flag for GOTO/CALL
LengthMLI	Length of a MPL instruction code in words – 1
MCRx	See Tables MCRx & AND/OR masks for ENIO#n / DISIO#n and Table MCRx & PxDIR addresses
ORdis	16-bit OR mask. See Table MCRx & AND/OR masks for DISIO#n and Table MCRx & PxDIR addresses
ORen	16-bit OR mask. See Table MCRx & AND/OR masks for ENIO#n and Table MCRx & PxDIR addresses
ORin	16-bit OR mask.. See Table AND/OR masks for SETIO#n IN
ORm	16-bit user-defined OR mask
ORout	16-bit OR mask. See Table AND/OR masks for SETIO#n OUT
ORrst	16-bit OR mask. See Table AND/OR masks for ROUT#n

ORset	16-bit OR mask. See Table AND/OR masks for SOUT#n
PxDIR	See Table PxDIR & Bit_msk for V16=IN#n and Table MCRx & PxDIR addresses
DM	RAM memory for MPL data
PM	RAM memory for MPL programs
SPI	E2ROM memory for MPL programs
TM	Type of memory. When used in syntax TM should be replaced by <i>DM</i> or <i>PM</i> or <i>SPI</i> . When used in code, see Table TM values.
VAR	Any 16/32 –bit MPL data i.e.: register, parameter, variable, user-variable
VAR16	Any 16-bit integer MPL data
VAR16D	A 16-bit integer MPL parameter or user-variable, used as destination:
VAR16S	Any 16-bit integer MPL data used as source
VAR32	Any 32-bit long or fixed MPL data i.e.: parameter, variable, user-variable
VAR32(L)	16LSB of a 32-bit long or fixed variable (seen as a 16-bit integer)
VAR32(H)	16MSB of a 32-bit long or fixed variable (seen as a 16-bit integer)
VAR32D	A 32-bit long or fixed MPL parameter of user variable, used as destination
VAR32S	Any 32-bit long or fixed MPL data
value16	16-bit integer value
value32	32-bit long or fixed value
value32(L)	16LSB of a 32-bit long or fixed value
value32(H)	16MSB of a 32-bit long or fixed value

6.2.4.3. Instructions Categories

6.2.4.3.1. Motion configuration

Syntax	Description
CIRCLE	Define circular segment for vector mode
CPA	Command Position is Absolute
CPR	Command Position is Relative
EXTREF	Set external reference type
INITCAM addrS, addrD	Copy CAM table from EEPROM (addrS address) to RAM (addrD address)
LPLANE	Define coordinate system for linear interpolation mode
MODE CS	Set MODE Cam Slave
MODE GS	Set MODE Gear Slave
MODE LI	Set MODE Linear Interpolation
MODE PC	Set MODE Position Contouring
MODE PE	Set MODE Position External
MODE PP	Set MODE Position Profile
MODE PSC	Set MODE Position S-Curve
MODE PT	Set MODE PT
MODE PVT	Set MODE PVT
MODE SC	Set MODE Speed Contouring
MODE SE	Set MODE Speed External
MODE SP	Set MODE Speed Profile
MODE TC	Set MODE Torque Contouring
MODE TEF	Set MODE Torque External Fast
MODE TES	Set MODE Torque External Slow
MODE TT	Set MODE Torque Test
MODE VC	Set MODE Voltage Contouring
MODE VEF	Set MODE Voltage External Fast
MODE VES	Set MODE Voltage External Slow
MODE VM	Set MODE Vector Mode
MODE VT	Set MODE Voltage Test
PTP	Define a PT point
PVTP	Define a PVT point

<u>REG_OFF</u>	Disable superposed mode
<u>REG_ON</u>	Enable superposed mode
<u>RGM</u>	Reset electronic gearing/camming master mode
<u>SEG</u>	Define a contouring segment
<u>SETPT</u>	Setup PT mode operation
<u>SETPVT</u>	Setup PVT mod operation
<u>SGM</u>	Set electronic gearing/camming master mode
<u>TUM0</u>	Target update mode 0
<u>TUM1</u>	Target update mode 1
<u>VPLANE</u>	Define coordinate system for Vector Mode
<u>VSEG</u>	Define linear segment for vector mode

6.2.4.3.2. Motor commands

Syntax	Description
<u>AXISOFF</u>	AXIS is OFF (deactivate control)
<u>AXISON</u>	AXIS is ON (activate control)
<u>ENDINIT</u>	END of Initialization
<u>RESET</u>	RESET drive / motor
<u>SAP</u>	Set Actual Position
<u>STA</u>	Set Target position = Actual position
<u>STOP</u>	STOP motion
<u>STOP!</u>	STOP motion when the programmed event occurs
<u>UPD</u>	Update motion mode and parameters. Start motion
<u>UPD!</u>	Update motion mode and parameters when the programmed event occurs

6.2.4.3.3. Events

Syntax	Description
!ALPO	Set event when absolute load position is over a value
!ALPU	Set event when absolute load position is under a value
!AMPO	Set event when absolute motor position over a value
!AMPU	Set event when absolute motor position under a value
!CAP	Set event when a capture input goes low or high
!IN#n	Set event when digital input #n goes low or high
!LSN	Set event when the negative limit switch (LSN) goes low or high
!LSP	Set event when positive limit switch (LSP) goes low or high
!LSO	Set event when load speed is over a value
!LSU	Set event when load speed is under a value
!MC	Set event when the actual motion is completed
!MSO	Set event when motor speed is over a value
!MSU	Set event when motor speed is under a value
!PRO	Set event when position reference is over a value
!PRU	Set event when position reference is under a value
!RPO	Set event when relative load position is over a value
!RPU	Set event when relative load position is under a value
!RT	Set event after a wait time
!SRO	Set event if speed reference is over a value
!SRU	Set event if speed reference is under a value
!TRO	Set event if torque reference is over a value
!TRU	Set event if torque reference is under a value
!VO	Set event if a long/fixed variable is over a value
!VU	Set event if a long/fixed variable is under a value
!WAIT!	Wait until the programmed event occurs

6.2.4.3.4. Jumps and function calls

Syntax

[ABORT](#)

[CALL](#)

[CALLS](#)

[GOTO](#)

[RET](#)

Description

Abort the execution of a function called with CALLS

Call a MPL function

Cancelable CALL of a MPL function

Jump

Return from a MPL function

6.2.4.3.5. MPL interrupts

Syntax

[DINT](#)

[EINT](#)

[RETI](#)

Description

Disable globally all MPL interrupts

Enable globally all MPL interrupts

Return from a MPL Interrupt Service Routine

6.2.4.3.6. I/O handling (Firmware FAxx)

Syntax	Description
DISCAPI	Disable 1st capture/encoder index input to detect transitions
DIS2CAPI	Disable 2nd capture/encoder index input to detect transitions
DISLSN	Disable negative limit switch (LSN) input to detect transitions
DISLSP	Disable positive limit switch (LSP) input to detect transitions
EN2CAPI0	Enable 2nd capture/encoder index input to detect a high to low transition
EN2CAPI1	Enable 2nd capture/encoder index input to detect a low to high transition
ENCAP10	Enable 1st capture/encoder index input to detect a high to low transition
ENCAP11	Enable 1st capture/encoder index input to detect a low to high transition
ENLSN0	Enable negative limit switch (LSN) input to detect a high to low transition
ENLSN1	Enable negative limit switch (LSN) input to detect a low to high transition
ENLSP0	Enable positive limit switch (LSP) input to detect a low to high transition
ENLSP1	Enable positive limit switch (LSP) input to detect a high to low transition
OUTPORT	Set Enable, LSP, LSN and general purpose outputs OUT#28-31
ROUT#n	Set low the output line #n
SETIO#n	Set IO line #n as input or as output
SOUT#n	Set high the output line #n
V16D = IN#n	Read input #n. V16D = input #n status
V16D = INPUT1 , ANDm	V16D = logical AND between inputs IN#25 to IN#32 status and ANDm mask
V16D = INPUT2 , ANDm	V16D = logical AND between inputs IN#33 to IN#39 status and ANDm mask
V16D = INPORT , ANDm	V16D = status of inputs Enable, LSP, LSN plus IN#36 to IN#39

6.2.4.3.7. I/O handling (firmware FBxx)

Syntax	Description
<u>!CAP</u>	Set event on capture inputs
<u>!LSN</u>	Set event on negative limit switch input
<u>!LSP</u>	Set event on positive limit switch input
<u>DISCAPI</u>	Disable 1st capture/encoder index input to detect transitions
<u>DIS2CAPI</u>	Disable 2nd capture/encoder index input to detect transitions
<u>DISLSN</u>	Disable negative limit switch (LSN) input to detect transitions
<u>DISLSP</u>	Disable positive limit switch (LSP) input to detect transitions
<u>EN2CAPI0</u>	Enable 2nd capture/encoder index input to detect a high to low transition
<u>EN2CAPI1</u>	Enable 2nd capture/encoder index input to detect a low to high transition
<u>ENCAPI0</u>	Enable 1st capture/encoder index input to detect a high to low transition
<u>ENCAPI1</u>	Enable 1st capture/encoder index input to detect a low to high transition
<u>ENLSN0</u>	Enable negative limit switch (LSN) input to detect a high to low transition
<u>ENLSN1</u>	Enable negative limit switch (LSN) input to detect a low to high transition
<u>ENLSP0</u>	Enable positive limit switch (LSP) input to detect a low to high transition
<u>ENLSP1</u>	Enable positive limit switch (LSP) input to detect a high to low transition
<u>user_var = IN(n)</u>	Read input n in the user variable user_var
<u>user_var = IN(n1, n2, n3, ...)</u>	Read inputs n1, n2, n3,... in the user variable user_var
<u>OUT(n) =value16</u>	Set the output line as specified by value16
<u>OUT(n1, n2, n3, ...) =value16</u>	Set the output lines n1 n2, n3 as specified by value16
<u>SetAsInput(n)</u>	Set the I/O line #n as an input
<u>SetAsOutput(n)</u>	Set the I/O line #n as an output
<u>SRB</u>	Set/reset bits from a MPL data
<u>STOP!</u>	Stop motion with the acceleration/deceleration set in CACC, when the programmed event occurs
<u>UPD!</u>	Update the motion mode and/or the motion parameters when the programmed event occurs
<u>WAIT!</u>	Wait until the programmed event occurs. If the command is followed by value16, the wait ends after the time interval specified in this 16-bit integer value. Value16 is measured in <u>time units</u>

6.2.4.3.8. Assignment and data transfer

Syntax	Description
V16 = label	V16 = &label
V16D = V16S	V16D = V16S
V16 = val16	V16 = val16
V16D = V32S(H)	V16D = V32S(H)
V16D = V32S(L)	V16D = V32S(L)
V16D, dm = V16S	V16D = V16S (fa)
V16D, dm = val16	V16D = val16 (fa)
V16D = (V16S), TM	V16D = (V16S) from TM memory
V16D = (V16S+), TM	V16D = (V16S) from TM memory, then V16S += 1
(V16D), TM = V16S	(V16D) from TM memory = V16S
(V16D), TM = val16	(V16D) from TM memory = val16
(V16D+), TM = V16S	(V16D) from TM memory = V16S, then V16D += 1
(V16D+), TM = val16	(V16D) from TM memory = val16, then V16D += 1
V32(H) = val16	V32(H) = val16
V32(L) = val16	V32(H) = val16
V32D(H) = V16S	V32D(H) = V16
V32D(L) = V16S	V32D(L) = V16
V16D = -V16S	V16D = -V16S
V32D = V32S	V32D = V32S
V32 = val32	V32 = val32
V32D = V16S << N	V32D = V16S left-shifted by N
V32D, dm = V32S	V32D from dm = V32S (fa)
V32D, dm = val32	V32 from dm = val32 (fa)
V32D = (V16S), TM	V32D = (V16S) from TM memory
V32D = (V16S+), TM	V32D = (V16S) from TM memory, then V16S += 2
(V16D), TM = V32S	(V16D) from TM memory = V32S
(V16D), TM = val32	(V16D) from TM memory = val32
(V16D+), TM = V32S	(V16D) from TM memory = V32S, then V16D += 2
(V16D+), TM = val32	(V16D) from TM memory = val32, then V16D += 2
V32D = -V32S	V32D = -V32S

6.2.4.3.9. Arithmetic and logic operations

Syntax

[V16 += val16](#)

[V16D += V16S](#)

[V32 += val32](#)

[V32D += V32S](#)

[V16 -= val16](#)

[V16D -= V16S](#)

[V32 -= val32](#)

[V32D -= V32S](#)

[V16 * val16 << N](#)

[V16 * val16 >> N](#)

[V16A * V16B << N](#)

[V16A * V16B >> N](#)

[V32 * V16 << N](#)

[V32 * V16 >> N](#)

[V32 * val16 << N](#)

[V32 * val16 >> N](#)

[V32=V16](#)

[PROD <<= N](#)

[V16 <<= N](#)

[V32 <<= N](#)

[PROD >>= N](#)

[V16 >>= N](#)

[V32 >>= N](#)

[SRB V16, ANDm, ORm](#)

[SRBL V16, ANDm, ORm](#)

Description

V16 = V16 + val16

V16D = V16D + V16S

V32 = V32 + val32

V32D = V32D + V32S

V16 = V16 - val16

V16D = V16D - V16S

V32 = V32 - val32

V32D = V32D - V32S

48-bit product register = (V16 * val16) >> N

48-bit product register = (V16 * val16) >> N

48-bit product register = (V16A * V16B) << N

48-bit product register = (V16A * V16B) >> N

48-bit product register = (V32 * V16) << N

48-bit product register = (V32 * V16) >> N

48-bit product register = (V32 * val16) << N

48-bit product register = (V32 * val16) >> N

Divide V32 to V16

Left shift 48-bit product register by N

Left shift V16 by N

Left shift V32 by N

Right shift 48-bit product register by N

Right shift V16 by N

Right shift V32 by N

Set / Reset Bits from V16

Set / Reset Bits from V16 (fa)

6.2.4.3.10. Multiple axis control and monitoring

Syntax

[\[A/G\] { MPL Instr}](#)

[\[A/G\] V16D = V16S](#)

Description

Send MPL instruction to [A/G]

[A/G] V16D = local V16S

[A/G] V16D, dm = V16S	[A/G] V16D = local V16S (fa)
[A/G] (V16D), TM = V16S	[A/G] (V16D), TM = local V16S
[A/G] (V16D+), TM = V16S	[A/G] (V16D), TM = local V16S, then V16D += 1
[A/G] V32D = V32S	[A/G] V32D = local V32S
[A/G] V32D, dm = V32S	[A/G] V32D = local V32S (fa)
[A/G] (V16D), TM = V32S	[A/G] (V16D), TM = local V32S
[A/G] (V16D+), TM = V32S	[A/G] (V16D), TM = local V32S, then V16D += 2
V16D = [A] V16S	Local V16D = [A] V16S
V16D = [A] V16S, dm	Local V16D = [A] V16S, dm (fa)
V16D = [A] (V16S), TM	Local V16D = [A] (V16S), dm
V16D = [A] (V16S+), TM	Local V16D = [A] (V16S), dm, then V16S += 1
V32D = [A] V32S	Local V32D = [A] V32S
V32D = [A] V32S, dm	Local V32D = [A] V32S, dm (fa)
V32D = [A] (V16S), TM	Local V32D = [A] (V16S), TM
V32D = [A] (V16S+), TM	Local V32D = [A] (V16S), TM, then V16S += 2
ADDGRID (value16_1, value16_2,...)	Add groups to the Group ID
AXISID	Set Axis ID
GROUPID (value16_1, value16_2,...)	Set GROUP ID
SETSYNC	Enable/disable synchronization between axes
SEND	Send to host the contents of a MPL variable
REMGRID (value16_1, value16_2,...)	Remove groups from the Group ID

6.2.4.3.11. Miscellaneous

Syntax	Description
<u>BEGIN</u>	BEGIN of a MPL program
<u>CANBR val16</u>	Set CAN bus baud rate
<u>CHECKSUM, TM Start, Stop, V16D</u>	V16D=Checksum between Start and Stop addresses from TM
<u>ENEEPROM</u>	Enables EEPROM usage after it was disabled by the initialization of SSI or ENDat encoders
<u>END</u>	END of a MPL program
<u>ENDINIT</u>	END of INITialization part of the MPL program
<u>FAULTR</u>	Reset FAULT status. Return to normal operation
<u>LOCKEEPROM</u>	Locks or unlocks the EEPROM write protection
<u>NOP</u>	No Operation
<u>SAVE</u>	Save setup data in the EEPROM memory
<u>SCIBR V16</u>	Set RS-232/Rs485 serial communication interface (SCI) baud rate
<u>STARTLOG V16</u>	Start the data acquisition
<u>STOPLOG</u>	Stop the data acquisition

6.2.4.3.12. On line commands

Syntax	Description
(?)GiveMeData	Ask one axis to return a 16/32 bit data from memory
TakeData	Answer to GiveMeData request
(??)GiveMeData2	Ask a group of axes to return each a 16/32 bit data from memory
TakeData2	Answer to GiveMeData2 request
GetMPLData	Ask one axis to return a MPL data
TakeData	Answer to Get MPL Data request
GetVersion	Ask one axis the firmware version
TakeVersion	Answer to Get version request
Get checksum	Ask one axis to return the checksum between 2 addresses from its MPL memory
Take checksum	Answer to Get checksum request
PING	Ask a group of axes to return their axis ID
PONG	Answer to a PING request
GETERROR	Get last error reported by slaves
SAVEERROR	Save slave error in EEPROM

Remark: The online instructions are intended only for host/master usage and cannot reside in a MPL program. Therefore their syntax is fictive, its only goal being to identify these commands.

In the [Binary Code Viewer](#) you can “emulate” a **GiveMeData** request for a MPL variable using syntax **?name** and a **GiveMeData2** request using syntax **??name**. In both cases, **name** is the MPL variable name.

In the [Command interpreter](#), you can check the value of any MPL data, by sending a **GiveMeData** request using the syntax **?name**, where name is the MPL data name. The value returned with the **TakeData** answer is displayed. Through the command interpreter you may also send a **Get checksum** request using the syntax: **CHECKSUM Start_address, Stop_address**. The value returned with **Take checksum** is displayed.

6.2.4.3.13. Obsolete Instructions

The obsolete instructions listed below have been replaced with or included as functionality in other MPL commands. The obsolete instructions may still be used with their syntax (except the ADDGRID, GROUPID and REMGRID commands), but in this case you can't benefit from the extended functionalities of their equivalents.

Obsolete syntax	Replace syntax	Remarks
ADDGRID <i>value16</i>	ADDGRID (<i>value_1, value_2,...</i>)	The binary code is identical; the syntax was changed to allow setting adding more than one group. The old syntax is no more supported
DISIO#n	–	Not required anymore. All the I/O pins are already set
ENIO#n	–	Not required anymore. All the I/O pins are already set
GROUPID <i>value16</i>	GROUPID (<i>value_1, value_2,...</i>)	The binary code is identical; the syntax was changed to allow setting adding more than one group. The old syntax is no more supported
MODE CS0		
MODE CS1	MODE CS	
MODE CS2		
MODE CS3		
MODE GS0		
MODE GS1	MODE GS	
MODE GS2		
MODE GS3		
MODE PC0		
MODE PC1	MODE PC	
MODE PC2		
MODE PC3		
MODE PE0		
MODE PE1	MODE PE	
MODE PE2		
MODE PE3		
MODE PP0		
MODE PP1	MODE PP	
MODE PP2		
MODE PP3		
MODE PPD0	–	It is seen as a particular case of electronic gearing

MODE PPD1		
MODE PPD2		
MODE PPD3		
MODE SC0		
MODE SC1	MODE SC	-
MODE SE0		
MODE SE1	MODE SE	-
MODE SP0		
MODE SP1	MODE SP	-
MODE SPD0	-	
MODE SPD1		
RAOU	-	Handled automatically
REMGRID <i>value16</i>	REMGRID(value_1, value_2,...)	The binary code is identical; the syntax was changed to allow setting adding more than one group. The old syntax is no more supported
SAOU	-	Handled automatically
SPIBR V16	-	Handled automatically
STOP0		
STOP1		
STOP2	STOP	-
STOP3		
STOP1!		
STOP2!		
STOP3!	STOP!	-
STOP3!		

6.2.5. Instructions descriptions

6.2.5.1.1. !ALPO Set event when absolute load position >

Syntax

!ALPO *value32* ! if **AbsoluteLoadPositionOver** *value32*

!ALPO *VAR32* ! if **AbsoluteLoadPositionOver** *VAR32*

Operands *VAR32*: long variable
value32: 32-bit long immediate value

Type	MPL Program	On-line
	X	X

Binary code

!ALPO *value32*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	0	0	0	1	0	0	1	0	0	1	0
0	0	0	0	0	0	1	0	0	0	1	0	1	0	0	0
LOWORD(<i>value32</i>)															
HIWORD(<i>value32</i>)															

!ALPO *VAR32*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	0	0	1	1	0	0	1	0	0	1	0
0	0	0	0	0	0	1	0	0	0	1	0	1	0	0	0
& <i>VAR32</i>															

Description Sets the event condition when the load absolute position is equal or over the specified value or the value of the specified variable. After you have programmed an event, you can do the following actions:

- Change the motion mode and/or the parameters when the event occurs, with command **UPD!**
- Stop the motion when the event occurs, with command **STOP!**
- Wait for the programmed event to occur, with command **WAIT!**

The programmed event is automatically erased when the event occurs or if the timeout for the **WAIT!** command expires.

Remark: After setting **UPD!** or **STOP!** you need to wait until the programmed event occurs using **WAIT!**, otherwise, the program will continue with the next instructions that may override the event monitoring.

Execution Activates the monitoring of the event, when load absolute position \geq *value32*, respectively *VAR32*. This operation erases a previous programmed event that has occurred.

Example

```
//Stop motion when load position >= 3 rev  
//Position feedback: 500 lines encoder (2000 counts/rev)  
!ALPO 6000; //Set event: when load absolute position is >= 3 rev  
STOP!;//Stop the motion when the event occurs  
WAIT!;//Wait until the event occurs
```

6.2.5.1.2. !ALPU Set event when absolute load position <

Syntax

!ALPU *value32* ! if **AbsoluteLoadPositionUnder** *value32*

!ALPU *VAR32* ! if **AbsoluteLoadPositionUnder** *VAR32*

Operands *VAR32*: long variable
 value32: 32-bit long immediate value

Type	MPL Program	On-line
	X	X

Binary code

!ALPU value32

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	0	0	1	0	0	0	1	0	1	0	0	0
LOWORD(<i>value32</i>)															
HIWORD(<i>value32</i>)															

!ALPU VAR32

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	0	0	1	1	0	0	0	0	0	1	1
0	0	0	0	0	0	1	0	0	0	1	0	1	0	0	0
& <i>VAR32</i>															

Description Sets the event condition when the load absolute position is equal or under the specified value or the value of the specified variable. After you have programmed an event, you can do the following actions:

- Change the motion mode and/or the parameters when the event occurs, with command **UPD!**
- Stop the motion when the event occurs, with command **STOP!**
- Wait for the programmed event to occur, with command **WAIT!**

The programmed event is automatically erased when the event occurs or if the timeout for the **WAIT!** command expires.

Remark: After setting **UPD!** or **STOP!** you need to wait until the programmed event occurs using **WAIT!**, otherwise, the program will continue with the next instructions that may override the event monitoring.

Execution Activates the monitoring of the event, when load absolute position <= *value32*, respectively *VAR32*. This operation erases a previous programmed event that has occurred.

Example

```
//Change speed command when load absolute position is <= 10 rev
```

```
//Position feedback: 500 lines encoder (2000 counts/rev)
!ALPU 20000;//Set event: when load absolute position is <= 10 rev
CSPD = 13.3333;//new slew speed command = 500[rpm]
UPD!; //execute on event
WAIT!;//Wait until the event occurs
```

6.2.5.1.3. !AMPO Set event when absolute motor position >

Syntax

!AMPO *value32* ! if **AbsoluteMotorPositionOver** *value32*

!AMPO *VAR32* ! if **AbsoluteMotorPositionOver** *VAR32*

Operands *VAR32*: long variable
value32: 32-bit long immediate value

Type	MPL Program	On-line
	X	X

Binary code

!AMPO *value32*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	0	0	0	1	0	0	1	0	0	1	0
0	0	0	0	1	0	0	1	1	0	0	0	1	0	0	0
LOWORD(<i>value32</i>)															
HIWORD(<i>value32</i>)															

!AMPO *VAR32*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	0	0	1	1	0	0	1	0	0	1	0
0	0	0	0	1	0	0	1	1	0	0	0	1	0	0	0
& <i>VAR32</i>															

Description Sets the event condition when the motor absolute position is equal or over the specified value or the value of the specified variable. After you have programmed an event, you can do the following actions:

- Change the motion mode and/or the parameters when the event occurs, with command **UPD!**
- Stop the motion when the event occurs, with command **STOP!**
- Wait for the programmed event to occur, with command **WAIT!**

The programmed event is automatically erased when the event occurs or if the timeout for the **WAIT!** command expires.

Remark: After setting **UPD!** or **STOP!** you need to wait until the programmed event occurs using **WAIT!**, otherwise, the program will continue with the next instructions that may override the event monitoring.

Execution Activates the monitoring of the event, when motor absolute position \geq *value32*, respectively *VAR32*. This operation erases a previous programmed event that has occurred.

Example

```
//Reverse when motor position >= 1rev
//Position feedback: 500 lines encoder (2000 counts/rev)
CACC = 0.1591; //acceleration rate = 500[rad/s^2]
CSPD = 40; //jog speed = 1200[rpm]
MODE SP; // set trapezoidal speed profile mode
UPD; //execute immediate
CSPD = -40; //jog speed = -1200[rpm]
!AMPO 2000; // Set event: when motor absolute position >= 1 rot
WAIT!; //Wait until the event occurs
UPD; //Update. Speed command is reversed
```

Remark: You can activate a new motion on a programmed event in 2 ways:

- Set **UPD!** command then wait the event with **WAIT!**. This will activate the new motion immediately when the event occurs
- Wait the event with **WAIT!**, then update the motion with **UPD**. This will activate the new motion with a slight delay compared with the first option

6.2.5.1.4. !AMPU Set event when absolute load position <=

Syntax

!AMPU *value32* ! if **AbsoluteMotorPositionUnder** *value32*

!AMPU *VAR32* ! if **AbsoluteMotorPositionUnder** *VAR32*

Operands *VAR32*: long variable
value32: 32-bit long immediate value

Type	MPL Program	On-line
	X	X

Binary code

!AMPU *value32*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	1	1	0	0	0	1	0	0	0
LOWORD(<i>value32</i>)															
HIWORD(<i>value32</i>)															

!AMPU *VAR32*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	0	0	1	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	1	1	0	0	0	1	0	0	0
& <i>VAR32</i>															

Description Sets the event condition when the motor absolute position is equal or under the specified value or the value of the specified variable. After you have programmed an event, you can do the following actions:

- Change the motion mode and/or the parameters when the event occurs, with command **UPD!**
- Stop the motion when the event occurs, with command **STOP!**
- Wait for the programmed event to occur, with command **WAIT!**

The programmed event is automatically erased when the event occurs or if the timeout for the **WAIT!** command expires.

Remark: After setting **UPD!** or **STOP!** you need to wait until the programmed event occurs using **WAIT!**, otherwise, the program will continue with the next instructions that may override the event monitoring.

Execution Activates the monitoring of the event, when motor absolute position <= *value32*, respectively *VAR32*. This operation erases a previous programmed event that has occurred.

Example

```
//Stop when motor position <= -3 rev
//Position feedback: 500 lines encoder (2000 counts/rev)
CACC = 0.1591; //acceleration rate = 500[rad/s^2]
CSPD = -40; //jog speed = 1200[rpm]
MODE SP;
UPD; //execute immediate
!AMPU -6000; // Set event: when motor position is < -3rev
STOP!; //Stop when the event occurs
WAIT!; //Wait until the event occurs
```

6.2.5.1.5. !CAP Set event when function of capture input

Syntax

!CAP ! if **CAP**tured triggered

Operands -

Type	MPL Program	On-line
	X	X

Binary code

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

Description Sets the event condition when the programmed transition occurs on one of the 2 capture inputs. Typically, on the capture inputs are connected the 1st and 2nd encoder index. When the programmed transition occurs on either of these inputs, the following happens:

- Motor position **APOS_MT** is captured and memorized in the MPL variable **CAPPOS**, except the case of open-loop systems, where reference position **TPOS** is captured instead
- Master position **APOS2** or load position **APOS_LD** is captured and memorized in the MPL variable **CAPPOS2**, except the case of steppers controlled open loop with an encoder on the load, when load position is captured in **CAPPOS**.

The selection between master and load position is done as follows: load position is saved in **CAPPOS2** only for the setup configurations which use different sensors for load and motor and foresee a transmission ratio between them. For all the other setup configurations, the master position is saved in **CAPPOS2**

After you have programmed an event, you can do the following actions:

- Change the motion mode and/or the parameters when the event occurs, with command **UPD!**
- Stop the motion when the event occurs, with command **STOP!**
- Wait for the programmed event to occur, with command **WAIT!**

The programmed event is automatically erased when the event occurs or if the timeout for the **WAIT!** command expires.

Remark: After setting **UPD!** or **STOP!** you need to wait until the programmed event occurs using **WAIT!**, otherwise, the program will continue with the next instructions that may override the event monitoring.

Execution Activates the monitoring of the event, when the programmed transition (low to high or high to low) occurs on the selected capture input. This operation erases a previous programmed event that has occurred.

Example

```
//Stop motion on next encoder index
```

```
ENCAP11; //Enable 1st capture input for low->high transitions
!CAP; // Set event on 1st capture (low->high transition)
STOP!;//Stop the motion when event occurs
WAIT!;//Wait until the event occurs
CPOS = CAPPOS; // new command position = captured position
CPA; //position command is absolute
MODE PP;
TUM1; //set Target Update Mode 1
UPD; //execute immediate
!MC; WAIT!; //wait for completion
```

6.2.5.1.6. !IN Set event when function of digital input

Syntax

!IN#n 0 ! if Input#n is 0
!IN#n 1 ! if Input#n is 1

Operands n: input line number (0<=n<=39)

Type	MPL Program	On-line
	X	X

Binary code

!IN#n 0															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	0	0	0	1	1	0	1	1	0	1	1
PxDATDIR															
Bit_mask															

!IN#n 1															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	0	0	0	1	1	0	1	1	0	1	0
PxDATDIR															
Bit_mask															

Description Sets the event condition when the digital input #n becomes 0, respectively 1. The condition of the input #n is tested at each slow loop sampling period. After you have programmed an event, you can do the following actions:

- Change the motion mode and/or the parameters when the event occurs, with command **UPD!**
- Stop the motion when the event occurs, with command **STOP!**
- Wait for the programmed event to occur, with command **WAIT!**

The programmed event is automatically erased when the event occurs or if the timeout for the **WAIT!** command expires.

Remark: After setting **UPD!** or **STOP!** you need to wait until the programmed event occurs using **WAIT!**, otherwise, the program will continue with the next instructions that may override the event monitoring.

Execution Activates monitoring of the event when the digital input #n becomes 0 (!IN#n 0), respectively 1 (!IN#n 1). This operation erases a previous programmed event that has occurred.

Example

```
// Start motion when digital input #36 is high
!IN#36 1; // set event when input #36 is high
//Position profile. Position feedback: 500-lines encoder
```

```
CACC = 0.3183;//acceleration rate = 1000[rad/s^2]
CSPD = 100;//slew speed = 3000[rpm]
CPOS = 20000;//position command = 10[rot]
CPA; //position command is absolute
MODE PP;
TUM1; //set Target Update Mode 1
UPD!; //execute on event
WAIT!;//Wait until the event occurs
```

6.2.5.1.7. !LSN Set event when function of LSN input

Syntax

!LSN ! if **LimitSwitchNegative** active

Operands -

Type	MPL Program	On-line
	X	X

Binary code

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

Description Sets the event condition when the programmed transition occurs at the negative limit switch input. After you have programmed an event, you can do the following actions:

- Change the motion mode and/or the parameters when the event occurs, with command **UPD!**
- Stop the motion when the event occurs, with command **STOP!**
- Wait for the programmed event to occur, with command **WAIT!**

The programmed event is automatically erased when the event occurs or if the timeout for the **WAIT!** command expires.

***Remark:** After setting **UPD!** or **STOP!** you need to wait until the programmed event occurs using **WAIT!**, otherwise, the program will continue with the next instructions that may override the event monitoring.*

Execution Activates monitoring of the event when the programmed transition occurs at the negative limit switch input. This operation erases a previous programmed event that has occurred.

Example

```
//Reverse when negative limit switch is reached
//Position feedback: 500 lines encoder (2000 counts/rev)
CACC = 0.0637; //acceleration rate = 200[rad/s^2]
CSPD = -16.6667; //jog speed = -500[rpm]
MODE SP;
UPD; //execute immediate
ENLSN1;//Enable negative limit switch for low->high transitions
!LSN; //Set event on negative limit switch(low->high transition)
WAIT!;//Wait until the event occurs
!MC; // limit switch is active -> quick stop mode active
WAIT!;// wait until the motor stops because only then the new
// motion commands are accepted
```

```
CSPD = 40;      //jog speed = 1200[rpm]
MODE SP;       //after quick stop set again the motion mode
UPD;          //execute immediate
```

6.2.5.1.8. !LSP Set event when function of LSP input

Syntax

!LSP ! if **LimitSwitchPositive** active

Operands –

Type	MPL Program	On-line
	X	X

Binary code

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

Description Sets the event condition when the programmed transition occurs at the positive limit switch input. After you have programmed an event, you can do the following actions:

- Change the motion mode and/or the parameters when the event occurs, with command **UPD!**
- Stop the motion when the event occurs, with command **STOP!**
- Wait for the programmed event to occur, with command **WAIT!**

The programmed event is automatically erased when the event occurs or if the timeout for the **WAIT!** command expires.

***Remark:** After setting **UPD!** or **STOP!** you need to wait until the programmed event occurs using **WAIT!**, otherwise, the program will continue with the next instructions that may override the event monitoring.*

Execution Activates monitoring of the event when the programmed transition occurs at the positive limit switch input. This operation erases a previous programmed event that has occurred.

Example

```
//Reverse when positive limit switch is reached
//Position feedback: 500 lines encoder (2000 counts/rev)
CACC = 0.0637; //acceleration rate = 200[rad/s^2]
CSPD = 16.6667; //jog speed = 500[rpm]
MODE SP;
UPD; //execute immediate
ENLSP1;//Enable positive limit switch for low->high transitions
!LSP; //Set event on positive limit switch(low->high transition)
WAIT!;//Wait until the event occurs
!MC; // limit switch is active -> quick stop mode active
WAIT!;// wait until the motor stops because only then the new
// motion commands are accepted
```

```
CSPD = -40;      //jog speed = -1200[rpm]
MODE SP;        //after quick stop set again the motion mode
UPD;            //execute immediate
```

6.2.5.1.9. IMC Set event when motion complete

Syntax

IMC !(set event) if **MotionComplete**

Operands -

Type	MPL Program	On-line
	X	X

Binary code

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

Description Sets the event condition when the actual motion is completed. The motion complete is set in the following conditions:

- During position control:
 - If UPGRADE.11=1, when the position reference arrives at the position to reach (commanded position) and the position error remains inside a *settle band* defined by **POSOKLIM**, for a preset *stabilize time* interval defined by **TONPOSOK**
 - If UPGRADE.11=0, when the position reference arrives at the position to reach (commanded position)
- During speed control, when the speed reference arrives at the commanded speed

The motion complete condition is reset when a new motion is started i.e. when the update command – **UPD** is executed.

Remark: *In case of steppers controlled open-loop, the motion complete condition for positioning is always set when the position reference arrives at the position to reach independently of the UPGRADE.11 status.*

Execution Activates monitoring of the event when the actual motion is completed. This operation erases a previous programmed event that has occurred.

Example

```
//Execute successive position profiles
// Position feedback: 500 lines encoder (2000 counts/rev)
POSOKLIM = 10; //Set settle band to 0.005[rot]
TONPOSOK = 10; //Set stabilize time to 0.01[s]
SRB UPGRADE, 0xFFFF, 0x0800; // motion complete with settle band
CACC = 0.3183;//acceleration rate = 1000[rad/s^2]
CSPD = 100;//slew speed = 3000[rpm]
CPOS = 20000;//position command = 10[rot]
CPR; //position command is relative
```

```
MODE PP;  
TUM1; //set Target Update Mode 1  
UPD; //execute immediate  
!MC; WAIT!; // set event and wait for motion complete  
... // start here next move
```

6.2.5.1.10. !PRO Set event when position reference >

Syntax

!PRO *value32* ! if **PositionReferenceOver** *value32*

!PRO *VAR32* ! if **PositionReferenceOver** *VAR32*

Operands *VAR32*: long variable
value32: 32-bit long immediate value

Type	MPL Program	On-line
	X	X

Binary code

!PRO *value32*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	0	0	0	1	0	0	1	0	0	0	0
0	0	0	0	0	0	1	0	1	0	1	0	1	1	1	0
LOWORD(<i>value32</i>)															
HIWORD(<i>value32</i>)															

!PRO *VAR32*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	0	0	1	1	0	0	1	0	0	0	0
0	0	0	0	0	0	1	0	1	0	1	0	1	1	1	0
& <i>VAR32</i>															

Description Sets the event condition when the position reference is equal or over the specified value or the value of the specified variable. After you have programmed an event, you can do the following actions:

- Change the motion mode and/or the parameters when the event occurs, with command **UPD!**
- Stop the motion when the event occurs, with command **STOP!**
- Wait for the programmed event to occur, with command **WAIT!**

The programmed event is automatically erased when the event occurs or if the timeout for the **WAIT!** command expires.

Remark: After setting **UPD!** or **STOP!** you need to wait until the programmed event occurs using **WAIT!**, otherwise, the program will continue with the next instructions that may override the event monitoring.

Execution Activates the monitoring of the event, when position reference \geq *value32*, respectively *VAR32*. This operation erases a previous programmed event that has occurred.

Example:

```
//Stop motion when position reference  $\geq$  3 rev
```

```
//Position feedback: 500 lines encoder (2000 counts/rev)
!PRO 6000; //Set event: when motor position reference is >= 3 rev
STOP!;//Stop the motion when the event occurs
WAIT!;//Wait until the event occurs
```

6.2.5.1.11. IPRU Set event when position reference <

Syntax

IPRU value32 ! if **PositionReferenceUnder value32**

IPRU VAR32 ! if **PositionReferenceUnder VAR32**

Operands VAR32: long variable
value32: 32-bit long immediate value

Type	MPL Program	On-line
	X	X

Binary code

IPRU value32

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	0	0	0	1	0	0	0	0	0	0	1
0	0	0	0	0	0	1	0	1	0	1	0	1	1	1	0
LOWORD(value32)															
HIWORD(value32)															

IPRU VAR32

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	0	0	1	1	0	0	0	0	0	0	1
0	0	0	0	0	0	1	0	1	0	1	0	1	1	1	0
&VAR32															

Description Sets the event condition when the position reference is equal or under the specified value or the value of the specified variable. After you have programmed an event, you can do the following actions:

- Change the motion mode and/or the parameters when the event occurs, with command **UPD!**
- Stop the motion when the event occurs, with command **STOP!**
- Wait for the programmed event to occur, with command **WAIT!**

The programmed event is automatically erased when the event occurs or if the timeout for the **WAIT!** command expires.

Remark: After setting **UPD!** or **STOP!** you need to wait until the programmed event occurs using **WAIT!**, otherwise, the program will continue with the next instructions that may override the event monitoring.

Execution Activates the monitoring of the event, when position reference <= value32, respectively VAR32. This operation erases a previous programmed event that has occurred.

Example:

```
//Stop motion when position reference >= 3 rev
```

```
//Position feedback: 500 lines encoder (2000 counts/rev)
!PRU 6000; //Set event: when position reference is >= 3 rev
STOP!;//Stop the motion when the event occurs
WAIT!;//Wait until the event occurs
```

6.2.5.1.12. !SRO Set event when speed reference >

Syntax

!SRO *value32* ! if SpeedReferenceOver *value32*

!SRO *VAR32* ! if SpeedReferenceOver *VAR32*

Operands *VAR32*: fixed variable
value32: 32-bit fixed immediate value

Type	MPL Program	On-line
	X	X

Binary code

!SRO value32

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	0	0	0	1	0	0	1	0	0	0	0
0	0	0	0	0	0	1	0	1	0	1	0	1	1	1	0
LOWORD(<i>value32</i>)															
HIWORD(<i>value32</i>)															

!SRO VAR32

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	0	0	1	1	0	0	1	0	0	0	0
0	0	0	0	0	0	1	0	1	0	1	0	1	1	1	0
& <i>VAR32</i>															

Description Sets the event condition when the speed reference is equal or over the specified value or the value of the specified variable. After you have programmed an event, you can do the following actions:

- Change the motion mode and/or the parameters when the event occurs, with command **UPD!**
- Stop the motion when the event occurs, with command **STOP!**
- Wait for the programmed event to occur, with command **WAIT!**

The programmed event is automatically erased when the event occurs or if the timeout for the **WAIT!** command expires.

Remark: After setting **UPD!** or **STOP!** you need to wait until the programmed event occurs using **WAIT!**, otherwise, the program will continue with the next instructions that may override the event monitoring.

Execution Activates the monitoring of the event, when speed reference \geq *value32*, respectively *VAR32*. This operation erases a previous programmed event that has occurred.

Example:

```
//Stop motion when speed reference  $\geq$  315 rpm
```

```
//Position feedback: 500 lines encoder (2000 counts/rev)
!SRO 10.5; //Set event: when speed reference is >= 315 rpm
STOP!;//Stop the motion when the event occurs
WAIT!;//Wait until the event occurs
```


6.2.5.1.13. !SRU Set event when speed reference <=

Syntax

!SRU value32 ! if SpeedReferenceUnder value32

!SRU VAR32 ! if SpeedReferenceUnder VAR32

Operands VAR32: fixed variable
value32: 32-bit fixed immediate value

Type	MPL Program	On-line
	X	X

Binary code

!SRU value32

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	0	0	0	1	0	0	0	0	0	0	1
0	0	0	0	0	0	1	0	1	0	1	0	1	1	1	0
LOWORD(value32)															
HIWORD(value32)															

!SRU VAR32

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	0	0	1	1	0	0	0	0	0	0	1
0	0	0	0	0	0	1	0	1	0	1	0	1	1	1	0
&VAR32															

Description Sets the event condition when the speed reference is equal or under the specified value or the value of the specified variable. After you have programmed an event, you can do the following actions:

- Change the motion mode and/or the parameters when the event occurs, with command **UPD!**
- Stop the motion when the event occurs, with command **STOP!**
- Wait for the programmed event to occur, with command **WAIT!**

The programmed event is automatically erased when the event occurs or if the timeout for the **WAIT!** command expires.

Remark: After setting **UPD!** or **STOP!** you need to wait until the programmed event occurs using **WAIT!**, otherwise, the program will continue with the next instructions that may override the event monitoring.

Execution Activates the monitoring of the event, when speed reference <= value32, respectively VAR32. This operation erases a previous programmed event that has occurred.

Example:

```
//Motor is decelerating. Start a position profile when speed
```

```
//reference < 600 rpm
//Position feedback: 500 lines encoder (2000 counts/rev)
!SRU 20; //Set event: when position reference is <= 3 rev
// prepare new motion mode
CACC = 0.3183;//acceleration rate = 1000[rad/s^2]
CSPD = 100;//slew speed = 3000[rpm]
CPOS = 20000;//position command = 10[rot]
CPR; //position command is relative
MODE PP;
TUM1; //set Target Update Mode 1
UPD!; //execute on event
WAIT!;//Wait until the event occurs
```

6.2.5.1.14. !TRO Set event when torque reference >=

Syntax

!TRO *value32* ! if TorqueReferenceOver *value32*

!TRO *VAR32* ! if TorqueReferenceOver *VAR32*

Operands *VAR32*: fixed variable
value32: 32-bit fixed immediate value

Type	MPL Program	On-line
	X	X

Binary code

!TRO *value32*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	0	0	0	1	0	0	1	0	0	0	0
0	0	0	0	0	0	1	0	1	0	1	0	1	1	1	0
LOWORD(<i>value32</i>)															
HIWORD(<i>value32</i>)															

!TRO *VAR32*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	0	0	1	1	0	0	1	0	0	0	0
0	0	0	0	0	0	1	0	1	0	1	0	1	1	1	0
& <i>VAR32</i>															

Description Sets the event condition when the current/torque reference is equal or over the specified value or the value of the specified variable. After you have programmed an event, you can do the following actions:

- Change the motion mode and/or the parameters when the event occurs, with command **UPD!**
- Stop the motion when the event occurs, with command **STOP!**
- Wait for the programmed event to occur, with command **WAIT!**

The programmed event is automatically erased when the event occurs or if the timeout for the **WAIT!** command expires.

Remark: After setting **UPD!** or **STOP!** you need to wait until the programmed event occurs using **WAIT!**, otherwise, the program will continue with the next instructions that may override the event monitoring.

Execution Activates the monitoring of the event, when current/torque reference >= *value32*, respectively *VAR32*. This operation erases a previous programmed event that has occurred.

Example:

```
// Motor will reach a hard stop. Disable control when torque
// reference > 1 A = 1984 internal current units
!TRO 1984.0; // set event when torque reference > 1 A
WAIT!; // Wait until the event occurs
AXISOFF; // disable control
```

6.2.5.1.15. !TRU Set event when torque reference <=

Syntax

!TRU *value32* ! if TorqueReferenceUnder *value32*

!TRU *VAR32* ! if TorqueReferenceUnder *VAR32*

Operands *VAR32*: fixed variable
value32: 32-bit fixed immediate value

Type	MPL Program	On-line
	X	X

Binary code

!TRU *value32*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	0	0	0	1	0	0	0	0	0	0	1
0	0	0	0	0	0	1	0	1	0	1	0	1	1	1	0
LOWORD(<i>value32</i>)															
HIWORD(<i>value32</i>)															

!TRU *VAR32*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	0	0	1	1	0	0	0	0	0	0	1
0	0	0	0	0	0	1	0	1	0	1	0	1	1	1	0
& <i>VAR32</i>															

Description Sets the event condition when the current/torque reference is equal or under the specified value or the value of the specified variable. After you have programmed an event, you can do the following actions:

- Change the motion mode and/or the parameters when the event occurs, with command **UPD!**
- Stop the motion when the event occurs, with command **STOP!**
- Wait for the programmed event to occur, with command **WAIT!**

The programmed event is automatically erased when the event occurs or if the timeout for the **WAIT!** command expires.

Remark: After setting **UPD!** or **STOP!** you need to wait until the programmed event occurs using **WAIT!**, otherwise, the program will continue with the next instructions that may override the event monitoring.

Execution Activates the monitoring of the event, when current/torque reference <= *value32*, respectively *VAR32*. This operation erases a previous programmed event that has occurred.

Example

```
// Disable control when torque reference > 1 A = 1984 IU
!TRO 1984.0; // set event when torque reference > 1 A
WAIT!;//Wait until the event occurs
AXISOFF; // disable control
```

6.2.5.1.16. !RPO Set event when relative load/motor position >

Syntax

!RPO *value32* ! if **RelativePositionOver** *value32*
!RPO *VAR32* ! if **RelativePositionOver** *VAR32*

Operands *VAR32*: long variable
value32: 32-bit long immediate value

Type	MPL Program	On-line
	X	X

Binary code

!RPO *value32*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	0	0	0	1	0	0	1	0	1	0	0
0	0	0	0	0	0	1	0	1	0	1	1	1	0	1	0
LOWORD(<i>value32</i>)															
HIWORD(<i>value32</i>)															

!RPO *VAR32*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	0	0	1	1	0	0	1	0	1	0	0
0	0	0	0	0	0	1	0	1	0	1	1	1	0	1	0
& <i>VAR32</i>															

Description Sets the event condition when the load relative position is equal or over the specified value or the value of the specified variable. The relative position is the load displacement from the beginning of the actual movement.

Remark: The origin for the relative position measurement (MPL variable **POS0**) is set function of the target update mode. Under **TUM1**, **POS0 = TPOS**. Under **TUM0**, **POS0=APOS_LD**.

After you have programmed an event, you can do the following actions:

- Change the motion mode and/or the parameters when the event occurs, with command **UPD!**
- Stop the motion when the event occurs, with command **STOP!**
- Wait for the programmed event to occur, with command **WAIT!**

The programmed event is automatically erased when the event occurs or if the timeout for the **WAIT!** command expires.

Remark: After setting **UPD!** or **STOP!** you need to wait until the programmed event occurs using **WAIT!**, otherwise, the program will continue with the next instructions that may override the event monitoring.

Execution Activates the monitoring of the event, when load relative position \geq *value32*, respectively *VAR32*. This operation erases a previous programmed event that has occurred.

Example

```
//Stop motion when after moving 3 rev  
//Position feedback: 500 lines encoder (2000 counts/rev)  
!RPO 6000; //Set event: when load relative position is >= 3 rev  
STOP!;//Stop the motion when the event occurs  
WAIT!;//Wait until the event occurs
```


6.2.5.1.17. !RPU Set event when relative load/motor position <

Syntax

!RPU *value32* ! if **RelativePositionUnder** *value32*

!RPU *VAR32* ! if **RelativePositionUnder** *VAR32*

Operands *VAR32*: long variable
value32: 32-bit long immediate value

Type	MPL Program	On-line
	X	X

Binary code

!RPU *value32*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	0	0	0	1	0	0	0	0	1	0	1
0	0	0	0	0	0	1	0	1	0	1	1	1	0	1	0
LOWORD(<i>value32</i>)															
HIWORD(<i>value32</i>)															

!RPU *VAR32*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	0	0	1	1	0	0	0	0	1	0	1
0	0	0	0	0	0	1	0	1	0	1	1	1	0	1	0
& <i>VAR32</i>															

Description Sets the event condition when the load relative position is equal or under the specified value or the value of the specified variable. The relative position is the load displacement from the beginning of the actual movement.

Remark: The origin for the relative position measurement (MPL variable **POS0**) is set function of the target update mode. Under **TUM1**, **POS0 = TPOS**. Under **TUM0**, **POS0=APOS_LD**.

After you have programmed an event, you can do the following actions:

- Change the motion mode and/or the parameters when the event occurs, with command **UPD!**
- Stop the motion when the event occurs, with command **STOP!**
- Wait for the programmed event to occur, with command **WAIT!**

The programmed event is automatically erased when the event occurs or if the timeout for the **WAIT!** command expires.

Remark: After setting **UPD!** or **STOP!** you need to wait until the programmed event occurs using **WAIT!**, otherwise, the program will continue with the next instructions that may override the event monitoring.

Execution Activates the monitoring of the event, when load relative position \leq *value32*, respectively *VAR32*. This operation erases a previous programmed event that has occurred.

Example

```
//Move negative and change speed command after 10 rev
//Position feedback: 500 lines encoder (2000 counts/rev)
!RPU 20000;//Set event: when load relative position is <= 10 rev
CSPD = 13.3333;//new slew speed command = 500[rpm]
UPD!; //execute on event
WAIT!;//Wait until the event occurs
```

6.2.5.1.18. !RT Set event after a wait time

Syntax

!RT *value32* ! if RelativeTime >= *value32*
!RT *VAR32* ! if RelativeTime >= *VAR32*

Operands *VAR32*: long variable
value32: 32-bit long immediate value

Type	MPL Program	On-line
	X	X

Binary code

!RT *value32*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	0	0	0	1	0	1	1	1	0	0	1
0	0	0	0	0	0	1	0	1	1	0	0	0	0	1	0
LOWORD(<i>value32</i>)															
HIWORD(<i>value32</i>)															

!RT *VAR32*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	0	0	1	1	0	1	1	1	0	0	1
0	0	0	0	0	0	1	0	1	1	0	0	0	0	1	0
& <i>VAR32</i>															

Description Sets the event condition when the relative time is equal or greater than the 32-bit value or the value of the specified long variable. The relative time **RTIME** is computed with formula: **RTIME = ATIME – TIME0**, where **ATIME** is a 32-bit absolute time counter, incremented by 1 at each slow loop sampling period and

TIME0 is the **ATIME** value when the wait event was set. After power on, **TIME0** is set to 0. **RTIME** is updated together with **ATIME**, at each slow loop sampling period.

Remark: ATIME and RTIME start ONLY after the execution of the ENDINIT (end of initialization) command. Therefore you should not set wait events before executing this command

After you have programmed an event monitoring you need to wait until the programmed event occurs, using the MPL command **WAIT!**. Otherwise, the program will continue with the next instructions that may override the event monitoring.

Execution Activates the monitoring of the event when system relative time >= *value32*, respectively *VAR32*. This operation erases a previous programmed event that has occurred.

Example

```
//Introduce a 100 ms delay
```

```
!RT 100;    // set event: After a wait of 100 slow-loop periods
            // 1 slow-loop period = 1ms
WAIT!;     // wait the event to occur
```

6.2.5.1.19. IMSO Set event when motor speed >=

Syntax

IMSO *value32* ! if **MotorSpeedOver** *value32*

IMSO *VAR32* ! if **MotorSpeedOver** *VAR32*

Operands *VAR32*: fixed variable
value32: 32-bit fixed immediate value

Type	MPL Program	On-line
	X	X

Binary code

IMSO *value32*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	0	0	0	1	0	0	1	0	1	1	0
0	0	0	0	0	0	1	0	0	0	1	0	1	1	0	0
LOWORD(<i>value32</i>)															
HIWORD(<i>value32</i>)															

IMSO *VAR32*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	0	0	1	1	0	0	1	0	1	1	0
0	0	0	0	0	0	1	0	0	0	1	0	1	1	0	0
& <i>VAR32</i>															

Description Sets the event condition when the motor speed is equal or over the 32-bit value or the value of the specified fixed variable. After you have programmed an event, you can do the following actions:

- Change the motion mode and/or the parameters when the event occurs, with command **UPD!**
- Stop the motion when the event occurs, with command **STOP!**
- Wait for the programmed event to occur, with command **WAIT!**

The programmed event is automatically erased when the event occurs or if the timeout for the **WAIT!** command expires.

Remark: After setting **UPD!** or **STOP!** you need to wait until the programmed event occurs using **WAIT!**, otherwise, the program will continue with the next instructions that may override the event monitoring.

Execution Activates the monitoring of the event when motor speed >= *value32*, respectively *VAR32*. This operation erases a previous programmed event that has occurred.

Example

```
//Motor is accelerating. Stop motion when motor
```

```
//speed > 600 rpm
//Position feedback: 500 lines encoder (2000 counts/rev)
!MSO 20; //Set event: when motor speed is > 600 rpm
STOP!;//Stop the motion when the event occurs
WAIT!;//Wait until the event occurs
```

6.2.5.1.20. IMSU Set event when motor speed <=

Syntax

!MSU *value32* ! if **MotorSpeedUnder** *value32*

!MSU *VAR32* ! if **MotorSpeedUnder** *VAR32*

Operands *VAR32*: fixed variable
value32: 32-bit fixed immediate value

Type	MPL Program	On-line
	X	X

Binary code

!MSU *value32*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	0	0	0	1	0	0	0	0	1	1	1
0	0	0	0	0	0	1	0	0	0	1	0	1	1	0	0
LOWORD(<i>value32</i>)															
HIWORD(<i>value32</i>)															

!MSU *VAR32*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	0	0	1	1	0	0	0	0	1	1	1
0	0	0	0	0	0	1	0	0	0	1	0	1	1	0	0
& <i>VAR32</i>															

Description Sets the event condition when the motor speed is equal or under the 32-bit value or the value of the specified fixed variable. After you have programmed an event, you can do the following actions:

- Change the motion mode and/or the parameters when the event occurs, with command **UPD!**
- Stop the motion when the event occurs, with command **STOP!**
- Wait for the programmed event to occur, with command **WAIT!**

The programmed event is automatically erased when the event occurs or if the timeout for the **WAIT!** command expires.

Remark: After setting **UPD!** or **STOP!** you need to wait until the programmed event occurs using **WAIT!**, otherwise, the program will continue with the next instructions that may override the event monitoring.

Execution Activates the monitoring of the event when motor speed <= *value32*, respectively *VAR32*. This operation erases a previous programmed event that has occurred.

Example

```
//Motor is decelerating. Start a position profile when motor
//speed < 600 rpm
```

```
//Position feedback: 500 lines encoder (2000 counts/rev)
!MSU 20; //Set event: when motor speed is < 600 rpm
// prepare new motion mode
CACC = 0.3183;//acceleration rate = 1000[rad/s^2]
CSPD = 100;//slew speed = 3000[rpm]
CPOS = 20000;//position command = 10[rot]
CPR; //position command is relative
MODE PP;
TUM1; //set Target Update Mode 1
UPD!; //execute on event
WAIT!;//Wait until the event occurs
```


6.2.5.1.21. !LSO Set event when load speed >

Syntax

!LSO *value32* ! if LoadSpeedOver *value32*
!LSO *VAR32* ! if LoadSpeedOver *VAR32*

Operands *VAR32*: fixed variable
value32: 32-bit fixed immediate value

Type	MPL Program	On-line
	X	X

Binary code

!LSO *value32*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	0	0	0	1	0	0	1	0	1	1	0
0	0	0	0	1	0	0	1	1	0	0	0	1	0	1	0
LOWORD(<i>value32</i>)															
HIWORD(<i>value32</i>)															

!LSO *VAR32*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	0	0	1	1	0	0	1	0	1	1	0
0	0	0	0	1	0	0	1	1	0	0	0	1	0	1	0
& <i>VAR32</i>															

Description Sets the event condition when the load speed is equal or over the 32-bit value or the value of the specified fixed variable. After you have programmed an event, you can do the following actions:

- Change the motion mode and/or the parameters when the event occurs, with command **UPD!**
- Stop the motion when the event occurs, with command **STOP!**
- Wait for the programmed event to occur, with command **WAIT!**

The programmed event is automatically erased when the event occurs or if the timeout for the **WAIT!** command expires.

Remark: After setting **UPD!** or **STOP!** you need to wait until the programmed event occurs using **WAIT!**, otherwise, the program will continue with the next instructions that may override the event monitoring.

Execution Activates the monitoring of the event when load speed \geq *value32*, respectively *VAR32*. This operation erases a previous programmed event that has occurred.

Example

```
//Stop motion when load speed > 600 rpm
```

```
//Load Position feedback: 500 lines encoder (2000 counts/rev)
!LSO 20; //Set event: when load speed is > 600 rpm
STOP!;//Stop the motion when the event occurs
WAIT!;//Wait until the event occurs
```

6.2.5.1.22. !LSU Set event when load speed <

Syntax

!LSU *value32* ! if LoadSpeedUnder *value32*

!LSU *VAR32* ! if LoadSpeedUnder *VAR32*

Operands *VAR32*: fixed variable
value32: 32-bit fixed immediate value

Type	MPL Program	On-line
	X	X

Binary code

!LSU *value32*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	0	0	0	1	0	0	0	0	1	1	1
0	0	0	0	1	0	0	1	1	0	0	0	1	0	1	0
LOWORD(<i>value32</i>)															
HIWORD(<i>value32</i>)															

!LSU *VAR32*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	0	0	1	1	0	0	0	0	1	1	1
0	0	0	0	1	0	0	1	1	0	0	0	1	0	1	0
& <i>VAR32</i>															

Description Sets the event condition when the load speed is equal or under the 32-bit value or the value of the specified fixed variable. After you have programmed an event, you can do the following actions:

- Change the motion mode and/or the parameters when the event occurs, with command **UPD!**
- Stop the motion when the event occurs, with command **STOP!**
- Wait for the programmed event to occur, with command **WAIT!**

The programmed event is automatically erased when the event occurs or if the timeout for the **WAIT!** command expires.

Remark: After setting **UPD!** or **STOP!** you need to wait until the programmed event occurs using **WAIT!**, otherwise, the program will continue with the next instructions that may override the event monitoring.

Execution Activates the monitoring of the event when load speed <= *value32*, respectively *VAR32*. This operation erases a previous programmed event that has occurred.

Example

```
// Start a position profile when load speed < 600 rpm
// Load Position feedback: 500 lines encoder (2000 counts/rev)
```

```
!LSU 20; //Set event: when motor speed is < 600 rpm
// prepare new motion mode
CACC = 0.3183;//acceleration rate = 1000[rad/s^2]
CSPD = 100;//slew speed = 3000[rpm]
CPOS = 20000;//position command = 10[rot]
CPR; //position command is relative
MODE PP;
TUM1; //set Target Update Mode 1
UPD!; //execute on event
WAIT!;//Wait until the event occurs
```

6.2.5.1.23. !VO Set event when variable >=

Syntax

!VO VAR32A, value32 ! if Var32AOver value32

!VO VAR32A, VAR32B ! if Var32AOver VAR32B

Operands VAR32A: fixed or long variable
 VAR32B: fixed or long variable
 value32: 32-bit fixed or long immediate value

Type

MPL Program	On-line
X	X

Binary code

!VO VAR32A, value32

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	0	0	0	1	0	0	1	0	0	0	0
&VAR32A															
LOWORD(value32)															
HIWORD(value32)															

!VO VAR32A, VAR32B

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	0	0	1	1	0	0	1	0	0	0	0
&VAR32A															
&VAR32B															

Description Sets the event condition when the selected variable (any 32-bit fixed or long MPL data) is equal or over the specified value or the value of another 32-bit variable. After you have programmed an event, you can do the following actions:

- Change the motion mode and/or the parameters when the event occurs, with command **UPD!**
- Stop the motion when the event occurs, with command **STOP!**
- Wait for the programmed event to occur, with command **WAIT!**

The programmed event is automatically erased when the event occurs or if the timeout for the **WAIT!** command expires.

Remark: After setting **UPD!** or **STOP!** you need to wait until the programmed event occurs using **WAIT!**, otherwise, the program will continue with the next instructions that may override the event monitoring.

Execution Activates the monitoring of the event when the selected variable >= value32, respectively VAR32. This operation erases a previous programmed event that has occurred.

Example

```
//Wait until master position MREF > 500 counts, then activate
//electronic gearing slave mode
!VO MREF, 500; //Set event when variable MREF is >= 500
GEAR = 1; // gear ratio
GEARMASTER = 1; // Gear ratio denominator
GEARSLAVE = 1; // Gear ratio numerator
EXTREF 2; // read master from 2nd encoder or pulse & dir
MASTERRES = 2000; // master resolution
MODE GS; //Set as slave, position mode
TUM1; //Set Target Update Mode 1
SRB UPGRADE, 0xFFFF, 0x0004;//UPGRADE.2=1 enables CACC limitation
CACC = 0.3183; //Limit maximum acceleration at 1000[rad/s^2]
UPD!; //execute on event
```

6.2.5.1.24. !VU Set event when variable <=

Syntax

!VU VAR32A, value32 ! if Var32A Under value32

!VU VAR32A, VAR32B ! if Var32A Under VAR32B

Operands VAR32A: fixed or long variable
 VAR32B: fixed or long variable
 value32: 32-bit fixed or long immediate value

Type	MPL Program	On-line
	X	X

Binary code

!VU VAR32A, value32

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	0	0	0	1	0	0	0	0	0	0	1
&VAR32A															
LOWORD(value32)															
HIWORD(value32)															

!VU VAR32A, VAR32B

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	0	0	1	1	0	0	0	0	0	0	1
&VAR32A															
&VAR32B															

Description Sets the event condition when the selected variable (any 32-bit fixed or long MPL data) is equal or under the specified value or the value of another 32-bit variable. After you have programmed an event, you can do the following actions:

- Change the motion mode and/or the parameters when the event occurs, with command **UPD!**
- Stop the motion when the event occurs, with command **STOP!**
- Wait for the programmed event to occur, with command **WAIT!**

The programmed event is automatically erased when the event occurs or if the timeout for the **WAIT!** command expires.

Remark: After setting **UPD!** or **STOP!** you need to wait until the programmed event occurs using **WAIT!**, otherwise, the program will continue with the next instructions that may override the event monitoring.

Execution Activates the monitoring of the event when the selected variable <= value32, respectively VAR32. This operation erases a previous programmed event that has occurred.

Example

```
//Wait until master position MREF < 500 counts, then activate
//electronic gearing slave mode
!VU MREF, 500; //Set event when variable MREF is <= 500
GEAR = 1; // gear ratio
GEARMASTER = 1; // Gear ratio denominator
GEARSLAVE = 1; // Gear ratio numerator
EXTREF 2; // read master from 2nd encoder or pulse & dir
MASTERRES = 2000; // master resolution
MODE GS; //Set as slave, position mode
TUM1; //Set Target Update Mode 1
SRB UPGRADE, 0xFFFF, 0x0004;//UPGRADE.2=1 enables CACC limitation
CACC = 0.3183; //Limit maximum acceleration at 1000[rad/s^2]
UPD!; //execute on event
```


6.2.5.1.25. GiveMeData/TakeData

Syntax

?VAR Ask one axis to return a 16/32 bit value from memory
 – Answer to GiveMeData request

Operands

VAR: 16/32-bit MPL data: register, parameter, variable or user variable

Type

MPL Program	On-line
—	X

Remark: The online instructions are intended only for host/master usage and cannot reside in a MPL program. Therefore their syntax is fictive, the only goal being to identify these commands.

In the [Command interpreter](#), you can check the value of any MPL data, by sending a **GiveMeData** request with syntax **?VAR**. The value returned with the **TakeData** answer, is displayed. The same syntax may be used in the [Binary Code Viewer](#) to “emulate” a **GiveMeData** request and a **TakeData** answer

Binary code

CAN Identifier: Operation Code and Axis ID (destination axis)
Data word (1): Sender Axis ID
Data word (2): Request Data Address

Description

Through **GiveMeData** command an external device can request data from one drive/motor. The requested data can be:

- A MPL data from the RAM memory for data (dm)
- A memory location from the RAM memory for MPL programs (pm)
- A memory location from the EEPROM SPI-connected memory (spi)

The dimension of the requested data is specified in the binary code through the **VT** bit: 0 – 16-bit, 1 – 32-bit. The data is identified by its memory address and type:

TypeMem	
DM	01
PM	00
SPI	10

In the expeditor address, bit **H** – the host bit – must be set to 1 only if the host sends the **GiveMeData** request via serial RS-232 link. For details, see [serial communication protocol](#) description.

The answer to a **GiveMeData** command is a **TakeData** message including the expeditor Axis ID, the address of the data returned and its value.

Remark: The **GiveMeData** and **TakeData** commands must be used only for data exchanges between 2 devices. In a multi-axis CAN bus network, the **GiveMeData** command must be sent to a single axis. If this command is sent to a group of drives, the

TakeData answers from different drives will have all the same identifier and therefore can't be correctly identified.

6.2.5.1.26. GiveMeData2/TakeData2

Syntax

??VAR Ask a group of axes to return each a 16/32 bit data from memory
 – Answer to GiveMeData2 request

Operands VAR: 16/32-bit MPL data: register, parameter, variable or user variable

Type	MPL Program	On-line
	—	X

***Remark:** The online instructions are intended only for host/master usage and cannot reside in a MPL program. Therefore their syntax is fictive, the only goal being to identify these commands.*

*In the [Binary Code Viewer](#) you can to “emulate” a **GiveMeData2** request and a **TakeData2** answer with syntax ??VAR.*

Binary code

GiveMeData2																	
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	0	1	1	0	0	1	0	0	0	0	0	TypeMem	0	0	VT		
0	0	0	0	AxisID								0	0	0	0		
Memory address from where to read data requested																	

TakeData2																	
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	0	1	0	1	0	VT	Expedito AxisID									
Memory address of data requested																	
Data requested – 16LSB																	
Data requested – 16MSB (if data is 32-bit)																	

Description Through **GiveMeData2** command an external device can request data from a group of drives/motors, using a multicast or broadcast message. The requested data can be:

- A MPL data from the RAM memory for data (dm)
- A memory location from the RAM memory for MPL programs (pm)
- A memory location from the EEPROM SPI-connected memory (spi)

The dimension of the requested data is specified in the binary code through the **VT** bit: 0 – 16-bit, 1 – 32-bit. The data is identified by its memory address and type:

TypeMem	
DM	01
PM	00
SPI	10

The answer to a **GiveMeData2** command is a **TakeData2** message including the expedito Axis ID, the address of the data returned and its value.

Remark: The **GiveMeData2** and command can be sent simultaneously to a group of drives/motors from a CAN bus network. Even if all the axes answer in the same time, the host will get the **TakeData2** answers one by one, prioritized in the ascending order of the expeditors' axis ID: axis 1 – highest priority, axis 255 – lowest priority. Hence these commands allow optimizing bus traffic, by sending for the same data, a single request to all the drives involved.

6.2.5.1.27. GetMPLData/TakeMPLData

Syntax

- Ask one axis to return a MPL data
- Answer to GetMPLData request

Operands

-

Type

MPL Program	On-line
—	X

Remark: The online instructions are intended only for host/master usage and cannot reside in a MPL program. Therefore their syntax is fictive, the only goal being to identify these commands.

Binary code

GetMPLData																	
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	0	1	0	0	VT	X	(9LSBs of &MPL data)										
0	0	0	0	AxisID								0	0	0	H		

TakeMPLData																	
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	0	1	0	1	VT	X	(9LSBs of &MPL data)										
Expeditor AxisID																	
Data requested – 16LSB																	
Data requested – 16MSB (if data is 32-bit)																	

Description Through **GetMPLData** command an external device can request a MPL data from one drive/motor. The dimension of the requested data is specified in the binary code through the **VT** bit: 0 – 16-bit, 1 – 32-bit. The MPL data is identified by its address. **GetMPLData** instruction uses a 9-bit **short address** for the MPL data. Bit value **X** specifies the address range:

X	Address range of MPL data
0	from 0x0200 to 0x03FF
1	from 0x0800 to 0x09FF

In the expeditor address, bit **H** – the host bit – must be set to 1 only if the host sends the **GetMPLData** request via serial RS-232 link. For details, see [serial communication protocol](#) description.

The answer to a **GetMPLData** command is a **TakeMPLData** message including the expeditor Axis ID, the address of the MPL data returned and its value.

The **GetMPLData** and **TakeMPLData** commands are optimized for requests of MPL data (registers, parameters, variables). For this type of data exchanges, **GetMPLData** and **TakeMPLData** provide shorter messages and occupy less communication bandwidth compared with **GiveMeData** and **TakeData**.

Remark: The **GetMPLData** and **TakeMPLData** commands must be used only for data exchanges between 2 devices. In a multi-axis CAN bus network, the **GetMPLData** command must be sent to a single axis. If this command is sent to a group of drives, the **TakeMPLData** answers from different drives will have all the same identifier and therefore can't be correctly identified.

6.2.5.1.28. GetVersion/TakeVersion

Syntax

- Ask one axis to return the firmware version
- Answer to GetVersion request

Operands

–

Type

MPL Program	On-line
—	X

Remark: The online instructions are intended only for host/master usage and cannot reside in a MPL program. Therefore their syntax is fictive, the only goal being to identify these commands.

Binary code

GetVersion

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

TakeVersion

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	0	0	0	0	0	1
ASCII code of first 2 digits of the firmware ID															
ASCII code of last digit + revision letter of the firmware ID															

Description

Through **GetVersion** command an external device can request the firmware version from one drive/motor. In the expeditor address, bit **H** – the host bit – must be set to 1 only if the host sends the **GetVersion** request via serial RS-232 link. For details, see [serial communication protocol](#) description.

The firmware version has the form: FxyzA, where xyz is the firmware number (3 digits) and A is the firmware revision. The answer to a **GetVersion** command is a **TakeVersion** message including the expeditor Axis ID and the ASCII code of 4 characters: 3 digits for the firmware number + 1 letter for the firmware revision.

*Remark: The **GetVersion** and **TakeVersion** commands must be used only between 2 devices. In a multi-axis CAN bus network, the **GetMPLData** command must be sent to a single axis. If this command is sent to a group of drives, the **TakeVersion** answers from different drives will have all the same identifier and therefore can't be correctly identified.*

6.2.5.1.29. GetChecksum/TakeChecksum

Syntax

Checksum Start, End

Ask one axis to return the checksum between Start and Stop addresses from its MPL memory

Answer to **GetChecksum** request

Operands

Start. 16-bit unsigned integer value representing the checksum start address

End: 16-bit unsigned integer value representing the checksum end address

Type	MPL Program	On-line
	—	X

Remark: The online instructions are intended only for host/master usage and cannot reside in a MPL program. Therefore their syntax is fictive, the only goal being to identify these commands.

In the [Command interpreter](#), you can get a checksum between 2 MPL program addresses by sending a **GetChecksum** request with the syntax:

Checksum Start, End

Where, **Start, End** represent the start and end addresses for the checksum. The value returned with the **TakeChecksum** answer, is displayed.

CHECKSUM		Start		Stop											
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	1	0	0	TypeMem	1	0	0	0	0	0
& Host															
Start address															
Stop address															

Answer to CHECKSUM		Start		Stop											
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	1	0	0	TypeMem	1	0	0	0	0	0
value16															

Description Through **GetChecksum** command an external device can check the integrity of the data saved in a drive/motor EEPROM or RAM memory. The memory type is selected automatically function of the start and the end addresses.

In the expeditor address, bit **H** – the host bit – must be set to 1 only if the host sends the **GiveMeData** request via serial RS-232 link. For details, see [serial communication protocol](#) description.

The answer to a **GetChecksum** command is a **TakeChecksum**, which returns the expeditor axis ID, and the checksum result i.e. the sum modulo 65536 of all the memory locations between the start and the end addresses.

6.2.5.1.30. = Assign a 16-bit value to a MPL variable or a memory location

Syntax

<code>VAR16D = label</code>	set VAR16D to value of a label
<code>VAR16D = value16</code>	set VAR16D to value16
<code>VAR16D = VAR16S</code>	set VAR16D to VAR16S value
<code>VAR16D = VAR32S(L)</code>	set VAR16D to VAR32S(L) value
<code>VAR16D = VAR32S(H)</code>	set VAR16D to VAR32S(H) value
<code>VAR16D, dm = value16</code>	set VAR16D from dm to value16

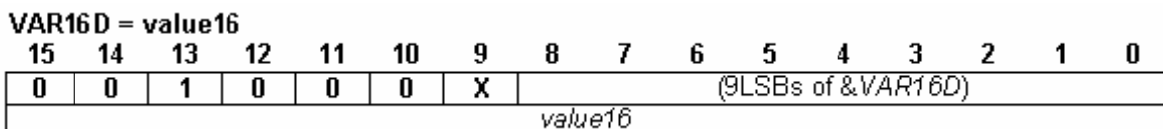
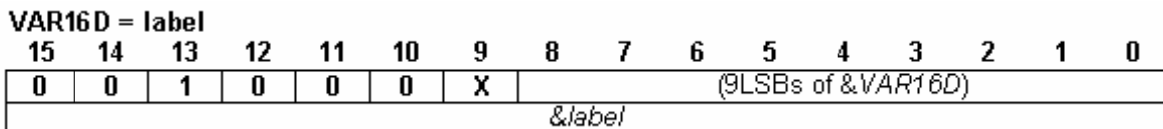
$VAR16D, dm = VAR16S$	set $VAR16D$ from dm to $VAR16S$
$VAR16D = (VAR16S), TypeMem$	set $VAR16D$ to $\&(VAR16S)$ from TM
$VAR16D = (VAR16S+), TypeMem$	set $VAR16D$ to $\&(VAR16S)$ from TM , then $VAR16S += 1$
$(VAR16D), TypeMem = value16$	set $\&(VAR16D)$ from TM to $value16$
$(VAR16D), TypeMem = VAR16S$	set $\&(VAR16D)$ from TM to $VAR16S$
$(VAR16D+), TypeMem = value16$	set $\&(VAR16D)$ from TM to $value16$, then $VAR16D += 1$
$(VAR16D+), TypeMem = VAR16S$	set $\&(VAR16D)$ from TM to $VAR16S$, then $VAR16D += 1$
$VAR32D(L) = value16$	set $VAR32D$ low word to $value16$
$VAR32D(L) = VAR16S$	set $VAR32D(L)$ to $VAR16S$ value
$VAR32D(H) = value16$	set $VAR32D$ high word to $value16$
$VAR32D(H) = VAR16S$	set $VAR32D(H)$ to $VAR16S$ value

Legend: D (destination), S (source).

- Operands**
- label*: 16-bit address of a MPL instruction label
 - value16*: 16-bit integer immediate value
 - VAR16x*: integer variable VAR16x
 - VAR32x(L)*: the low word of VAR32x long variable
 - VAR32x(H)*: the high word of VAR32x long variable
 - Dm*: data memory operand
 - TypeMem*: memory operand.
 - $(VAR16x)$: contents of variable VAR16x, representing a 16-bit address of a variable

Type	MPL Program	On-line
	X	X

Binary code



VAR16D = VAR16S

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	0	X	(9LSBs of &VAR16D)								
&VAR16S															

VAR16D = VAR32S(L)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	0	X	(9LSBs of &VAR16D)								
&VAR32S															

VAR16D = VAR32S(H)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	0	X	(9LSBs of &VAR16D)								
&VAR32S + 1															

VAR16D, dm = value16

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0
&VAR16D															
value16															

VAR16D, dm = VAR16S

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	0	0	0	0	0	0	0	1	0	1	0	0
&VAR16D															
&VAR16S															

VAR16D = (VAR16S), TypeMem

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	0	0	0	1	1	0	0	0	TypeMem	0	0	0
&VAR16S															
&VAR16D															

VAR16D = (VAR16S+), TypeMem

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	0	0	0	1	0	0	0	0	TypeMem	0	0	0
&VAR16S															
&VAR16D															

(VAR16D), TypeMem = value16

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	0	0	0	0	1	0	1	0	TypeMem	0	0	0
&VAR16D															
value16															

(VAR16D), TypeMem = VAR16S

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	0	0	0	0	1	0	1	1	TypeMem	0	0	0
&VAR16D															
&VAR16S															

(VAR16D+), TypeMem = value16															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	0	0	0	0	0	0	1	0	TypeMem	0	0	0
&VAR16D															
value16															

(VAR16D+), TypeMem = VAR16S															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	0	0	0	0	0	0	1	1	TypeMem	0	0	0
&VAR16D															
&VAR16S															

VAR32D(L) = value16																	
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
0	0	1	0	0	0	X	(9LSBs of &VAR32D)										
value16																	

VAR32D(L) = VAR16S																	
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
0	0	1	0	1	0	X	(9LSBs of &VAR32D)										
&VAR16S																	

VAR32D(H) = value16																	
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
0	0	1	0	0	0	X	(9LSBs of &VAR32D+1)										
value16																	

VAR32D(H) = VAR16S																	
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
0	0	1	0	1	0	X	(9LSBs of &VAR32D+1)										
&VAR16S																	

Description Assigns a 16-bit value to a MPL variable or a memory location. The options are:

The destination is 16-bit MPL variable and the source is: a 16-bit immediate value, a label, 16-bit MPL variable, high or low part of a 32-bit MPL variable or the contents of a memory location whose address is indicated by a 16-bit MPL variable (a pointer).

The destination is a memory location whose address is indicated by a 16-bit MPL variable (a pointer) and the source is: a 16-bit immediate value or a 16-bit MPL variable.

The destination is the high or low part of a 32-bit MPL variable and the source is: a 16-bit immediate value or a 16-bit MPL variable.

If the pointer variable is followed by a + sign, after the assignment, it is incremented by 1. The memory location can be of 3 types: RAM for data (dm), RAM for MPL programs (pm), EEPROM SPI-connected for MPL programs (spi).

TypeMem	
DM	01
PM	00
SPI	10

Some instructions use a 9-bit **short address** for the destination variable. Bit value **X** specifies the destination address range:

X	Address range of MPL data
0	from 0x0200 to 0x03FF
1	from 0x0800 to 0x09FF

All predefined or user-defined MPL data are inside these address ranges, hence these instructions can be used without checking the variables addresses. However, considering future developments, the MPL also includes assignment instructions using a **full address** where the destination address can be any 16-bit value. In this case destination variable is followed by “,dm”.

Execution Copies a 16-bit value from the source to the destination

Example1

```
int Var1;
Label1:    // Label1 = MPL program address
...
Var1 = Label1;
```

Before instruction

Label1 0x1234
Var1 x

After instruction

Label1 0x1234
Var1 0x1234

Example2

```
int Var1;
...
Var1 = 26438;
```

Before instruction

Var1 x

After instruction

Var1 26438

Example3

```
int Var1, Var2;  
...  
Var2 = Var1;
```

Before instruction		After instruction	
Var2	0x56AB	Var2	0x56AB
Var1	x	Var1	0x56AB

Example4

```
int Var1;  
long Var3;  
...  
Var1 = Var3(L);
```

Before instruction		After instruction	
Var3	0x56ABCD98	Var3	0x56ABCD98
Var1	x	Var1	0xCD98

Example5

```
int Var1;  
long Var3;  
....  
Var1 = Var3(H);
```

Before instruction		After instruction	
Var3	0x56ABCD98	Var3	0x56ABCD98
Var1	x	Var1	0x56AB

Example6

```
int Var1;  
...  
Var1, dm = 3321;
```

Before instruction

Var1 x

After instruction

Var1 3321

Example7

```
int Var1, Var2;  
...  
Var1, dm = Var2;
```

Before instruction

Var1 0x0A01

Var2 x

After instruction

Var1 0x0A01

Var2 0x0A01

Example8

```
int Var1, pVar2;  
...  
Var1 = (pVar2), dm;
```

Before instruction

pVar2 0x0A01

Data memory

0x0A01 0x1234

Var1 x

After instruction

pVar2 0x0A01

Data memory

0x0A01 0x1234

Var1 0x1234

Example9

```
int Var1, pVar2;  
...  
Var1 = (pVar2+), dm;
```

Before instruction

pVar2 0x0A01

Data memory

0x0A01 0x1234

Var1 x

After instruction

pVar2 0x0A02

Data memory

0x0A02 0x0014

Var1 0x0014

Example10

```
int pVar1;  
...  
(pVar1), spi = 0x5422;
```

Before instruction

pVar1 0x5100

SPI data memory

0x1100 x

After instruction

pVar1 0x5100

SPI data memory

0x1100 0x5422

(SPI memory offset is 0x4000,
i.e. SPI addr = var.addr -
0x4000)

Example11

```
int pVar1;  
...  
(pVar1+), spi = 0x5422;
```

Before instruction

pVar1 0x5100

SPI data memory

0x1100 x

After instruction

pVar1 0x5101

SPI data memory

0x1100 0x5422

*(SPI memory offset is 0x4000,
i.e. SPI addr = var.addr -
0x4000)*

Example12

```
int pVar1, Var2;  
...  
(pVar1), pm = Var2;
```

Before instruction

pVar1 0x8200

Var2 0xA987

pm data
memory

0x8200 x

After instruction

pVar1 0x8200

Var2 0xA987

pm data
memory

0x8200 0xA987

Example13

```
int pVar1, Var2;  
...  
(pVar1+), pm = Var2;
```

Before instruction

pVar1 0x8200

Var2 0xA987

pm *data*
memory

0x8200 x

After instruction

pVar1 0x8201

Var2 0xA987

pm *data*
memory

0x8200 0xA987

Example14

```
long Var5;  
...  
Var5(H) = 0xAA55 ;
```

Before instruction

Var5 0x12344321

After instruction

Var5 0xAA554321

Example15

```
long Var5;  
...  
Var5(L) = 0xAA55 ;
```

Before instruction

Var5 0x12344321

After instruction

Var5 0x1234AA55

Example16

```
int Var1;  
long Var5;  
...  
Var5(H) = Var1;
```

Before instruction		After instruction	
Var1	0x7711	Var1	0x7711
Var5	0x12344321	Var5	0x77114321

Example17

```
int Var1;  
long Var5;  
...  
Var5(L) = Var1;
```

Before instruction		After instruction	
Var1	0x7711	Var1	0x7711
Var5	0x12344321	Var5	0x12347711

6.2.5.1.31. = Read digital input(s) and assign a 16-bit MPL variable with their value (Firmware version F $_{xx}$)

Syntax

$VAR16D = IN\#n$ read input # n into $VAR16D$

$VAR16D = INPUT1, ANDm$ read inputs $IN\#25$ to $IN\#32$ into $VAR16D$ with $ANDm$

$VAR16D = INPUT2, ANDm$ read input $IN\#33$ to $IN\#39$ into $VAR16D$ with $ANDm$

$VAR16D = INPORT, ANDm$ read Enable, LSP, LSN and $IN\#36$ to $IN\#39$ into $VAR16D$ with $ANDm$

Operands

$Var16D$: integer variable

$IN\#n$: the source is input n ($0 \leq n \leq 39$)

$INPUT1$: the source is inputs #25 to #32

$INPUT2$: the source is inputs #33 to #39

$ANDm$: a 16-bit mask for filtering the inputs. A logical AND is performed between the inputs read and the $ANDm$ mask

$INPORT$: the source is 7 inputs: Enable, LSP, LSN, #39, #38, #37 and #36

Type

MPL Program	On-line
X	X

Binary code

$VAR16D = IN\#n$

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	0	X	(9LSBs of &VAR16D)								
$PxDATDIR$															
0	0	0	0	0	0	0	0	Bit_mask							

$VAR16D = INPUT1, ANDm$

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	0	X	(9LSBs of &VAR16D)								
0	1	1	1	0	0	0	0	1	0	0	1	0	1	0	1
0	0	0	0	0	0	0	0	$ANDm$							

$VAR16D = INPUT2, ANDm$

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	0	X	(9LSBs of &VAR16D)								
0	1	1	1	0	0	0	0	1	0	0	1	0	1	1	0
0	0	0	0	0	0	0	0	$ANDm$							

VAR16D = INPORT, ANDm

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	0	X	(9LSBs of &VAR16D)								
ANDm															

Description Read digital input(s) and assign a 16-bit MPL variable with their value. When a single input is read (IN#n), the destination variable is set to 0 when the input is 0 (low) and to a non-zero value when the input is 1 (high). When multiple inputs are read with **INPUT1** or **INPUT2**, each of the 8LSB of the destination variable shows one input status: 0 – input is 0 (low), 1 – input is 1 (high) after passing through the ANDm mask. The inputs are assigned from bit 0 to 7 in ascending order (IN#25 – bit 0, IN#26 – bit 1, etc.). **INPORT** works like **INPUT1** / **INPUT2** except the bit assignment in the destination variable: Enable – bit 15, LSN – bit 14, LSP – bit 13, #39 – bit 3, #38 – bit 2, #37 – bit 1, #36 – bit 0.

In MPL the I/O lines are numbered: #0 to #39. Each product has a specific number of inputs and outputs, therefore only a part of the 40 I/O lines is used. The I/O numbering is common for all the products; hence each product has its own list of available I/Os.

These instructions use a 9-bit **short address** for the destination variable. Bit 9 value **X** specifies the destination address range:

X	Address range of MPL data
0	from 0x0200 to 0x03FF
1	from 0x0800 to 0x09FF

Execution Read input(s) and set their status in reserved bits from the destination

PxDATDIR & Bit_mask		
#n	PxDATDIR	Bit_mask
#0	0x7098	0x0001
#1	0x7098	0x0002
#2	0x7098	0x0004
#3	0x7098	0x0008
#4	0x7098	0x0010
#5	0x7098	0x0020
#6	0x7098	0x0040
#7	0x7098	0x0080
#8	0x709A	0x0001
#9	0x709A	0x0002
#10	0x709A	0x0004
#11	0x709A	0x0008
#12	0x709A	0x0010
#13	0x709A	0x0020
#14	0x709A	0x0040
#15	0x709A	0x0080
#16	0x709C	0x0001
#17	0x709C	0x0002
#18	0x709C	0x0004
#19	0x709C	0x0008

#n	PxDATDIR	Bit_mask
#20	0x709C	0x0010
#21	0x709C	0x0020
#22	0x709C	0x0040
#23	0x709C	0x0080
#24	0x709E	0x0001
#25	0x7095	0x0001
#26	0x7095	0x0002
#27	0x7095	0x0004
#28	0x7095	0x0008
#29	0x7095	0x0010
#30	0x7095	0x0020
#31	0x7095	0x0040
#32	0x7095	0x0080
#33	0x7096	0x0001
#34	0x7096	0x0002
#35	0x7096	0x0004
#36	0x7096	0x0008
#37	0x7096	0x0010
#38	0x7096	0x0020
#39	0x7096	0x0040

Example1

```
int Var1;
...
Var1 = IN#14;
```

Before instruction		After instruction	
IN#14 status	1	IN#14 status	1
Var1	x	Var1	0x0040
<i>Bit#6 of Var1 has logic value of IN#14. Remaining bits are set to 0.</i>			

Example2

```
int Var1;
...
Var1 = INPUT1, 0x00E7;
```

Before instruction									After instruction								
IN #	32	31	30	29	28	27	26	25	IN#	32	31	30	29	28	27	26	25
Status	0	1	1	0	1	1	0	1	Status	0	1	1	0	1	1	0	1
Var1					x				Var1								0x0065

IN#		32	31	30	29	28	27	26	25	
Inputs status		0	1	1	0	1	1	0	1	Bitwise operation
And_Mask		1	1	1	0	0	1	1	1	
Var1		0	1	1	0	0	1	0	1	

Example3

```
int Var1;
...
Var1 = INPUT2, 0x00E7;
```

Before instruction								After instruction							
IN#	39	38	37	36	35	34	33	IN#	39	38	37	36	35	34	33
Status	1	0	0	1	1	0	1	Status	1	0	0	1	1	0	1
Var1				x				Var1					0x0085		

IN#	39	38	37	36	35	34	33	
Inputs status	1	0	0	1	1	0	1	Bitwise operation
And_Mask	1	1	0	0	1	1	1	
Var1	1	0	0	0	1	0	1	

Example4

```
int Var1;
...
Var1 = INPORT, 0xE00F;
```

Before instruction								After instruction							
IN#	Enable	LSN	LSP	39	38	37	36	IN#	Enable	LSN	LSP	39	38	37	36
status	1	0	1	1	0	1	1	status	1	0	1	1	0	1	1
Var1					X			Var1							0xA00B

6.2.5.1.32. = Read digital input(s) and assign a 16-bit MPL variable with their value (Firmware version FBxx)

Syntax

VAR16D = *IN*(*n1*, *n2*,...) Read input **n1**, **n2** into VAR16D

Operands

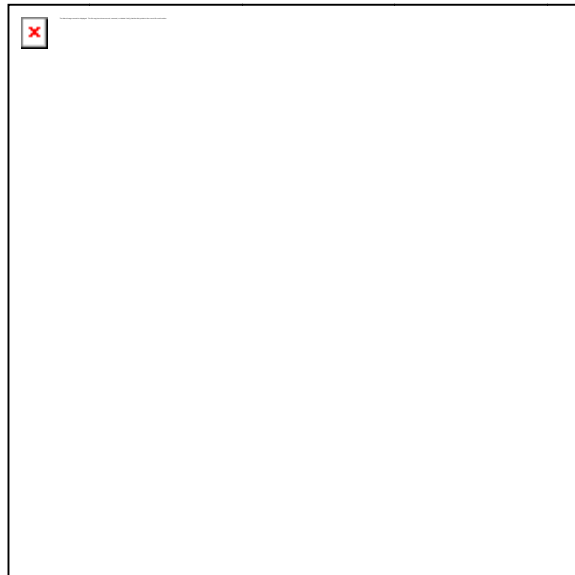
Var16D: integer variable

IN(*n1*, *n2*,...): the source inputs *n1*, *n2*, ...

Type

MPL Program	On-line
X	X

Binary code



Description

Read digital input(s) and assign a 16-bit MPL variable with their value. When a single input is read, **IN**(*n*), the destination variable is set to 0 when the input *n* is 0 (low) and to a non-zero value when the input *n* is 1 (high). When multiple inputs are read, **IN**(*n1*, *n2*,...), each bit of the destination variable shows one input status: 0 – input is 0 (low), 1 – input is 1 (high).

In MPL the input lines are numbered from 0 to 15. Each product has a specific number of inputs, therefore only a part of the 15 input lines is used.

These instructions use a 9-bit **short address** for the destination variable. Bit 9 value **X** specifies the destination address range:

X	Address range of MPL data
0	from 0x0200 to 0x03FF
1	from 0x0800 to 0x09FF

Execution

Read input(s) and set their status in the corresponding bits from the destination.

Example1

```
int Var1;  
...  
Var1 = IN(4);
```

Before instruction		After instruction	
IN(4) status	1	IN(4) status	1
Var1	x	Var1	0x0010
<i>Bit#4 of Var1 has logic value of IN(4). Remaining bits are set to 0.</i>			

Example1

```
int Var1;  
...  
Var1 = IN(4, 9);
```

Before instruction		After instruction	
IN(4) status	1	IN(4) status	1
IN(9) status	1	IN(9) status	1
Var1	x	Var1	0x0210
<i>Bit#4 of Var1 has logic value of IN(4). Bit#9 of Var1 has logic value of IN(9). Remaining bits are set to 0.</i>			

6.2.5.1.33. = Assign a 32-bit value to a MPL variable or a memory location

Syntax

$VAR32D = value32$ set $VAR32D$ to $value32$
 $VAR32D = VAR32S$ set $VAR32D$ to $VAR32S$ value
 $VAR32D = VAR16S \ll N$ set $VAR32D$ to $VAR16S \ll N$
 $VAR32D, DM = value32$ set long $VAR32D$ from DM to $value32$
 $VAR32D, DM = VAR32S$ set long $VAR32D$ from DM to $VAR32S$
 $VAR32D = (VAR16S), TypeMem$ set $VAR32D$ to $\&(VAR16S)$ from TM
 $VAR32D = (VAR16S+), TypeMem$ set $VAR32D$ to $\&(VAR16S)$ from TM , then $VAR16S += 2$
 $(VAR16D), TypeMem = value32$ set $\&(VAR16D)$ from TM to $value32$
 $(VAR16D), TypeMem = VAR32S$ set $\&(VAR16D)$ from TM to $VAR32S$
 $(VAR16D+), TypeMem = value32$ set $\&(VAR16D)$ from TM to $value32$, then $VAR16D += 2$
 $(VAR16D+), TypeMem = VAR32S$ set $\&(VAR16D)$ from TM to $VAR32S$, then $VAR16D += 2$

Operands $value32$: 32-bit long immediate value
 $VAR32x$: long variable $VAR32x$
 DM : data memory operand
 $TypeMem$: memory operand. One of dm (0x1), pm (0x0) or spl (0x2) values
 $(VAR16x)$: contents of variable $VAR16x$, representing a 16-bit address of a variable

Type

MPL Program	On-line
X	X

Binary code

$VAR32D = value32$

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	X	(9LSBs of $\&VAR32D$)								
							LOWORD($value32$)								
							HIWORD($value32$)								

VAR32D = VAR32S

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	1	X	(9LSBs of &VAR32D)								
&VAR32S															

VAR32D = VAR16S << N

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	1	0	0	1	0	1	1	N (0 ≤ N ≤ 16)				
&VAR32D															
&VAR16S															

VAR32D, dm = value32

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	0	0	0	0	0	0	0	0	0	1	0	1
&VAR32D															
LOWORD(value32)															
HIWORD(value32)															

VAR32D, dm = VAR32S

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	0	0	0	0	0	0	0	1	0	1	0	1
&VAR32D															
&VAR32S															

VAR32D = (VAR16S), TypeMem

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	0	0	0	1	1	0	0	0	TypeMem	0	0	1
&VAR16S															
&VAR32D															

VAR32D = (VAR16S+), TypeMem

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	0	0	0	1	0	0	0	0	TypeMem	0	0	1
&VAR16S															
&VAR32D															

(VAR16D), TypeMem = value32

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	0	0	0	0	1	0	1	0	TypeMem	0	0	1
&VAR16D															
LOWORD(value32)															
HIWORD(value32)															

(VAR16D), TypeMem = VAR32S

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	0	0	0	0	1	0	1	1	TypeMem	0	0	1
&VAR16D															
&VAR32S															

(VAR16D+), TypeMem = value32															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	0	0	0	0	0	0	1	0	TypeMem	0	1	
&VAR16D															
LOWORD(value32)															
HIWORD(value32)															

(VAR16D+), TypeMem = VAR32S															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	0	0	0	0	0	0	1	1	TypeMem	0	1	
&VAR16D															
&VAR32S															

Description Assigns a 32-bit value to a MPL variable or a memory location. The options are:

The destination is 32-bit MPL variable and the source is: a 32-bit immediate value, a 32-bit MPL variable, a 16-bit MPL variable left shifted by 0 to 16 bits, or the contents of 2 consecutive memory locations with the lower address indicated by a 16-bit MPL variable (a pointer). Left shift is done with sign extension.

The destination is 2 memory locations with the lower address indicated by a 16-bit MPL variable (a pointer) and the source is: a 32-bit immediate value or a 32-bit MPL variable.

If the pointer variable is followed by a + sign, after the assignment, it is incremented by 2. The memory location can be of 3 types: RAM for data (dm), RAM for MPL programs (pm), EEPROM SPI-connected for MPL programs (spi).

TypeMem	
DM	01
PM	00
SPI	10

Some instructions use a 9-bit **short address** for the destination variable. Bit value **X** specifies the destination address range:

X	Address range of MPL data
0	from 0x0200 to 0x03FF
1	from 0x0800 to 0x09FF

All predefined or user-defined MPL data are inside these address ranges, hence these instructions can be used without checking the variables addresses. However, considering future developments, the MPL also includes assignment instructions using a **full address** where the destination address can be any 16-bit value. In this case destination variable is followed by “**dm**”.

Execution Copies a 32-bit value from the source to the destination

Example1

```
long Var1;  
...  
Var1 = 0x1122AABB;
```

Before instruction

Var1 x

After instruction

Var1 0x1122AABB

Example2

```
long Var1, Var2;  
...  
Var1 = Var2;
```

Before instruction

Var2 0xAABC1234

Var1 x

After instruction

Var2 0xAABC1234

Var1 0xAABC1234

Example3

```
int Var1;  
long Var2;  
...  
Var2 = Var1 << 4;
```

Before instruction

Var1 0x9876

Var2 x

After instruction

Var1 0x9876

Var2 0x00098760

Example4

```
long Var1;  
...  
Var1, dm = 0x1122AABB;
```

Before instruction

Var1 x

After instruction

Var1 0x1122AABB

Example5

```
long Var1, Var2;  
...  
Var1, dm = Var2;
```

Before instruction

Var2 0xAABC1234

Var1 x

After instruction

Var2 0xAABC1234

Var1 0xAABC1234

Example6

```
long Var1;  
int pVar2;  
...  
Var1 = (pVar2), dm;
```

Before instruction

pVar2 0x96AB

Data memory

0x96AB 0x1234

0x96AC 0xABCD

Var1 x

After instruction

pVar2 0x96AB

Data memory

0x96AB 0x1234

0x96AC 0xABCD

Var1 0xABCD1234

Example7

```
long Var1;  
int pVar2;  
...  
Var1 = (pVar2+), dm;
```

Before instruction**After instruction**

pVar2	0x0A02	pVar2	0x0A04
<i>Data memory</i>		<i>Data memory</i>	
0x0A02	0x1234	0x0A02	0x1234
0x0A03	0xABCD	0x0A03	0xABCD
Var1	x	Var1	0xABCD1234

Example8

```
int pVar1;
...
(pVar1), spi = 0x5422AFCD;
```

Before instruction		After instruction	
pVar1	0x5100	pVar1	0x5100
<i>SPI memory</i>	<i>data</i>	<i>SPI data memory</i>	
0x1100	x	0x1100	0xAFCD
0x1101	x	0x1101	0x5422
<i>(SPI memory offset is 0x4000, i.e. SPI addr = var.addr - 0x4000)</i>			

Example9

```
int pVar1;
long Var2;
...
(pVar1), pm = Var2;
```

Before instruction		After instruction	
pVar1	0x8200	pVar1	0x8200
Var2	0xA98711EF	Var2	0xA98711EF

<i>pm data memory</i>		<i>pm data memory</i>	
0x8200	x	0x8200	0x11EF
0x8201	x	0x8201	0xA987

Example10

```
int pVar1;
...
(pVar1+), pm = 0x5422AFCD;
```

Before instruction		After instruction	
pVar1	0x8200	pVar1	0x8202
<i>pm data memory</i>		<i>pm data memory</i>	
0x8200	x	0x8200	0xAFCD
0x8201	x	0x8201	0x5422

Example11

```
int pVar1;
long Var2;
...
(pVar1+), pm = Var2;
```

Before instruction		After instruction	
pVar1	0x8200	pVar1	0x8202
Var2	0xA98711E F	Var2	0xA98711EF
<i>Pm data memory</i>		<i>pm data memory</i>	
0x8200	x	0x8200	0x11EF
0x8201	x	0x8201	0xA987

Remark: When destination is 2 consecutive memory locations and the source is an immediate value, the MPL compiler checks the type and the dimension of the immediate value and based on this generates the

binary code for a 16-bit or a 32-bit data transfer. Therefore if the immediate value has a decimal point, it is automatically considered as a fixed value. If the immediate value is outside the 16-bit integer range (-32768 to +32767), it is automatically considered as a long value. However, if the immediate value is inside the integer range, in order to execute a 32-bit data transfer it is necessary to add the suffix **L** after the value, for example: **200L** or **-1L**.

Examples:

```
user_var = 0x29E;           // write CPOS address in pointer variable user_var
(user_var),dm = 1000000;    // write 1000000 (0xF4240) in the CPOS parameter i.e
                            // 0x4240 at address 0x29E and 0xF at address 0x29F
(user_var+),dm = -1;       // write -1 (0xFFFF) in CPOS(L). CPOS(H) remains
                            // unchanged. CPOS is (0xFFFF) i.e. 1048575,
                            // and user_var is incremented by 2
user_var = 0x29E;           // write CPOS address in pointer variable user_var
(user_var+),dm = -1L;       // write -1L long value (0xFFFFFFFF) in CPOS i.e.
                            // CPOS(L) = 0xFFFF and CPOS(H) = 0xFFFF,
                            // user_var is incremented by 2
user_var = 0x2A0;           // write CSPD address in pointer variable user_var
(user_var),dm = 1.5;        // write 1.5 (0x18000) in the CSPD parameter i.e
                            // 0x8000 at address 0x2A0 and 0x1 at address 0x2A1
```


VAR16D = [Axis] (VAR16S+), TypeMem

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	1	1	0	1	0	0	0	0	TypeMem	0	0	0
0	0	0	0	Axis								0	0	0	0
&VAR16S															
&VAR16D															

Description Assigns a 16-bit local MPL variable with data got from another axis. The source on the remote axis can be: a 16-bit MPL variable or a memory location whose address is indicated by a 16-bit MPL variable (a pointer) from the remote axis. If the pointer variable is followed by a + sign, after the assignment, it is incremented by 1.

***Remark:** If the MPL variables from the remote axis are user variables, these must be declared in the local axis too. Moreover, for correct operation, these variables must have the same address in both axes, which means that they must be declared on each axis on the same position. Typically, when working with data transfers between axes, it is advisable to establish a block of user variables that may be the source, destination or pointer of data transfers, and to declare these data on all the axes as the first user variables. This way you can be sure that these variables have the same address on all the axes.*

The memory location can be of 3 types: RAM for data (dm), RAM for MPL programs (pm), EEPROM SPI-connected for MPL programs (spi).

TypeMem	
DM	01
PM	00
SPI	10

One instruction uses a 9-bit **short address** for the source variable. Bit value **X** specifies the destination address range:

X	Address range of MPL data
0	from 0x0200 to 0x03FF
1	from 0x0800 to 0x09FF

All predefined or user-defined MPL data are inside these address ranges, hence these instructions can be used without checking the variables addresses. However, considering future developments, the MPL also includes assignment instructions using a **full address** where the destination address can be any 16-bit value. In this case destination variable is followed by “,dm”.

Execution Copies a 16-bit value from the remote source to the local destination

Example1

```
int VarLoc, VarExt;
...
VarLoc = [15]VarExt;
```

Before instruction

After instruction

VarLoc on local axis	x	VarLoc on local axis	0x1234
VarExt on axis 15	0x1234	VarExt on axis 15	0x1234

Example2

```
int VarLoc, VarExt;
...
VarLoc = [15]VarExt, dm;
```

Before instruction

VarLoc on local axis	x
VarExt on axis 15	0x1234

After instruction

VarLoc on local axis	0x1234
VarExt on axis 15	0x1234

Example3

```
int VarLoc, pVarExt;
...
VarLoc = [15](pVarExt), dm;
```

Before instruction

pVarExt on axis 15	0x1234
At dm address 0x1234 on axis 15	0xFEDC
VarLoc on local axis	x

After instruction

pVarExt on axis 15	0x1234
At dm address 0x1234 on axis 15	0xFEDC
VarLoc on local axis	0xFEDC

Example4

```
int VarLoc, pVarExt;
...
VarLoc = [15](pVarExt+), dm;
```

Before instruction

pVarExt on axis 15 0x1234

At dm address 0x1234 0xFEDC
on axis 15

VarLoc on local axis x

After instruction

pVarExt on axis 15 0x1235

At dm address 0x1234 0xFEDD
on axis 15

VarLoc on local axis 0xFEDC

Description Assigns a 32-bit local MPL variable with data got from another axis. The source on the remote axis can be: a 32-bit MPL variable or 2 consecutive memory location with lower address indicated by a 16-bit MPL variable (a pointer) from the remote axis. If the pointer variable is followed by a + sign, after the assignment, it is incremented by 2.

Remark: *If the MPL variables from the remote axis are user variables, these must be declared in the local axis too. Moreover, for correct operation, these variables must have the same address in both axes, which means that they must be declared on each axis on the same position. Typically, when working with data transfers between axes, it is advisable to establish a block of user variables that may be the source, destination or pointer of data transfers, and to declare these data on all the axes as the first user variables. This way you can be sure that these variables have the same address on all the axes.*

The memory location can be of 3 types: RAM for data (dm), RAM for MPL programs (pm), EEPROM SPI-connected for MPL programs (spi).

TypeMem	
DM	01
PM	00
SPI	10

One instruction uses a 9-bit **short address** for the source variable. Bit value **X** specifies the destination address range:

X	Address range of MPL data
0	from 0x0200 to 0x03FF
1	from 0x0800 to 0x09FF

All predefined or user-defined MPL data are inside these address ranges, hence these instructions can be used without checking the variables addresses. However, considering future developments, the MPL also includes assignment instructions using a **full address** where the destination address can be any 16-bit value. In this case destination variable is followed by “,dm”.

Execution Copies a 32-bit value from the remote source to the local destination

Example1

```
long VarLoc, VarExt;
...
VarLoc = [15]VarExt;
```

Before instruction		After instruction	
VarLoc on local axis	x	VarLoc on local axis	0x1234ABCD
VarExt on axis 15	0x1234ABCD	VarExt on axis 15	0x1234ABCD

Example2

```

long VarLoc, VarExt;
...
VarLoc = [15]VarExt, dm;

```

Before instruction

VarLoc on local axis x

VarExt on axis 15 0xF0E1A2B3

After instruction

VarLoc on local axis 0xF0E1A2B3

VarExt on axis 15 0xF0E1A2B3

Example3

```

long VarLoc;
int pVarExt;
...
VarLoc = [15](pVarExt), dm;

```

Before instruction

pVarExt on axis 15 0x1234

At dm address 0x1234 0xFEDC
on axis 15

At dm address 0x1235 0x2233
on axis 15

VarLoc on local axis x

After instruction

pVarExt on axis 15 0x1234

At dm address 0x1234 0xFEDC
on axis 15

At dm address 0x1235 0x2233
on axis 15

VarLoc on local axis 0x2233FEDC

Example4

```

long VarLoc;
int pVarExt;
...
VarLoc = [15](pVarExt+), dm;

```

Before instruction

After instruction

pVarExt on axis 15	0x1234	pVarExt on axis 15	0x1236
At dm address 0x1234	0xFEDC	At dm address 0x1234	0xFEDF
on axis 15		on axis 15	
At dm address 0x1235	0x2233	At dm address 0x1235	0x2233
on axis 15		on axis 15	
VarLoc on local axis	X	VarLoc on local axis	0x2233FEDC

6.2.5.1.36. = Assign a 16-bit value to a MPL variable or a memory location from another axis or group of axes

Syntax

<i>[Axis/Group]</i> VAR16D = VAR16S	<i>[A/G]</i> VAR16D = local VAR16S
<i>[Axis/Group]</i> VAR16D, dm = VAR16S	<i>[A/G]</i> VAR16D, dm = local VAR16S
<i>[Axis/Group]</i> (VAR16D), TypeMem = VAR16S	<i>[A/G]</i> &(VAR16D), TM = local VAR16S
<i>[Axis/Group]</i> (VAR16D+), TypeMem = VAR16S	<i>[A/G]</i> &(VAR16D), TM = local VAR16S, then V16D+=1

Operands

VAR16x: integer variable VAR16x

Axis/Group:

- An integer 1 to 255 representing an Axis ID
- G followed by an integer 1 to 8 representing one of the 8 groups
- B for a broadcast to all axes

dm: data memory operand

TypeMem: memory operand. One of dm (0x1), pm (0x0) or spi (0x2) values

(VAR16x): contents of variable VAR16x, representing a 16-bit address of a variable

Type

MPL Program	On-line
X	X

Binary code

[Axis/Group] VAR16D = VAR16S

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	0	X	(9LSBs of &VAR16D)								
0	0	0	A/G	Axis/Group								0	0	0	0
&VAR16S															

[Axis/Group] VAR16D, dm = VAR16S

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	1	0	0	0	0	0	0	1	0	1	0	0
0	0	0	A/G	Axis/Group								0	0	0	0
&VAR16D															
&VAR16S															

[Axis/Group] (VAR16D), TypeMem = VAR16S

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	1	0	0	0	1	0	1	1	TypeMem	0	0	
0	0	0	A/G	Axis/Group								0	0	0	0
&VAR16D															
&VAR16S															

[Axis/Group] (VAR16D+), TypeMem = VAR16S

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	1	0	0	0	0	0	1	1	TypeMem	0	0	
0	0	0	A/G	Axis/Group								0	0	0	0
&VAR16D															
&VAR16S															

Description Sends the value of a 16-bit local MPL variable to another axis or group of axes. The remote destination can be a 16-bit MPL variable or a memory location whose address is indicated by a 16-bit MPL variable (a pointer) from the remote axis/axes. If the pointer variable is followed by a + sign, after the assignment, it is incremented by 1. In the binary code, Axis/Group represents either an Axis ID (if A/G=0) or a Group ID (if A/G = 1). A transmission with Group ID can be:

- For all the axes from a single group, if one bit from the 8-bit Group ID is 1
- A broadcast to all the axes, if the Group ID = 0

Remark: If the MPL variables from the remote axis are user variables, these must be declared in the local axis too. Moreover, for correct operation, these variables must have the same address in both axes, which means that they must be declared on each axis on the same position. Typically, when working with data transfers between axes, it is advisable to establish a block of user variables that may be the source, destination or pointer of data transfers, and to declare these data on all the axes as the first user variables. This way you can be sure that these variables have the same address on all the axes.

The memory location can be of 3 types: RAM for data (dm), RAM for MPL programs (pm), EEPROM SPI-connected for MPL programs (spi).

TypeMem	
DM	01
PM	00
SPI	10

One instruction uses a 9-bit **short address** for the source variable. Bit value **X** specifies the destination address range:

X	Address range of MPL data
0	from 0x0200 to 0x03FF
1	from 0x0800 to 0x09FF

All predefined or user-defined MPL data are inside these address ranges, hence these instructions can be used without checking the variables addresses. However, considering future developments, the MPL also includes assignment instructions using a **full address** where the destination address can be any 16-bit value. In this case destination variable is followed by “,dm”.

Execution Copies a 16-bit value from a local source to a remote destination

Example1

```
int VarLoc, VarExt;
...
[15]VarExt, dm = VarLoc;
```

Before instruction

VarLoc on local axis 0x1234
 VarExt on axis 15 x

After instruction

VarLoc on local axis 0x1234
 VarExt on axis 15 0x1234

Example2

```
int VarLoc, pVarExt;
...
[G8](pVarExt), dm = VarLoc;
```

Before instruction

VarLoc on local axis 0xFEDC

After instruction

VarLoc on local axis 0xFEDC

pVarExt on all axes of 0x1234
group 8 is the same

At dm address 0x1234 x
on all axes of group 8

pVarExt on all axes of 0x1234
group 8 is the same

At dm address 0x1234 0xFEDC
on all axes of group 8

Example3

```
int VarLoc, pVarExt;  
...  
[G8](pVarExt+), dm = VarLoc;
```

Before instruction

VarLoc on local axis 0xFEDC

pVarExt on all axes of 0x1234
group 8 is the same

At dm address 0x1234 x
on all axes of group 8

After instruction

VarLoc on local axis 0xFEDD

pVarExt on all axes of 0x1235
group 8 is the same

At dm address 0x1234 0xFEDC
on all axes of group 8

6.2.5.1.37. = Assign a 32-bit value to a MPL variable or a memory location from another axis or group of axes

Syntax

[Axis/Group] VAR32D = VAR32S [A/G] long VAR32D = local VAR32S
 [Axis/Group] VAR32D, DM = VAR32S [A/G] long VAR32D, DM = local VAR32S
 [Axis/Group] (VAR16D), TypeMem = VAR32S [A/G] &(VAR16D), TM = local VAR32S
 [Axis/Group] (VAR16D+), TypeMem = VAR32S [A/G] &(VAR16D), TM = local VAR32S, then V1DS+=2

Operands

VAR32x: long variable VAR32x

Axis/Group:

- An integer 1 to 255 representing an Axis ID
- G followed by an integer 1 to 8 representing one of the 8 groups
- B for a broadcast to all axes

dm: data memory operand

TypeMem: memory operand. One of dm (0x1), pm (0x0) or spi (0x2) values

(VAR16x): contents of variable VAR16x, representing a 16-bit address of a variable

Type

MPL Program	On-line
X	X

Binary code

[Axis/Group] VAR32D = VAR32S

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	0	1	1	1	1	X	(9LSBs of &VAR32D)												
0	0	0	A/G	Axis/Group								0	0	0	0				
&VAR32S																			

[Axis/Group] VAR32D, dm = VAR32S

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	1	0	0	0	0	0	0	1	0	1	0	1
0	0	0	A/G	Axis/Group								0	0	0	0
& VAR32D															
& VAR32S															

[Axis/Group] (VAR16D), TypeMem = VAR32S

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	1	0	0	0	1	0	1	1	TypeMem	0	1	
0	0	0	A/G	Axis/Group								0	0	0	0
& VAR16D															
& VAR32S															

[Axis/Group] (VAR16D+), TypeMem = VAR32S

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	1	0	0	0	0	0	1	1	TypeMem	0	1	
0	0	0	A/G	Axis/Group								0	0	0	0
& VAR16D															
& VAR32S															

Description Sends the value of a 32-bit local MPL variable to another axis or group of axes. The remote destination can be a 32-bit MPL variable or 2 consecutive memory locations with lower address indicated by a 16-bit MPL variable (a pointer) from the remote axis/axes. If the pointer variable is followed by a + sign, after the assignment, it is incremented by 2. In the binary code, Axis/Group represents either an Axis ID (if A/G=0) or a Group ID (if A/G = 1). A transmission with Group ID can be:

- For all the axes from a single group, if one bit from the 8-bit Group ID is 1
- A broadcast to all the axes, if the Group ID = 0

Remark: If the MPL variables from the remote axis are user variables, these must be declared in the local axis too. Moreover, for correct operation, these variables must have the same address in both axes, which means that they must be declared on each axis on the same position. Typically, when working with data transfers between axes, it is advisable to establish a block of user variables that may be the source, destination or pointer of data transfers, and to declare these data on all the axes as the first user variables. This way you can be sure that these variables have the same address on all the axes.

The memory location can be of 3 types: RAM for data (dm), RAM for MPL programs (pm), EEPROM SPI-connected for MPL programs (spi).

TypeMem	
DM	01
PM	00
SPI	10

One instruction uses a 9-bit **short address** for the source variable. Bit value **X** specifies the destination address range:

X	Address range of MPL data
0	from 0x0200 to 0x03FF
1	from 0x0800 to 0x09FF

All predefined or user-defined MPL data are inside these address ranges, hence these instructions can be used without checking the variables addresses. However, considering future developments, the MPL also includes assignment instructions using a **full address** where the destination address can be any 16-bit value. In this case destination variable is followed by “,dm”.

Execution Copies a 32-bit value from a local source to a remote destination

Example1

```
long VarLoc, VarExt;
...
[15]VarExt = VarLoc;
```

Before instruction		After instruction	
VarLoc on local axis	0x1234ABCD	VarLoc on local axis	0x1234ABCD
VarExt on axis 15	x	VarExt on axis 15	0x1234ABCD

Example2

```
long VarLoc, VarExt;
...
[15]VarExt, dm = VarLoc;
```

Before instruction		After instruction	
VarLoc on local axis	0xF0E1A2B3	VarLoc on local axis	0xF0E1A2B3
VarExt on axis 15	x	VarExt on axis 15	0xF0E1A2B3

Example3

```
long VarLoc;  
int pVarExt;  
...  
[15](pVarExt), dm = VarLoc;
```

Before instruction		After instruction	
VarLoc on local axis	0x2233FEDC	VarLoc on local axis	0x2233FEDC
pVarExt on axis 15	0x1234	pVarExt on axis 15	0x1234
At dm address 0x1234 x on axis 15		At dm address 0x1234	0xFEDC on axis 15
At dm address 0x1235 x on axis 15		At dm address 0x1235	0x2233 on axis 15

Example4

```
long VarLoc;  
int pVarExt;  
...  
[15](pVarExt+), dm = VarLoc;
```

Before instruction		After instruction	
VarLoc on local axis	0x2233FEDC	VarLoc on local axis	0x2233FEDC
pVarExt on axis 15	0x1234	pVarExt on axis 15	0x1236
At dm address 0x1234 x on axis 15		At dm address 0x1234	0xFEDE on axis 15
At dm address 0x1235 x on axis 15		At dm address 0x1235	0x2233 on axis 15



6.2.5.1.38. MPL Send MPL command

Syntax [Axis/Group] {MPL command; }

Operands Axis/Group:

- An integer 1 to 255 representing an Axis ID
- G followed by an integer 1 to 8 representing one of the 8 groups
- B for a broadcast to all axes

MPL command: any single axis MPL instruction

Type	MPL Program	On-line
	X	—

Binary code

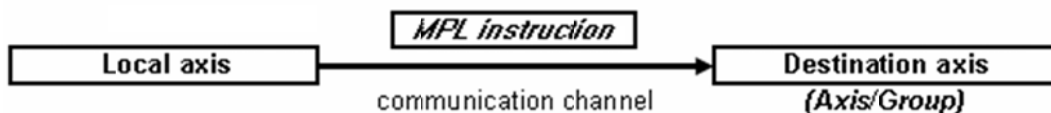
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	0	1	0	length(MLI) - 1								
0	0	0	A/G	Axis/Group								0	0	0	0
Instruction word 1 (operation code)															
Instruction word 2 (data)															
...															
Instruction word (length(MLI)) (data)															

Description When an axis executing a MPL program encounters this instruction, sends the MPL command specified between { } to another axis or group of axes. At destination, the command is executed as any other on-line MPL command received via a communication channel. In the binary code, Axis/Group represents either an Axis ID (if A/G=0) or a Group ID (if A/G = 1). A transmission with Group ID can be:

- For all the axes from a single group, if one bit from the 8-bit Group ID is 1
- A broadcast to all the axes, if the Group ID = 0

Remark: You may specify between { } multiple commands, separated by semicolons “;”. The MPL compiler splits them in single commands, each having the above binary code. The single commands are sent in the same order as set in the command between { }

Execution Send the “MPL Command” between { } to the destination



Example

```
[G1]{CPOS=2000;} ; // send a new CPOS command to all axes from group 1
[G1]{UPD} ;           // send an update command to all the axes from group 1
                       // all axes from group 1 will start to move simultaneously
[B]{STOP} ;          // broadcast a STOP command to all axes from the network
```

6.2.5.1.39. -- Assign a MPL variable with the negate of another MPL variable

Syntax

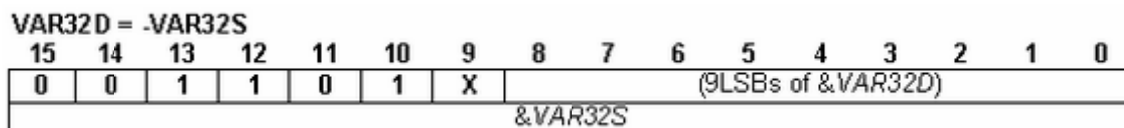
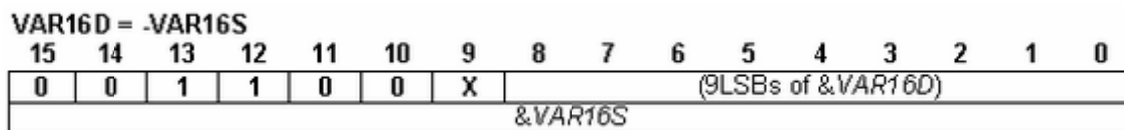
`VAR16D = -VAR16S` set `VAR16D` to `-VAR16S` value

`VAR32D = -VAR32S` set `VAR32D` to `-VAR32S` value

Operands `VAR16D`: destination integer variable
 `VAR16S`: source integer variable
 `VAR32D`: destination long/integer variable
 `VAR32S`: source long/integer variable

Type	MPL Program	On-line
	X	X

Binary code



Description Assigns a 16-bit / 32-bit variable with the negate of another 16-bit / 32-bit variable. The instruction uses a 9-bit **short address** for the destination variable. Bit value **X** specifies the destination address range:

X	Address range of MPL data
0	from 0x0200 to 0x03FF
1	from 0x0800 to 0x09FF

Execution Copies the negate of a 16-bit or 32-bit value from the source to the destination

Example

```
int Var1;
long Var2;
...
Var1 = - Var1;
Var2 = -Var2;
```

Before instruction		After instruction	
Var1	1256	Var1	-1256
Var2	-224500	Var2	224500

6.2.5.1.40. +

Syntax

`VAR16 += value16` add to `VAR16` `value16`
`VAR16D += VAR16S` add to `VAR16D` the `VAR16S` value
`VAR32 += value32` add to `VAR32` `value32`
`VAR32D += VAR32S` add to `VAR32D` the `VAR32S` value

Operands

`VAR16D`: destination integer variable
`VAR16S`: source integer variable
`VAR32D`: destination long/fixed variable
`VAR32S`: source long/fixed variable
`value16`: 16-bit immediate integer value
`value32`: 32-bit immediate long value

Type

MPL Program	On-line
X	X

Binary code

VAR16 += value16

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	X	(9LSBs of &VAR16)								
<code>value16</code>															

VAR16D += VAR16S

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	X	(9LSBs of &VAR16D)								
&VAR16S															

VAR32 += value32

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	1	X	(9LSBs of &VAR32)								
LOWORD(<code>value32</code>)															
HIWORD(<code>value32</code>)															

VAR32D += VAR32S

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	X	(9LSBs of &VAR32D)								
&VAR32S															

Description

Adds a 16-bit / 32-bit immediate value or the value of the 16-bit / 32-bit source variable to the 16-bit / 32-bit destination variable. The instructions use a 9-bit **short address** for the destination variable. Bit value **X** specifies the destination address range:

X	Address range of MPL data
0	from 0x0200 to 0x03FF
1	from 0x0800 to 0x09FF

Execution Destination variable = destination variable + immediate value or source variable

Example

```
int Var1, Var2, Var3;  
long Var10, Var11, Var12;  
...  
Var1 += 125;  
Var3 += Var2;  
Var10 += 128000;  
Var12 += Var11;
```

Before instruction		After instruction	
Var1	1256	Var1	1381
Var2	-22450	Var2	-22450
Var3	22500	Var3	50
Var10	-1201	Var10	126799
Var11	25	Var11	25
Var12	12500	Var12	12525

6.2.5.1.41. -

Syntax

- VAR16 -= value16* subtract from *VAR16 value16*
- VAR16D -= VAR16S* subtract from *VAR16D VAR16S* value
- VAR32 -= value32* subtract from *VAR32 value32*
- VAR32D -= VAR32S* subtract from *VAR32D VAR32S* value

Operands

- VAR16D*: destination integer variable
- VAR16S*: source integer variable
- VAR32D*: destination long/fixe d variable
- VAR32S*: source long/fixe d variable
- value16*: 16-bit immediate integer value

value32: 32-bit immediate long value

Type	MPL Program	On-line
	X	X

Binary code

VAR16 := value16

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	X	(9LSBs of &VAR16)								
<i>value16</i>															

VAR16D := VAR16S

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	0	X	(9LSBs of &VAR16D)								
&VAR16S															

VAR32 := value32

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	X	(9LSBs of &VAR32)								
LOWORD(<i>value32</i>)															
HIWORD(<i>value32</i>)															

VAR32D := VAR32S

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	1	X	(9LSBs of &VAR32D)								
&VAR32S															

Description Subtracts a 16-bit / 32-bit immediate value or the value of the 16-bit / 32-bit source variable from the 16-bit / 32-bit destination variable. The instructions use a 9-bit **short address** for the destination variable. Bit value **X** specifies the destination address range:

X	Address range of MPL data
0	from 0x0200 to 0x03FF
1	from 0x0800 to 0x09FF

Execution Destination variable = destination variable - immediate value or source variable

Example

```
int Var1, Var2, Var3;
long Var10, Var11, Var12;
...
Var1 -= 125;
Var3 -= Var2;
Var10 -= 128000;
Var12 -= Var11;
```

Before instruction	After instruction
---------------------------	--------------------------

Var1	1256	Var1	1131
Var2	-22450	Var2	-22450
Var3	22500	Var3	44950
Var10	-1201	Var10	-129201
Var11	25	Var11	25
Var12	12500	Var12	12475

6.2.5.1.42. * Multiply

Syntax

$VAR16 * VALUE16 \gg N$	$PROD = (VAR16 * value16) \gg N$
$VAR16 * VALUE16 \ll N$	$PROD = (VAR16 * value16) \ll N$
$VAR16A * VAR16B \gg N$	$PROD = (VAR16A * VAR16B) \gg N$
$VAR16A * VAR16B \ll N$	$PROD = (VAR16A * VAR16B) \ll N$
$VAR32 * VALUE16 \gg N$	$PROD = (VAR32 * value16) \gg N$
$VAR32 * VALUE16 \ll N$	$PROD = (VAR32 * value16) \ll N$
$VAR32 * VAR16 \gg N$	$PROD = (VAR32 * VAR16) \gg N$
$VAR32 * VAR16 \ll N$	$PROD = (VAR32 * VAR16) \ll N$

Operands *VAR16D*: destination integer variable
 VAR16S: source integer variable
 VAR32D: destination long/fixe variable
 VAR32S: source long/fixe variable
 value16: 16-bit immediate integer value
 value32: 32-bit immediate long value
 N: result shift factor

Type

MPL Program	On-line
X	X

Binary code

VAR16 * VALUE16 >> N

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	1	1	0	0	0	0	0	0	<i>N (0<N<15)</i>			
&VAR16															
VALUE16															

VAR16 * VALUE16 << N

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	1	1	0	0	0	0	1	0	<i>N (0<N<15)</i>			
&VAR16															
VALUE16															

VAR16A * VAR16B >> N

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	1	1	0	0	1	0	0	0	<i>N (0<N<15)</i>			
&VAR16A															
&VAR16B															

VAR16A * VAR16B << N

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	1	1	0	0	1	0	1	0	<i>N (0<N<15)</i>			
&VAR16A															
&VAR16B															

VAR32 * VALUE16 >> N

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	1	1	0	1	0	0	0	0	<i>N (0<N<15)</i>			
&VAR32															
VALUE16															

VAR32 * VALUE16 << N

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	1	1	0	1	0	0	1	0	<i>N (0<N<15)</i>			
&VAR32															
VALUE16															

VAR32 * VAR16 >> N

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	1	1	0	1	1	0	0	0	<i>N (0<N<15)</i>			
&VAR32															
&VAR16															

VAR32 * VAR16 << N

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	1	1	0	1	1	0	1	0	<i>N (0<N<15)</i>			
&VAR32															
&VAR16															

Description Multiplies 2 operands. The first operand (left one) can be a 16-bit or 32-bit MPL variable. The second operand (right one) can be a 16-bit immediate value or another 16-bit MPL variable. The result is saved in a dedicated **48-bit product register** left or right shifted by 0 to 15 bits. The MPL long variables **PROD** and **PRODH** show the 32LSB respectively the 32 MSB of the product register.

Execution **Product register** = (first operand * second operand) shifted to left or right with the specified number of bits

Example1

```
int Var1;  
long var2;  
...  
Var1 * 0x125;  
Var2 = PROD;
```

Before instruction		After instruction	
Var1	0x1256	Var1	0x1256
Product register	x	Product register	0x00000014FC6E
Var2	x	Var2	0x0014FC6E

Example2

```
int Var1;  
long Var2;  
...  
Var1 * 0x125 << 12;  
Var2 = PRODH;
```

Before instruction		After instruction	
Var1	0x1256	Var1	0x1256
Product register	x	Product register	0x00014FC6E000
Var2	X	Var2	0x00014FC6

Example3

```
int Var2, Var3;  
long Var4;  
...  
Var2 * Var3 >> 4;  
Var4 = PROD;
```

Before instruction		After instruction	
Var2	0x1256	Var2	0x1256

Var3	0x125	Var3	0x125
Product register	x	Product register	0x00000014FC6
Var4	x	Var4	0x00014FC6

Example4

```
int Var2, Var3;
long Var7;
...
Var2 * Var3 << 8;
Var7 = PROD(H);
```

Before instruction		After instruction	
Var2	0x1256	Var2	0x1256
Var3	0x125	Var3	0x125
Product register	x	Product register	0x000014FC6E00
Var7	x	Var7	0x000014FC

Example5

```
long Var1, Var2;
...
Var1 * 0x125;
Var2 = PROD;
```

Before instruction		After instruction	
Var1	0x001256AB	Var1	0x1256
Product register	x	Product register	0x000014FD31B7
Var2	x	Var2	0x14FD31B7

Example6

```
long Var1, Var2;
...
```

```
Var1 * 0x125 << 12;
```

```
Var2 = PROD(H);
```

Before instruction

After instruction

Var1 0x001256AB

Var1 0x1256

Product register x

Product register 0x014FD31B7000

Var2 x

Var2 0x014FD31B

Example7

```
long Var2, Var9;  
int Var3;  
...  
Var2 * Var3 >> 4;  
Var9 = PROD(H);
```

Before instruction		After instruction	
Var2	0x001256AB	Var2	0x001256AB
Var3	0x125	Var3	0x125
Product register	x	Product register	0x0000014FD31B
Var9	x	Var9	0x0000014F

Example8

```
long Var2, Var9;  
int Var3;  
...  
Var2 * Var3 << 8;  
Var9 = PROD;
```

Before instruction		After instruction	
Var2	0x001256AB	Var2	0x001256AB
Var3	0x125	Var3	0x125
Product register	x	Product register	0x0014FD31B700
Var9	X	Var9	0xFD31B700

6.2.5.1.43. /**Syntax**

VAR32 /= *VAR16* divide *VAR32* with *VAR16*

Operands *VAR16*: the divisor, integer variable

VAR32: the dividend, fixed variable

Type	MPL Program	On-line
	X	X

Binary code

VAR32/=VAR16

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	1	0	0	0	0	0	0	0	0	0	1
&VAR32															
&VAR16															

Description The left operand – the dividend is divided by the right operand – the divisor, and the result is saved in the left operand. The dividend / quotient is a 32-bit fixed variable and the divisor a 16-bit integer variable.

Execution Left operand = left operand / right operand

Example

```
fixed var1; // Define fixed variable user_1
int var2; // Define integer variable user_2
var1 = 11.0;
var2 = 3;
var1 /= var2;
```

Before instruction		After instruction	
Var1	11.0 (0xB0000)	Var1	3.6666 (0x3AAAA)
Var2	3	Var2	3

6.2.5.1.44. >>

Syntax

- VAR16 >>= N shift VAR16 right by N
- VAR32 >>= N shift VAR32 right by N
- PROD >>= N shift PROD (product reg.) right by N

Operands VAR16: integer variable
 VAR32: long or fixed variable
 PROD: 48-bit product register
 N: shift factor

Type

MPL Program	On-line
X	X

Binary code

VAR16 >>= N

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	1	0	0	0	0	0	0	0	N (0<N<15)			
&VAR16															

VAR32 >>= N

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	1	0	0	1	0	0	0	0	N (0<N<15)			
&VAR32															

PROD >>= N

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	1	0	0	0	1	0	0	0	N (0<N<15)			
& PROD															

Description The operand is right shifted with the specified number of bits (N). High order bits are sign-extended and the low order bits are lost. If the operand is **PROD**, the entire 48-bit product register is right shifted.

Execution Variable = Value of variable shifted to right with N bits

Example1 int Var1;
...
Var1 >>= 4;

Before instruction

Var1 0x1256

After instruction

Var1 0x0125

Example2 long Var1;
...
Var1 >>= 12;

Before instruction		After instruction	
Var1	0x1256ABAB	Var1	0x0001256A

Example3 PROD >>= 4;

Before instruction		After instruction	
Product register	0x12560000ABCD	Product register	0x0012560000ABC

6.2.5.1.45. <<

Syntax

`VAR16 <<= N` shift `VAR16` left by `N`
`VAR32 <<= N` shift `VAR32` left by `N`
`PROD <<= N` shift `PROD` (product reg.) right by `N`

Operands `VAR16`: integer variable
 `VAR32`: long variable
 `PROD`: product register
 `N`: shift factor

Type

MPL Program	On-line
X	—

Binary code

VAR16 <<= N

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	1	0	0	0	0	0	1	0	<i>N (0<N<15)</i>			
&VAR16															

VAR32 <<= N

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	1	0	0	1	0	0	1	0	<i>N (0<N<15)</i>			
&VAR32															

PROD <<= N

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	1	0	0	0	1	0	1	0	<i>N (0<N<15)</i>			
&PROD															

Description The operand is left shifted with the specified number of bits (`N`). High order bits are lost and the low order bits are zeroed. If the operand is **PROD**, the entire 48-bit product register is left shifted.

Execution Variable = Value of variable shifted to left with `N` bits

Example1

```

int Var1;
...
Var1 <<= 4;
  
```

Before instruction		After instruction	
Var1	0x1256	Var1	0x2560

Example2

```

long Var1;
...
Var1 <<= 12;

```

Before instruction		After instruction	
Var1	0x1256ABAB	Var1	0x6AABAB000

Example3

```

PROD <<= 4;

```

Before instruction		After instruction	
PROD	0x12560000ABCD	PROD	0x2560000ABCD0

6.2.5.1.46. ABORT

Syntax

ABORT Abort cancelable MPL function

Operands –

Type	MPL Program	On-line
	X	X

Binary code

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

Description **ABORT** command cancels the execution of a MPL function called using **CALLS** instruction. After the execution of **ABORT**, the MPL program continues with the next instruction after the cancelable call of the function.

Example

```
....
CALLS First_function; //Cancelable call of First_function
STOP; //Stop the motion
....
END; //End of MPL program

First_Function: //definition of First_Function
....
CALL Second_Function; //Call function Second_Function
MODE PP;
....
RET; //Return from First_Function
Second_Function: //definition of First_Function
....
GOTO user_label, user_var,EQ;//Branch to user_labelif
//user_var ==0
ABORT; //Cancel the execution of First_Function
//Next MPL instruction executed is STOP;
user_label:
....
RET; //Return from Second_function
```

6.2.5.1.47. ADDGRID

Syntax

ADDGRID (*value_1, value_2,...*) Add the specified groups to **GROUP ID**

Operands *value_1, value_2*: specify a group number between 1 and 8

Type	MPL Program	On-line
	X	X

Binary code

ADDGRID value16															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0
<i>Value16</i>															

Description The command adds more groups to the **group ID**. On each axis, the group ID represents a filter for a multicast transmission. When a multicast message is received, each axis compares the message group ID with its own group ID. If the two group IDs have at least one group in common, the message is accepted. For example, if an axis is member of group 1 and group 3, it will receive all the messages sent with a group ID including group 1 or group 3. The group ID is an 8-bit integer value. Each bit corresponds to one group: bit 0 – group 1, bit 1 – group 2... bit 7 – group 8.

After the execution of this command, the group ID value is modified as follows:

- Bit 0 is set to 1, if (group) 1 occurs in the parenthesis
- Bit 1 is set to 1, if (group) 2 occurs in the parenthesis
- ...
- Bit 7 is set to 1, if (group) 8 occurs in the parenthesis.

Example

```
//local axis has group ID = 1 -> belongs to group 1
ADDGRID (2, 4); //local axis belongs also to groups 2 and 4
                //new group ID = 11 (00001011b)
.....
[G4] {STOP;} //send stop motion to all axes from group 4
                //local axis will stop too as member of group 4
```

6.2.5.1.48. AXISID

Syntax

AXISID *value16* Set **AXIS ID** address

AXISID *VAR16* Set **AXIS ID** with value of VAR16

Operands *value16*: immediate value between 1 and 255
 VAR16: integer variable

Type

MPL Program	On-line
X	X

Binary code

AXISID *value16*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1
<i>Value16</i>															

AXISID *VAR16*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	1
<i>&VAR16</i>															

Description The command changes the **axis ID**. In multiple-axis configurations, each axis is identified through a unique number between 1 and 255 – the axis ID. If the destination of a message is an axis ID, the message is received only by the axis with the same axis ID.

After the execution of this command, the axis ID is set with the immediate value specified or the value of the 16-bit variable.

Example

```
AXISID 10;                      //from now on, the local axis ID is 10
....
[10] {AXISID 9;}                //change the ID of axis 10 to 9 (this
                                 //instruction is send and executed on
                                 //the actual axis 10)
....
[9] {CSPD = 30;}                //Send CPOS = 30 to axis 9 (previous axis 10)
```

6.2.5.1.49. AXISOFF

Syntax

AXISOFF **AXIS** is **OFF** (deactivate control)

Operands –

Type	MPL Program	On-line
	X	X

Binary code

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

Description The command deactivates the drive control loops, the reference generator and the PWM output commands for the power stage (all the switching devices are off). All the measurements remain active and therefore the motor currents, speed, position as well as the supply voltage continue to be updated and monitored. The **AXISOFF** command is automatically generated when a protection is triggered or when the drive Enable input goes from status enabled to status disabled.

Example

```
// Execute repetitive moves. After each one, set AXISOFF.
// Motor may move freely. Restart after 20s. Position
// feedback: 500 lines incremental encoder (2000 counts/rev)
CACC = 0.3183;//acceleration rate = 1000[rad/s^2]
CSPD = 33.3333;//slew speed = 1000[rpm]
CPOS = 6000;//position command = 3[rot]
CPA; //position command is absolute
Loop: MODE PP; // position profile
CPOS += 6000; set new position command
UPD; //execute immediate
AXISON;      //Activate the control loops and PWM outputs
!MC; // set event on motion complete
WAIT!;//Wait until the event occurs i.e. the motor stops
AXISOFF; //Deactivate the control loops and PWM outputs
!RT 20000; //set a 20s delay (1s = 1000 slow loop samplings)
WAIT!;//Wait until the event occurs (to pass the 20s)
GOTO Loop; //Restart the motion
```

```
AXISON;      //Activate the control loops and PWM outputs
!MC; // set event on motion complete
WAIT!;//Wait until the event occurs i.e. the motor stops
AXISOFF; //Deactivate the control loops and PWM outputs
!RT 20000; //set a 20s delay (1s = 1000 slow loop samplings)
WAIT!;//Wait until the event occurs (to pass the 20s)
GOTO Loop; //Restart the motion
```

6.2.5.1.51. BEGIN

Syntax

BEGIN Beginning of a MPL program

Operands –

Type	MPL Program	On-line
	X	—

Binary code

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

Description This command must be the first in a MPL program. In the AUTORUN mode, the drive/motor reads the first EEPROM memory location at address 0x4000 and checks if the binary code is 0x649C corresponding to the MPL instruction **BEGIN**. If this condition is true, the MPL program from the EEPROM memory is executed starting with the next instruction after **BEGIN**. If the condition is false, the drive/motor enters in the slave mode and waits to receive commands from a host via a communication channel.

Example

```
BEGIN;           // Starting point of a MPL program
...
ENDINIT;        //End of initialization
...
END;            //end of main section of a MPL program
```

6.2.5.1.52. CALL

Syntax

CALL <i>Label</i>	Unconditional CALL
CALL <i>value16</i>	Unconditional CALL
CALL <i>VAR16</i>	Unconditional CALL
CALL <i>Label, VAR, Flag</i>	CALL if <i>VAR Flag</i> 0
CALL <i>value16, VAR, Flag</i>	CALL if <i>VAR Flag</i> 0
CALL <i>VAR16, VAR, Flag</i>	CALL if <i>VAR Flag</i> 0

Operands

Label: a label providing the 16-bit value of a MPL function address

Value16: immediate 16-bit of a MPL function address

VAR16: integer variable containing the MPL function address

VAR: 16 or 32-bit MPL test variable compared with 0

Flag: one of the conditions: EQ, NEQ, LT, LEQ, GT, GEQ

Type

MPL Program	On-line
X	X

Binary code

CALL *Label*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	1	0	0	0	0	0	0	0	0	0	1
<i>&Label</i>															

CALL *value16*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	1	0	0	0	0	0	0	0	0	0	1
<i>value16</i>															

CALL *VAR16*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	1	1	0	0	0	0	0	0	0	0	1
<i>&VAR16</i>															

CALL *Label, VAR, Flag*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	1	0	VT	<i>Flag</i>							1
<i>&VAR</i>															
<i>&Label</i>															

CALL *value16, VAR, Flag*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	1	0	VT	<i>Flag</i>							1
<i>&VAR</i>															
<i>value16</i>															

CALL *VAR16, VAR, Flag*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	1	1	VT	<i>Flag</i>							1
<i>&VAR32</i>															
<i>&VAR16</i>															

Description Calls a MPL function (subroutine). A MPL function is a set of MPL commands which starts with a label and ends with the **RET** instruction. The label gives the *MPL function address* and name. *MPL function address* may also be specified by an immediate value or by the value of a 16-bit MPL variable. The call can be unconditional or unconditional. In a conditional call, a condition is tested. If the condition is true the MPL function is executed, else the next MPL command is carried out. The condition is specified by a 16-bit or 32-bit *test variable* ($VT=0$ for 16-bit variable and $VT = 1$ for 32-bit variable) and a *test condition* added after the label with the *MPL function address*. The *test variable* is always compared with zero. The possible *test conditions* are:

EQ	if VAR = 0
NEQ	if VAR ≠ 0
LT	if VAR < 0
LEQ	if VAR ≤ 0
GT	if VAR > 0
GEQ	if VAR ≥ 0

Example

```
CALL Function1, var1, GEQ;    //call Function1 if i_var1 >= 0
CALL Function1, var1, EQ;    //call Function1, if i_var1 = 0
CALL Function1, var1, NEQ;   //call Function1, if i_var1 != 0
CALL Function1;              //call Function1 unconditionally
...
END;                          // end of MPL program main section
Function1:
...
RET;
```

6.2.5.1.53. CALLS

CALLS Cancelable call of a MPL function

Syntax

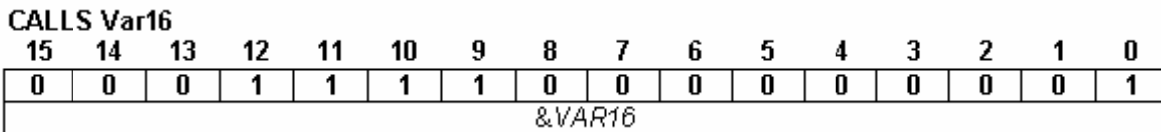
CALLS *Label* Cancelable **CALL** of a MPL function

CALLS *VAR16* Cancelable **CALL** with address set in VAR16

Operands *Label*: 16-bit program memory address
VAR16: integer variable

Type	MPL Program	On-line
	X	X

Binary code



Description Calls a MPL function (subroutine) with possibility to interrupt the function execution using **ABORT** command. This is a *cancelable call*. A MPL function is a set of MPL commands which starts with a label and ends with the **RET** instruction. The label gives the *MPL function address* and name. *MPL function address* may also be specified by an immediate value or by the value of a 16-bit MPL variable.

Only one function may be called with a cancelable call at a time. A cancelable call issued while another one is still active (the called function is in execution) is ignored and a command error is set in error register **MER.14**. Also status register low **SRL.7** is set. While a cancelable call is active, **SRL.8** = 1.

Example

```

...
CCALL fct1;          //cancelable call of Function1
STOP;
...
END;
Function1:          // Function1 definition
...
ABORT;              // if this command is encountered or

```

```
... // got via a communication channel
RET; // next instruction executed is STOP
```

6.2.5.1.54. CANBR

Syntax

CANBR *value16* Set **CAN**-bus **Baud Rate** to *value16*

CANBR *VAR16* Set **CAN**-bus **Baud Rate** to *VAR16*

Operands *value16*: 16-bit unsigned integer immediate value
 VAR16: 16-bit integer variable

Type	MPL Program	On-line
	X	X

Binary code

CANBR *value16*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0
Value16															

CANBR *VAR16*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0
&VAR16															

Description Sets the baud rate and bit sampling timing for the CAN-bus communication channel. The new baud settings can be provided either as an immediate value or by the value of a MPL variable. In both cases, the possible values are:

Baud rate [kb]	Value 16
125	0xF36C
250	0x736C
500	0x3273
800	0x412A
1000	0x1273

The current CAN-bus settings are saved in the MPL register **CBR**, and may be read at any moment. The CAN-bus baud rate is set at power on using the following algorithm:

- a. With the value read from the EEPROM setup table
- b. If the setup table is invalid, with the last baud rate read from a valid setup table
- c. If there is no baud rate set by a valid setup table, with 500kb

Remarks:

- Use this command when a drive/motor operates in AUTORUN (after power on starts to execute the MPL program from the EEPROM) and it must communicate with a host at a CAN baud rate different from the default value. In this case, the MPL program must start with a CAN baud rate change.

-
- *An alternate solution to the above case is to set via **CANBR** command the desired baud rate and then to save it in the EEPROM, with the command **SAVE**. After a reset, the drive/motor starts directly with the new baud rate, if the setup table was valid. Once set, the new default baud rate is preserved, even if the setup table is later on disabled, because the default CAN baud rate is stored in a separate area of the EEPROM.*

Example

```
CANBR 0x1273; // set CAN-bus for 1Mb
```

6.2.5.1.55. CHECKSUM

Syntax

CHECKSUM, *TM Val_S, Val_E, VAR_D* Checksum between addresses *Val_S* and *Val_E*

CHECKSUM, *TM VAR_S, VAR_E, VAR_D* Checksum between addresses set in variables *VAR_S* and *VAR_E*

Operands *Val_S*: 16-bit unsigned integer value representing the checksum start address
 Val_E: 16-bit unsigned integer value representing the checksum end address
 VAR_S: 16-bit variable containing the checksum start address
 VAR_E: 16-bit variable containing the checksum end address
 VAR_D: 16-bit variable containing the checksum result
 TM: Memory type (see TypeMem table below)

Type

MPL Program	On-line
X	X

Binary code

CHECKSUM, *TypeMem V16_Start, V16_Stop, VAR16D*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	1	1	0	<i>TypeMem</i>	1	0	0	0	0	0
& VAR16D															
V16_Start address															
V16_Stop address															

CHECKSUM, *TypeMem VAR16_Start, VAR16_Stop, VAR16D*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	1	1	1	<i>TypeMem</i>	1	0	0	0	0	0
& VAR16D															
&VAR16_Start															
&VAR16_Stop															

Description Computes the sum module 65536 of all the memory locations between a *start address* and an *end address*. The *start address* and the *end address* may be specified as 16-bit unsigned immediate values or via 2 16-bit MPL variables. The checksum result is saved in a 16-bit destination variable. The memory location can be of 3 types: RAM for data (dm), RAM for MPL programs (pm), EEPROM SPI-connected for MPL programs (spi).

TypeMem	
DM	01
PM	00
SPI	10

Example

```
// compute checksum between EEPROM addresses 0x5000 and 0x5007
int user_var;
....
```

```
CHECKSUM, spi 0x5000, 0x5007, user_var; // user_var = checksum value
```

Before instruction				After instruction			
user_var		x		user_var		0xD467	
EEPROM	start	address	0xB004	EEPROM	start	address	0xB004
0x5000				0x5000			
EEPROM	address	0x5001	0x0FF1	EEPROM	address	0x5001	0x0FF1
EEPROM	address	0x5002	0x0366	EEPROM	address	0x5002	0x0366
EEPROM	address	0x5003	0x0404	EEPROM	address	0x5003	0x0404
EEPROM	address	0x5004	0x0C09	EEPROM	address	0x5004	0x0C09
EEPROM	address	0x5005	0x0010	EEPROM	address	0x5005	0x0010
EEPROM	address	0x5006	0x00E7	EEPROM	address	0x5006	0x00E7
EEPROM	address	0x5007	0x0008	EEPROM	address	0x5007	0x0008

6.2.5.1.56. CIRCLE

Only available on multi-axis Motion Controller

Syntax

CIRCLE1 *Radius, Theta_inc* Vector **CIRCLE** segment
CIRCLE2 *Radius, Theta_start*

Operands *Radius* – circle radius
 Theta_start – start angle for circular segment
 Theta_inc – angle increment for circular segment

Type	MPL Program	On-line
	X	X

Binary code

CIRCLE1 <i>Radius, Theta_inc</i>															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	1	0	1	0	0	0	0	0	0	0	0
LOWORD(<i>Radius</i>)															
HIWORD(<i>Radius</i>)															
LOWORD(<i>Angle_inc</i>)															
HIWORD(<i>Angle_inc</i>)															

CIRCLE2 <i>Radius, Theta_start</i>															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	1	0	0	1	0	0	0	0	0	0	0
LOWORD(<i>M</i>)															
HIWORD(<i>M</i>)															
LOWORD(<i>Angle_start</i>)															
HIWORD(<i>Angle_start</i>)															

Description CIRCLE1 and CIRCLE2 define a circular segment for 2D trajectory executed in Vector Mode. Positive values for *Theta_inc* mean CCW movement while negative values mean CW movement.

Based on *Radius*, *Theta_inc* and *Theta_start* the MPL compiler from MotionPRO Developer computes the actual parameters used by the motion controller to generate the PVT points for slave axes.

If the points are sent from a host then the following relations must be used to compute the actual parameters of the circular segment:

$$\text{Angle_inc} = \frac{\text{Theta_inc}}{360^\circ} \times 2^{30}$$

$$\text{Angle_start} = \frac{\text{Theta_start}}{360^\circ} \times 2^{30}$$

$$M = \frac{1}{2\pi} \times \frac{1}{R} \times 2^{31}$$

Example

```
// Vector mode - circle with radius 3.14mm. Position feedbacks: 500
// lines incremental encoder

SETMODE 0xCF00; //Clear buffer
VPLANE (A, B, C); //Define coordinate system and tangent axis
RESRATIOX=0u;
RESRATIOY=0u;
NLINESTAN=2000;
MODE VM; // Set Vector Mode
// Circular segment of radius 3.14159[mm], with initial angle 0[deg] and
angle increment 360[deg])
CIRCLE1 1L, 360.; CIRCLE2 1L, 0.;
UPD; //Execute immediate
// Insert End Segment
VSEG1 0L, 0L; VSEG2 0L, 0L;
```

6.2.5.1.58. CPR

Syntax

CPR Command Position is Relative

Operands –

Type	MPL Program	On-line
	X	X

Binary code

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

Description After the execution of this instruction, all subsequent position commands will be considered as relative in the following motion modes: trapezoidal profiles, S-curve profiles, PVT and PT. This setting remains until the execution of a **CPA** command. In the trapezoidal profile mode, the position to reach can be computed in 2 ways: standard (default) or additive. In standard relative mode, the position to reach is computed by adding the position increment to the instantaneous position in the moment when the command is executed. In the additive relative mode, the position to reach is computed by adding the position increment to the previous position to reach, independently of the moment when the command was issued.

The additive relative mode is activated by setting **ACR.11** = 1 and is automatically disabled after an update command **UPD**, which sets **ACR.11** = 0 restoring the standard relative mode.

Example

```
//Position profile
//Position feedback: 500 lines incremental
// encoder (2000 counts/rev)
CACC = 0.3183;//acceleration rate = 1000[rad/s^2]
CSPD = 16.6667;//slew speed = 500[rpm]
CPOS = 7000;//position command = 3.5[rot]
CPR; //position command is relative
MODE PP;
TUM1; //set Target Update Mode 1
UPD; //execute immediate
!MC; WAIT!; //wait for completion
```

6.2.5.1.59. DINT

Syntax

6.2.5.1.60. DIS2CAPI

Syntax**DIS2CAPI** **DIS**able 2nd **CAP**ture Index**Operands** –

Type	MPL Program	On-line
	X	X

Binary code

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

Description After the execution of this instruction the 2nd capture/encoder index input capability to detect transitions is disabled.

Use the **EN2CAPI0** or **EN2CAPI1** instructions to enable this input capability to detect high-low or low-high transitions.

Example

```
DIS2CAPI; //disable 2nd capture/encoder index input
```

6.2.5.1.61. DISCAPI

Syntax**DISCAPI** **DIS**able **CAP**ture Index**Operands** –

Type	MPL Program	On-line
	X	X

Binary code

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

Description After the execution of this instruction the 1st capture/encoder index input capability to detect transitions is disabled.

Use the **ENCAPI0** or **ENCAPI1** instructions to enable this input capability to detect high-low or low-high transitions.

Example

```
DISCAPI; //disable 1st capture/encoder index input
```

6.2.5.1.64. EINT

Syntax

EINT Enable MPL **INT**errupts

Operands –

Type	MPL Program	On-line
	X	X

Binary code

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

Description After the execution of this instruction, the MPL interrupts are globally enabled. If an interrupt flag is set and the interrupt is enabled in the interrupt control register **ICR**, the interrupt request is accepted and the associated interrupt service routine is called. The MPL interrupts can be globally disabled using the **DINT** instruction.

Example

```
EINT; //globally enable all MPL interrupts
```

6.2.5.1.65. EN2CAPI0

Syntax

EN2CAPI0 Enable 2nd**CAP**ture Index 1->0

Operands –

Type	MPL Program	On-line
	X	X

Binary code

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

Description Enables 2nd capture/encoder index input capability to detect a transition from 1(high) to 0(low). When the first transition occurs:

- The input capability to detect transitions is disabled. It must be enabled again to detect another transition
- Motor position **APOS_MT** is captured and memorized in the MPL variable **CAPPOS**, except the case of open-loop systems, where reference position **TPOS** is captured instead
- Master position **APOS2** or load position **APOS_LD** is captured and memorized in the MPL variable **CAPPOS2**, except the case of steppers controlled open loop with an encoder on the load, when load position is captured in **CAPPOS**.

The selection between master and load position is done as follows: load position is saved in **CAPPOS2** only for the setup configurations which use different sensors for load and motor and foresee a transmission ratio between them. For all the other setup configurations, the master position is saved in **CAPPOS2**.

Example

```
//Stop motion on next 2nd encoder index
EN2CAPI0; //Set event: When the 2nd encoder index goes high->low
!CAP;
STOP!;//Stop the motion when event occurs
WAIT!;//Wait until the event occurs
CPOS = CAPPOS; // new command position = captured position
CPA; //position command is absolute
MODE PP;
TUM1; //set Target Update Mode 1
UPD; //execute immediate
!MC; WAIT!; //wait for completion
```

6.2.5.1.66. EN2CAPI1

Syntax

EN2CAPI1 Enable 2ndCAPture Index 0->1

Operands –

Type	MPL Program	On-line
	X	X

Binary code

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

Description Enables 2nd capture/encoder index input capability to detect a transition from 0(low) to 1(high). When the first transition occurs:

- The input capability to detect transitions is disabled. It must be enabled again to detect another transition
- Motor position **APOS_MT** is captured and memorized in the MPL variable **CAPPOS**, except the case of open-loop systems, where reference position **TPOS** is captured instead

- Master position **APOS2** or load position **APOS_LD** is captured and memorized in the MPL variable **CAPPOS2**, except the case of steppers controlled open loop with an encoder on the load, when load position is captured in **CAPPOS**.

The selection between master and load position is done as follows: load position is saved in **CAPPOS2** only for the setup configurations which use different sensors for load and motor and foresee a transmission ratio between them. For all the other setup configurations, the master position is saved in **CAPPOS2**.

Example

```
//Stop motion on next 2nd encoder index
EN2CAPI1; //Set event: When the 2nd encoder index goes low->high
!CAP;
STOP!;//Stop the motion when event occurs
WAIT!;//Wait until the event occurs
CPOS = CAPPOS; // new command position = captured position
CPA; //position command is absolute
MODE PP;
TUM1; //set Target Update Mode 1
UPD; //execute immediate
!MC; WAIT!; //wait for completion
```

6.2.5.1.67. ENCAPI0

Syntax

ENCAPI0 Enable **CAP**tured Index 1->0

Operands -

Type	MPL Program	On-line
	X	X

Binary code

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

Description Enables 1st capture/encoder index input capability to detect a transition from 1(high) to 0(low). When the first transition occurs:

- The input capability to detect transitions is disabled. It must be enabled again to detect another transition

-
- Motor position **APOS_MT** is captured and memorized in the MPL variable **CAPPOS**, except the case of open-loop systems, where reference position **TPOS** is captured instead
 - Master position **APOS2** or load position **APOS_LD** is captured and memorized in the MPL variable **CAPPOS2**, except the case of steppers controlled open loop with an encoder on the load, when load position is captured in **CAPPOS**.

The selection between master and load position is done as follows: load position is saved in **CAPPOS2** only for the setup configurations which use different sensors for load and motor and foresee a transmission ratio between them. For all the other setup configurations, the master position is saved in **CAPPOS2**.

Example

```
//Stop motion on next 1st encoder index
ENCAP10; //Set event: When the 1st encoder index goes high->low
!CAP;
STOP!;//Stop the motion when event occurs
WAIT!;//Wait until the event occurs
CPOS = CAPPOS; // new command position = captured position
CPA; //position command is absolute
MODE PP;
TUM1; //set Target Update Mode 1
UPD; //execute immediate
!MC; WAIT!; //wait for completion
```

6.2.5.1.68. ENCAPI1

Syntax

ENCAPI1 Enable CAPture Index 0->1

Operands -

Type	MPL Program	On-line
	X	X

Binary code

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

Description Enables 1st capture/encoder index input capability to detect a transition from 0(low) to 1(high). When the first transition occurs:

- The input capability to detect transitions is disabled. It must be enabled again to detect another transition
- Motor position **APOS_MT** is captured and memorized in the MPL variable **CAPPOS**, except the case of open-loop systems, where reference position **TPOS** is captured instead
- Master position **APOS2** or load position **APOS_LD** is captured and memorized in the MPL variable **CAPPOS2**, except the case of steppers controlled open loop with an encoder on the load, when load position is captured in **CAPPOS**.

The selection between master and load position is done as follows: load position is saved in **CAPPOS2** only for the setup configurations which use different sensors for load and motor and foresee a transmission ratio between them. For all the other setup configurations, the master position is saved in **CAPPOS2**.

Example

```
//Stop motion on next 1st encoder index
ENCAPI1; //Set event: When the 1st encoder index goes low->high
!CAP;
STOP!;//Stop the motion when event occurs
WAIT!;//Wait until the event occurs
CPOS = CAPPOS; // new command position = captured position
CPA; //position command is absolute
MODE PP;
TUM1; //set Target Update Mode 1
UPD; //execute immediate
!MC; WAIT!; //wait for completion
```


Description The **ENDINIT** instruction marks the END of the INITIALIZATION part of the MPL program. This command uses the available setup data to perform key initializations, but does not activate the controllers or the PWM outputs. These are activated with the **AXISON** command. After power on, the **ENDINIT** command may be executed only once. Subsequent **ENDINIT** commands are ignored. The first **AXISON** command must be executed only after the **ENDINIT** command.

Example

```
BEGIN;          // Starting point of a MPL program
...
ENDINIT;       //End of initialization
...
END;           //end of main section of a MPL program
```

6.2.5.1.71. ENEEPROM

Syntax

ENEEPROM **EN**nable communication with the **EEPROM**

Operands -

Type

MPL Program	On-line
X	X

Binary code

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

Description Enables the SPI-based communication with the drive/motor EEPROM after this was disabled by the initialization of feedback devices like SSI or EnDat encoders, which use the same SPI link as the EEPROM. This initialization is done during the **ENDINIT** command.

The **ENEEPROM** command is intended for the hosts working with ElectroCraft drives set in configurations with SSI or EnDat encoders as position feedback. Following this command, the internal SPI-link with the SSI or EnDat encoders is disabled and the SPI-link with the EEPROM is enabled. This offers access to the drive EEPROM after execution of the **ENDINIT** command, without resetting the drives.

***Remark:** The **ENEEPROM** command must be executed only **AFTER** issuing the commands **AXISOFF** and **END** which stop the motor control and MPL program execution*

Example

```
ENEEPROM; // enable EEPROM
```

6.2.5.1.72. ENIO

Syntax

ENIO

Operands –

Type	MPL Program	On-line
	X	X

Binary code

Description

Example

ENIO;

6.2.5.1.73. ENLSN0

Syntax

ENLSN0 Enable Limit Switch Negative 1->0

Operands –

Type	MPL Program	On-line
	X	X

Binary code

Firmware version [FAxx](#)

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

Firmware version [FBxx](#)

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

Description Enables negative limit switch input capability to detect a transition from 1(high) to 0(low). When the first transition occurs:

- The input capability to detect transitions is disabled. It must be enabled again to detect another transition
- Motor position **APOS_MT** is captured and memorized in the MPL variable **CAPPOS**, except the case of open-loop systems, where reference position **TPOS** is captured instead
- Master position **APOS2** or load position **APOS_LD** is captured and memorized in the MPL variable **CAPPOS2**, except the case of steppers controlled open loop with an encoder on the load, when load position is captured in **CAPPOS**.

The selection between master and load position is done as follows: load position is saved in **CAPPOS2** only for the setup configurations which use different sensors for load and motor and foresee a transmission ratio between them. For all the other setup configurations, the master position is saved in **CAPPOS2**.

Example

```
//Reverse when the active low negative limit switch is reached
//Position feedback: 500 lines encoder (2000 counts/rev)
CACC = 0.0637; //acceleration rate = 200[rad/s^2]
CSPD = -16.6667; //jog speed = -500[rpm]
MODE SP;
UPD; //execute immediate
ENLSN0;//Enable negative limit switch for high->low transitions
!LSN; //Set event on negative limit switch(high->low transition)
WAIT!;//Wait until the event occurs
!MC; // limit switch is active -> quick stop mode active
WAIT!;// wait until the motor stops because only then the new
// motion commands are accepted
CSPD = 40; //jog speed = 1200[rpm]
MODE SP; //after quick stop set again the motion mode
UPD; //execute immediate
```

6.2.5.1.74. ENLSN1

Syntax

ENLSN1 Enable Limit Switch Negative 0->1

Operands –

Type	MPL Program	On-line
	X	X

Binary code

Firmware version [FAxx](#)

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

Firmware version [FBxx](#)

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

Description Enables negative limit switch input capability to detect a transition from 0(low) to 1(high).
When the first transition occurs:

- The input capability to detect transitions is disabled. It must be enabled again to detect another transition
- Motor position **APOS_MT** is captured and memorized in the MPL variable **CAPPOS**, except the case of open-loop systems, where reference position **TPOS** is captured instead
- Master position **APOS2** or load position **APOS_LD** is captured and memorized in the MPL variable **CAPPOS2**, except the case of steppers controlled open loop with an encoder on the load, when load position is captured in **CAPPOS**.

The selection between master and load position is done as follows: load position is saved in **CAPPOS2** only for the setup configurations which use different sensors for load and motor and foresee a transmission ratio between them. For all the other setup configurations, the master position is saved in **CAPPOS2**.

Example

```
//Reverse when the active high negative limit switch is reached
//Position feedback: 500 lines encoder (2000 counts/rev)
CACC = 0.0637; //acceleration rate = 200[rad/s^2]
CSPD = -16.6667; //jog speed = -500[rpm]
MODE SP;
UPD; //execute immediate
ENLSN1;//Enable negative limit switch for low->high transitions
!LSN; //Set event on negative limit switch(low->high transition)
```

```
WAIT!;//Wait until the event occurs
!MC; // limit switch is active -> quick stop mode active
WAIT!;// wait until the motor stops because only then the new
      // motion commands are accepted
CSPD = 40;      //jog speed = 1200[rpm]
MODE SP;       //after quick stop set again the motion mode
UPD;           //execute immediate
```

6.2.5.1.75. ENLSP0

Syntax

ENLSP0 Enable Limit Switch Positive 1->0

Operands –

Type	MPL Program	On-line
	X	X

Binary code

Firmware version [FAxx](#)

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

Firmware version [FBxx](#)

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

Description Enables positive limit switch input capability to detect a transition from 1(high) to 0(low). When the first transition occurs:

- The input capability to detect transitions is disabled. It must be enabled again to detect another transition
- Motor position **APOS_MT** is captured and memorized in the MPL variable **CAPPOS**, except the case of open-loop systems, where reference position **TPOS** is captured instead
- Master position **APOS2** or load position **APOS_LD** is captured and memorized in the MPL variable **CAPPOS2**, except the case of steppers controlled open loop with an encoder on the load, when load position is captured in **CAPPOS**.

The selection between master and load position is done as follows: load position is saved in **CAPPOS2** only for the setup configurations which use different sensors for load and motor and foresee a transmission ratio between them. For all the other setup configurations, the master position is saved in **CAPPOS2**.

Example

```
//Reverse when the active low positive limit switch is reached
//Position feedback: 500 lines encoder (2000 counts/rev)
CACC = 0.0637; //acceleration rate = 200[rad/s^2]
CSPD = -16.6667; //jog speed = -500[rpm]
```

```
MODE SP;
UPD;           //execute immediate
ENLSP0; //Enable positive limit switch for high->low transitions
!LSP; //Set event on positive limit switch(high->low transition)
WAIT!; //Wait until the event occurs
!MC; // limit switch is active -> quick stop mode active
WAIT!; // wait until the motor stops because only then the new
        // motion commands are accepted
CSPD = 40;     //jog speed = 1200[rpm]
MODE SP;      //after quick stop set again the motion mode
UPD;          //execute immediate
```


6.2.5.1.76. ENLSP1

Syntax

ENLSP1 Enable Limit Switch Positive 0->1

Operands –

Type	MPL Program	On-line
	X	X

Binary code

Firmware version [FAxx](#)

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

Firmware version [FBxx](#)

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

Description Enables positive limit switch input capability to detect a transition from 0(low) to 1(high). When the first transition occurs:

- The input capability to detect transitions is disabled. It must be enabled again to detect another transition
- Motor position **APOS_MT** is captured and memorized in the MPL variable **CAPPOS**, except the case of open-loop systems, where reference position **TPOS** is captured instead
- Master position **APOS2** or load position **APOS_LD** is captured and memorized in the MPL variable **CAPPOS2**, except the case of steppers controlled open loop with an encoder on the load, when load position is captured in **CAPPOS**.

The selection between master and load position is done as follows: load position is saved in **CAPPOS2** only for the setup configurations which use different sensors for load and motor and foresee a transmission ratio between them. For all the other setup configurations, the master position is saved in **CAPPOS2**.

Example

```
//Reverse when the active high positive limit switch is reached
//Position feedback: 500 lines encoder (2000 counts/rev)
CACC = 0.0637; //acceleration rate = 200[rad/s^2]
CSPD = -16.6667; //jog speed = -500[rpm]
```

```
MODE SP;
UPD;           //execute immediate
ENLSP1;//Enable positive limit switch for low->high transitions
!LSP; //Set event on positive limit switch(low->high transition)
WAIT!;//Wait until the event occurs
!MC; // limit switch is active -> quick stop mode active
WAIT!;// wait until the motor stops because only then the new
        // motion commands are accepted
CSPD = 40;     //jog speed = 1200[rpm]
MODE SP;      //after quick stop set again the motion mode
UPD;          //execute immediate
```

6.2.5.1.77. EXTREF

Syntax

EXTREF *value* Set **EXT**ernal **REF**erence type

Operands *value*: type of reference 0, 1 or 2

Type	MPL Program	On-line
	X	X

Binary code

EXTREF 0

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

EXTREF 1

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

EXTREF 2

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

Description Sets the external reference type depending on the parameter *value*:

- *value* = 0: online – the reference is sent via a communication channel in one of the variables **EREF**P, **EREF**S, **EREF**T, **EREF**V function of the control mode
- *value* = 1: analogue – the reference is read from a dedicated analogue input
- *value* = 2: digital – the reference is provided as pulse & direction or encoder like signals

Example

```
EXTREF 1; // the external reference is read from the analogue
           // input dedicated for this purpose
```

6.2.5.1.78. FAULTR

Syntax

FAULTR **FAULT** Reset

Operands –

Type	MPL Program	On-line
	X	X

Binary code

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

Description Gets out the drive/motor from the [FAULT status](#) in which it enters when an error occurs. After a **FAULTR** command, most of the error bits from [MER](#) are cleared (set to 0), the Ready output (if present) is set to **ready** level, the Error output (if present) is set to **no error** level.

Remarks:

- The **FAULT** reset command does not change the status of *MER.15* (enable input on disabled level), *MER.7* (negative limit switch input active), *MER.6* (positive limit switch input active) and *MER.2* (invalid setup table)
- The drive/motor will return to **FAULT** status if there are errors when the **FAULTR** command is executed

Example

```
FAULTR;            // reset fault status
```

6.2.5.1.79. GOTO

Syntax

GOTO *Label* Unconditional **GOTO** to *Label*
GOTO *Value16* Unconditional **GOTO** to *Value16*
GOTO *VAR16* Unconditional **GOTO** to address stored in *VAR16Addr*
GOTO *Label, VAR, Flag* **GOTO** *Label* if *VAR Flag* 0
GOTO *Value16, VAR, Flag* **GOTO** *Value16* if *VAR Flag* 0
GOTO *VAR16, VAR, Flag* **GOTO** address set in *Var16Addr* if *VAR16 Flag* 0

Operands *Label*: a label providing the 16-bit value of a jump address
 Value16: immediate 16-bit jump address
 VAR16: integer variable containing the jump address
 VAR: 16 or 32-bit MPL test variable compared with 0

Flag: one of the conditions: EQ, NEQ, LT, LEQ, GT, GEQ

Type	MPL Program	On-line
	X	X

Binary code

GOTO *Label*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0
<i>&Label</i>															

GOTO *value16*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0
<i>value16</i>															

GOTO *VAR16*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	1	1	0	0	0	0	0	0	0	0	0
<i>&VAR16</i>															

GOTO *Label, VAR, Flag*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	1	0	VT	<i>Flag</i>							0
<i>&VAR</i>															
<i>&Label</i>															

GOTO *value16, VAR, Flag*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	1	0	VT	<i>Flag</i>							0
<i>&VAR</i>															
<i>value16</i>															

GOTO *VAR16, VAR, Flag*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	1	1	VT	<i>Flag</i>							0
<i>&VAR32</i>															
<i>&VAR16</i>															

Description Executes a jump to the MPL program position specified via the *jump address*. The *jump address* is provided via a label, an immediate value or by the value of a 16-bit MPL variable. The jump can be unconditional or unconditional. In a conditional jump, a condition is tested. If the condition is true the jump is executed, else the next MPL command is carried out. The condition is specified by a 16-bit or 32-bit *test variable* and a *test condition* added after the label with the *jump address*. The *test variable* is always compared with zero. The possible *test conditions* are:

- EQ if VAR = 0
- NEQ if VAR ≠ 0
- LT if VAR < 0
- LEQ if VAR ≤ 0
- GT if VAR > 0
- GEQ if VAR ≥ 0

Example

```
GOTO label1, var1, LT; // jump to label1 if var1 < 0
GOTO label2, var1, LEQ; // jump to label2 if var1 <= 0
GOTO label3, var1, GT; // jump to label3 if var1 > 0
GOTO label4; // unconditional jump to label4
GOTO var_address; // unconditional jump to jumps address
// provided by var_address value
```

6.2.5.1.80. GROUPID

Syntax

GROUPID (value_1, value_2,...) Set **GROUP ID** address

Operands *value_1, value_2*: specify a group number between 1 and 8

Type

MPL Program	On-line
X	X

Binary code

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0
<i>Value</i>															

Description The command sets the **group ID**. On each axis, the group ID represents a filter for a multicast transmission. When a multicast message is received, each axis compares the message group ID with its own group ID. If the two group IDs have at least one group in common, the message is accepted. For example, if an axis is member of group 1 and group 3, it will receive all the messages sent with a group ID including group 1 or group 3. The group ID is an 8-bit integer value. Each bit corresponds to one group: bit 0 – group 1, bit 1 – group 2... bit 7 – group 8.

After the execution of this command, the group ID value is set as follows:

- Bit 0 is set to 1, if (group) 1 occurs in the parenthesis, else it is set to 0
- Bit 1 is set to 1, if (group) 2 occurs in the parenthesis, else it is set to 0
- ...
- Bit 7 is set to 1, if (group) 8 occurs in the parenthesis, else it is set to 0.

Example

```
GROUPID (1, 3);     //local axis belongs to groups 1 and 3
...
[G3] {STOP;}        //send stop command to all axes from group 3
                    //the command is executed by local axis too
```

6.2.5.1.81. INITCAM

Syntax

INITCAM *LoadAddress, RunAddress* Copy cam table from EEPROM to RAM

Operands *LoadAddress*: 16-bit unsigned integer - cam table start address in the EEPROM
 RunAddress: 16-bit unsigned integer - cam table start address in the RAM

Type

MPL Program	On-line
X	X

Binary code

INITCAM *LoadAddress, RunAddress*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	1	0	0	0	0	0	0
<i>Load address</i>															
<i>Run address</i>															

Description Prepares a cam table for use. The cam tables are first downloaded into the EEPROM memory of the drive/motor, together with the rest of the MPL program. Then using **INITCAM** command a cam table is copied from the EEPROM memory into the drive/motor RAM memory. The *LoadAddress* is the start address in the EEPROM memory where the cam table was downloaded and the *RunAddress* is the start address in the RAM memory where to copy the cam table. After the execution of this command the MPL variable **CAMSTART** takes the value of the *RunAddress*

Example

```
INITCAM 18864,2560;       //Copy CAM table from EEPROM memory  
                          //(address 0x49B0) to RAM memory  
                          //(address 0xA00)
```

6.2.5.1.82. LOCKEEPROM

Syntax

LOCKEEPROM *value16* **LOCK/unlock EEPROM**

Operands *value16*: integer value between 0 and 3

Type	MPL Program	On-line
	X	X

Binary code

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	0	1	1	0	0	0	0	0	0	0	0	<i>Value16</i>

Description Locks or unlocks the EEPROM write protection. When the EEPROM is write-protected, it is not possible to write data into the EEPROM, with the exception of the MPL command **SAVE**. *Value16* may have the following values:

- 0 – Disables EEPROM write protection
- 1 – Enables write protection for the last quarter of the EEPROM
- 2 – Enables write protection for the last half of the EEPROM
- 3 – Enables write protection for the entire EEPROM

Example

```
//An EEPROM has 8Kwords. In the MPL program space occupies the
//address range: 4000-5FFFh.
LOCKEEPROM 0; // disable EEPROM write protection
LOCKEEPROM 1; // protect the address range: 5800-5FFFh,
LOCKEEPROM 2; // protect the address range: 5000-5FFFh
LOCKEEPROM 3; // protect the entire address range: 4000-5FFFh
```

6.2.5.1.83. LPLANE

Only available on multi-axis Motion Controller

Syntax

LPLANE (*X_axis*, *Y_axis*, *Z_axis*) Linear interpolation **PLANE**

Operands *X_axis*, *Y_axis*, *Z_axis*: slave axes defining the coordinate system

Type

MPL Program	On-line
X	X

Binary code

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

Description Sets the 2D/3D coordinate system for Linear Interpolation Mode using slave axes specified with *X_axis*, *Y_axis* and *Z_axis*.

Example

```
// 2D linear interpolated profile. Position feedbacks: 500 lines
//incremental encoder

SETMODE 0xCF00; //Clear buffer
LPLANE (A, C); //Slaves A and C define the coordinate system
MODE LI; // Set Linear Interpolation Mode
// Increment position with (X, Y) = (0.5[rot], 0.05[rot])
LPOS1 1000L, 100L; LPOS2 1000L, 100L;
UPD; //Execute immediate
// Increment position with (X, Y) = (0.05[rot], 0.5[rot])
LPOS1 100L, 1000L; LPOS2 100L, 1000L;
// Increment position with (X, Y) = (0.5[rot], 0.1[rot])
LPOS1 1000L, 200L; LPOS2 1000L, 200L;
// Increment position with (X, Y) = (0.5[rot], 0.5[rot])
LPOS1 1000L, 1000L; LPOS2 1000L, 1000L;
```

6.2.5.1.84. LPOS

Only available on multi-axis Motion Controller

Syntax

LPOS1 *Pos_X, Pos_Y, Pos_Z* 3D Linear interpolation **POS** segment

LPOS2 *Pos_X, Pos_Y, Pos_Z*

LPOS1 *Pos_X, Pos_Y* 2D Linear interpolation **POS** segment

LPOS2 *Pos_X, Pos_Y*

Operands *Pos_X*: X axis position increment for 2D/3D trajectory
 Pos_Y: Y axis position increment for 2D/3D trajectory
 Pos_Z: Z axis position increment for 3D trajectory

Type	MPL Program	On-line
	X	X

Binary code

LPOS1 *Pos_X, Pos_Y, Pos_Z*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0
LOWORD(M)															
HIWORD(M)															
LOWORD(X_PROJ)															
HIWORD(X_PROJ)															

LPOS2 *Pos_X, Pos_Y, Pos_Z*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	1	1	0	0	0	0	0	0
LOWORD(Y_PROJ)															
HIWORD(Y_PROJ)															
LOWORD(Z_PROJ)															
HIWORD(Z_PROJ)															

LPOS1 *Pos_X, Pos_Y*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0
LOWORD(M)															
HIWORD(M)															
LOWORD(X_PROJ)															
HIWORD(X_PROJ)															

LPOS2 *Pos_X, Pos_Y*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	1	1	0	0	0	0	0	0
LOWORD(Y_PROJ)															
HIWORD(Y_PROJ)															

LPOS1 <i>Pos_X, Pos_Y, Pos_Z</i>															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0
LOWORD(M)															
HIWORD(M)															
LOWORD(X_PROJ)															
HIWORD(X_PROJ)															

LPOS2 <i>Pos_X, Pos_Y, Pos_Z</i>															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	1	1	0	0	0	0	0	0
LOWORD(Y_PROJ)															
HIWORD(Y_PROJ)															
LOWORD(Z_PROJ)															
HIWORD(Z_PROJ)															

LPOS1 <i>Pos_X, Pos_Y</i>															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0
LOWORD(M)															
HIWORD(M)															
LOWORD(X_PROJ)															
HIWORD(X_PROJ)															

LPOS2 <i>Pos_X, Pos_Y</i>															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	1	1	0	0	0	0	0	0
LOWORD(Y_PROJ)															
HIWORD(Y_PROJ)															

Description LPOS1 and LPOS2 define a segment for 2D/3D trajectory executed in Linear Interpolation mode. Based on Pos_X, Pos_Y and Pos_Z the MPL compiler from MotionPRO Developer computes the actual parameters used by the motion controller to generate the PVT points for slave axes.

If the points are sent from a host then the following relations must be used to compute the actual parameters of the segment:

$$M = \sqrt{\text{Pos_X}^2 + \text{Pos_Y}^2 + \text{Pos_Z}^2}$$

$$X_PROJ = \frac{\text{Pos_X}}{M}$$

$$Y_PROJ = \frac{\text{Pos_Y}}{M}$$

$$Z_PROJ = \frac{\text{Pos_Z}}{M}$$

Example

```
// 2D linear interpolated profile. Position feedbacks: 500 lines
```

```
//incremental encoder

SETMODE 0xCF00; //Clear buffer
LPLANE (A, C); //Slaves A and C define the coordinate system
MODE LI; // Set Linear Interpolation Mode
// Increment position with (X, Y) = (0.5[rot], 0.05[rot])
LPOS1 1000L, 100L; LPOS2 1000L, 100L;
UPD; //Execute immediate
// Increment position with (X, Y) = (0.05[rot], 0.5[rot])
LPOS1 100L, 1000L; LPOS2 100L, 1000L;
// Increment position with (X, Y) = (0.5[rot], 0.1[rot])
LPOS1 1000L, 200L; LPOS2 1000L, 200L;
// Increment position with (X, Y) = (0.5[rot], 0.5[rot])
LPOS1 1000L, 1000L; LPOS2 1000L, 1000L;
```

6.2.5.1.85. MODE CS

Syntax

MODE CS Set axis in **MODE** Camming Slave

Operands –

Type	MPL Program	On-line
	X	X

Binary code

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

Description Sets the drive/motor to operate in the electronic camming slave mode. In this mode, the drive/motor performs a position control with reference set by a cam profile function of the master position. The cam profile is defined by a cam table – a set of (X, Y) points, where X is cam table input i.e. the master position and Y is the cam table output i.e. the corresponding slave position. Between the points the drive/motor performs a linear interpolation.

The new motion mode becomes effective at the next update command **UPD**.

Example

```
// Electronic camming slave. Master position is read from 2nd
// encoder inputs. Master resolution: 2000 counts/rev
CAMSTART = 0xF000; //Initialize CAM table start address
EXTREF 2; // master position read from P&D or 2nd encoder
CAMOFF = 200; //Cam offset from master
CAMX = 0.5; //Cam input correction factor
CAMY = 1.5; //Cam output correction factor
MASTERRES = 2000; // master resolution
MODE CS; //Set electronic camming slave mode
TUM1; //Set Target Update Mode 1
SRB ACR, 0xEFFF, 0x0000; //Camming mode: Relative
UPD; //execute immediate
```

```
// Increment position with (X, Y) = (0.5[rot], 0.1[rot])
LPOS1 1000L, 200L; LPOS2 1000L, 200L;
// Increment position with (X, Y) = (0.5[rot], 0.5[rot])
LPOS1 1000L, 1000L; LPOS2 1000L, 1000L;
```


The new motion mode becomes effective at the next update command **UPD**.

Example

```
// S-curve profile. Position feedback: 500 lines incremental
// encoder (2000 counts/rev)
TJERK = 50;//jerk = 2e+004[rad/s^3]
CACC = 0.3183;//acceleration rate = 1000[rad/s^2]
CSPD = 33.3333;//slew speed = 1000[rpm]
CPOS = 20000;//position command = 10[rot]
CPR; //position command is relative
MODE PSC; // set S-curve profile mode
SRB ACR, 0xFFFF, 0x0000; //Stop using an S-curve profile
UPD; //execute immediate
!MC; WAIT!; //wait for completion
```

6.2.5.1.92. MODE PT

Syntax

MODE PT **MODE** Position Time

Operands –

Type	MPL Program	On-line
	X	X

Binary code

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

Description Sets the drive/motor to operate in the PVT mode. In this mode, the drive/motor performs a position control. The built-in reference generator computes a positioning path using a series of points. Each point specifies the desired **Position**, and **Time**, i.e. contains a **PT** data. Between the PT points the reference generator performs a linear interpolation.

The new motion mode becomes effective at the next update command **UPD**.

Example

```
// PT sequence. Position feedback: 500 lines incremental
// encoder (2000 counts/rev)
SETPVT 0xC000; //Clear PT buffer, disable counter check
                //Don't change counter & buffer low condition
MODE PT; // Set PT Mode
TUM1;//Start from actual value of position reference
CPR;
PTP 2000L, 100U, 0; //PT(1[rot], 0.1[s])
UPD; //Execute immediate
PTP 0L, 100U, 0; //PT(1[rot],0.2[s])
PTP -2000L, 100U, 0; //PT(0[rot],0.3[s])
!MC; WAIT!; //wait for completion
```

6.2.5.1.99. MODE TT

Syntax

MODE TT MODE Torque Test

Operands –

Type	MPL Program	On-line
	X	X

Binary code

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

Description Sets the drive/motor to operate in torque test mode. In this mode a current command can be set using a test reference consisting of a limited ramp. For AC motors (like for example the brushless motors), the test mode offers also the possibility to rotate a current reference vector with a programmable speed. As result, these motors can be moved in an “open-loop” mode without using the position sensor. The main advantage of this test mode is the possibility to conduct in a safe way a series of tests, which can offer important information about the motor parameters, drive status and the integrity of the its connections.

***Remark:** The torque test mode has been foreseen to facilitate the testing during the setup phase. It is not intended for normal operation*

The new motion mode becomes effective at the next update command **UPD**.

Example

```
//Torque test mode, brushless AC motor. The drive has
//peak current 16.5A -> 32736 IU (internal current units)
//360° electric angle -> 65536 internal units
// fast loop sampling period = 0.1ms. Motor has 2 pole pairs
MODE TT; //Torque Test Mode
REFTST_A = 1984;//Reference saturation = 1[A]
RINCTST_A = 20;//Reference increment = 10[A/s]
THTST = 0;//Electric angle = 0[deg]
TINCTST = 7;//Electric angle increment ~= 2e+002[deg/s]
UPD; //update immediate
```


Operands –

Type	MPL Program	On-line
	X	X

Binary code

MODE VEF

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

MODE VES

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

Description Sets the drive/motor to operate in the voltage external mode. In this mode, the drive/motor performs a voltage control with a voltage reference provided by another device. There are 2 types of external references (selectable via the MPL instruction EXTREF):

- Analogue – read from a dedicated analogue input (MPL variable **AD5**)
 - Online – received online via a communication channel from a host and saved in the MPL variable **EREFT**

When the voltage reference is read from the analogue input, the reference update can be done in 2 ways:

- **MODE VES** – at each slow loop sampling period
- **MODE VEF** – at each fast loop sampling period

When the voltage reference is received online via a communication channel only option **MODE TES** is possible.

***Remark:** The voltage contouring mode has been foreseen for testing during the setup phase*

The new motion mode becomes effective at the next update command **UPD**.

Example

```
//Read voltage reference from variable EREFV
EREFV = 30; // EREFV initial = 30[IU]
EXTREF 0;
MODE VES; //External voltage
UPD; //execute immediate
```


6.2.5.1.105. OUT

Syntax

OUT(n1, n2, ...) = value16 **OUT**put value16 to I/O n1, n2, ...

OUT(n1, n2, ...) = VAR16 **OUT**put VAR16 value to n1, n2, ...

Operands n1, n2: IO line number
 value16: 16-bit integer immediate value
 VAR16: 16-bit integer variable

Type	MPL Program	On-line
	X	X

Binary code

OUT(Outputs_Mask) = value16

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	0	0	1	0	0	0	0	1	0	0	1
Outputs_mask															
value16															

OUT(Outputs_Mask) = VAR16

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	0	0	1	0	0	0	0	1	0	0	1
Outputs_mask															
&VAR16															

Description The instruction sets one or several output lines simultaneously with the immediate value or the value of the specified variable.

Each bit from the the output line has associated through its number identifier associated an control bit identified If the above bits from VAR are set to 1, the corresponding outputs are set high (1), else the outputs are set low (0).

In MPL the output lines are numbered from 0 to 15. Each product has a specific number of outputs, therefore only a part of the 15 output lines is used.

Warning! Check carefully your drive/motor for the available outputs. Do not use this command if any of the above outputs is not available. You can always set separately each of the outputs using the **OUT** command

This instruction uses a 9-bit **short address** for the destination variable. Bit 9 value **X** specifies the destination address range:

X	Address range of MPL data
0	from 0x0200 to 0x03FF
1	from 0x0800 to 0x09FF

Example

```
int user_var;
user_var = 0x800A; // setup user_var variable
OUTPORT user_var; // Send variable address to external output port
// The command sets high the outputs: #25/Ready, #31 and #29
// and low the outputs: #12/Error, #30 and #28
```

6.2.5.1.106. OUTPORT

Syntax

OUTPORT VAR16 **OUT**put VAR16 value to IOPORT

Operands VAR16: 16-bit integer variable

Type	MPL Program	On-line
	X	X

Binary code

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	1	X	(9LSBs of &VAR16)								

Description The instruction sets simultaneously the following drive/motor output lines:

- Ready output (#25/READY) – set by bit 15 from VAR16
- Error output (#12/ERROR) – set by bit 14 from VAR16
- General-purpose outputs: #31, #30, #29, #28 – set by bits 3, 2, 1, and 0 from VAR16

If the above bits from VAR are set to 1, the corresponding outputs are set high (1), else the outputs are set low (0).

In MPL the I/O lines are numbered: #0 to #39. Each product has a specific number of inputs and outputs, therefore only a part of the 40 I/O lines is used. The I/O numbering is common for all the products; hence each product has its own list of available I/Os.

Warning! Check carefully your drive/motor for the available outputs. Do not use this command if any of the above outputs is not available. You can always set separately each of the outputs using the **OUT** command

This instruction uses a 9-bit **short address** for the destination variable. Bit 9 value **X** specifies the destination address range:

X	Address range of MPL data
0	from 0x0200 to 0x03FF
1	from 0x0800 to 0x09FF

Example

```
int user_var;
user_var = 0x800A; // setup user_var variable
OUTPORT user_var; // Send variable address to external output port
// The command sets high the outputs: #25/Ready, #31 and #29
// and low the outputs: #12/Error, #30 and #28
```

6.2.5.1.107. PING/PONG

Syntax

PING *value16* Request the axis ID and firmware version from a group of axes
 – Answer to PING

Operands *value16*: 16 bit immediate value, used to compute each axis answer delay

Type	MPL Program	On-line
	—	X

Remark: The online instructions are intended only for host/master communication and cannot reside in a MPL program. Therefore their syntax is fictive, the only goal being to identify these commands.

In the [Command interpreter](#), you can check which drives/motors are connected in your network by sending a **PING** request with syntax **PING value16**. The **PONG** answers from all the axes are displayed in the [output window](#).

Binary code

PING <i>value16</i>															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	1	0	0	0	0	0	0	0	0	0
0	0	0	0	AxisID								0	0	0	H
value16															

PONG															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	1	0	Expeditor AxisID							
ASCII code of first 2 digits of the firmware ID															
ASCII code of last digit + revision letter of the firmware ID															

Description By broadcasting a **PING** command, the host/master can find the axis ID of all the drives/motors present in the network. When the PING request is sent via an RS-232 link the host bit (**H**) from the expeditor address must be set to 1. For details, see [serial communication protocol](#) description.

The operand of **PING**, **value16**, represents a time interval measured in μs . This time is multiplied with the axis ID of each axis to provide a time delay for sending the **PONG** answer. For example if value16 is 2000 then the drive/motor with axis ID = 100 will answer after a delay of $100 \times 2000\mu\text{s} = 0.2\text{s}$. The time delay is necessary only if the host is connected via an RS-232 link with one of the drives/motors from a CAN-bus network. If the host is directly connected on the CAN-bus network, value16 can be 0.

Remark: If the **PING** command is sent from the Command Interpreter without operand, the value16 is set by default at 2000. This value corresponds to the highest serial baud rate of 115200. For smaller baud rates the value16 must be increased proportionally.

Each axis will answer to a **PING** command with a **PONG** message, which provides the Axis ID and the firmware version of the expeditor. The firmware version has the form: FxyzA, where xyz is the firmware number (3 digits) and A is the firmware revision. The **PING** message will include the ASCII code of 4 characters: 3 digits for the firmware number + 1 letter for the firmware revision.

6.2.5.1.108. PTP

Syntax

PTP *P_value, T_value, C_value* Define a PT point via immediate values

PTP *P_var, T_var, C_value* Define a PT point via MPL variables

Operands *P_value* – 32-bit long integer immediate value: PT point position
T_value – 16-bit unsigned integer immediate value: PT point time
C_value – 7-bit integer immediate value, PVT point integrity counter
P_var – long variable, contains the PT point position
T_var – integer variable, contains the PVT point time

Type

MPL Program	On-line
X	X

Binary code

PTP *P_value, T_value, C_value*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	1	0	0	0	<i>C_value</i>						
LOWORD(<i>P_value</i>)															
HIWORD(<i>P_value</i>)															
<i>T_value</i>															

PTP *P_var, T_var, C_value*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	1	0	0	0	<i>C_value</i>						
& <i>P_var</i>															
& <i>T_var</i>															

Description Defines a PT point. The PT position and time values may be provided either as immediate values or via the values of 2 MPL variables.

A PT point also includes a 7-bit *integrity counter*. The host must increment by one the integrity counter each time when it sends a new PT point. If the integrity counter error checking is activated, every time when the drive/motor receives a new PT point, it compares its internally computed integrity counter value with the one sent with the **PTP** command. The PT point is accepted only if the two values are equal. If the values of the two integrity counters do not match, the integrity check error is triggered, the drive/motor sends the **PVTSTS** to the host with **PVTSTS.12 =1** and the PT point received is discarded. Each time a PT point is accepted, the drive/motor automatically increments its internal integrity counter.

Example

```
SETPT 0xCF00;    //Clear PT buffer
MODE PT;        // Set PT Mode
TUM1;           //Start from actual value of position reference
CPR;
PTP 2000L, 2000U, 0;  //PT(1[rot], 2[s])
UPD;            //Execute immediate
PTP 6000L, 500U, 0;  //PT(4[rot],2.5[s])
PTP -2000L, 500U, 0; //PT(3[rot],3[s])
!MC; WAIT!;      //wait for completion
```

6.2.5.1.109. PVTP

Syntax

PVTP *P_value, V_value, T_value, C_value* Define a PVT point via immediate values

PVTP *P_var, V_var, T_var, C_value* Define a PVT point via MPL variables

- Operands**
- P_value* – 24-bit long integer immediate value: PVT point position
 - V_value* – 24-bit fixed immediate value (16MSB integer part and 8LSB fractional part): PVT point velocity
 - T_value* – 9-bit integer immediate value: PVT point time
 - C_value* – 7-bit integer immediate value, PVT point integrity counter
 - P_var* – long variable, contains the PVT point position
 - V_var* – fixed variable, contains the PVT point velocity
 - T_var* – integer variable, contains the PVT point time

Type

MPL Program	On-line
X	X

Binary code

PVTP P_value, V_value, T_value, C_value

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	0	0	0	<i>C_value</i>						
<i>P_value</i> (16LSB)															
<i>P_value</i> (8MSB)								<i>V_value</i> (8LSB)							
<i>V_value</i> (16MSB)															
0	0	0	0	0	0	0	0	<i>T_value</i>							

PVTP P_var, V_var, T_var, C_value

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	0	1	0	<i>C_value</i>						
<i>&P_var</i>															
<i>&V_var</i>															
<i>&T_var</i>															

Description Defines a PVT point. The PVT position, velocity and time values may be provided either as immediate values or via the values of 3 MPL variables.

A PVT point also includes a 7-bit *integrity counter*. The host must increment by one the integrity counter each time when it sends a new PVT point. If the integrity counter error checking is activated, every time when the drive/motor receives a new PVT point, it compares it's internally computed integrity counter value with the one sent with the **PVTP** command. The PVT point is accepted only if the two values are equal. If the values of the two integrity counters do not match, the integrity check error is triggered, the drive/motor sends the **PVTSTS** to the host with **PVTSTS.12 = 1** and the PVT point received is discarded. Each time a PVT point is accepted, the drive/motor automatically increments its internal integrity counter.

Example

```
SETPVT 0xCF00;    //Clear PVT buffer
MODE PVT;        // Set PVT Mode
TUM1;           //Start from actual value of position reference
CPR;
PVTP 12000L, 0.04, 300U, 0;//PVT(6[rot], 1.199[rpm], 0.3[s])
UPD; //Execute immediate
PVTP -8000L, 0, 200U, 0;//PVT(2[rot], 0[rpm], 0.5[s])
!MC; WAIT!; //wait for completion
```

6.2.5.1.112. REMGRID

Syntax

REMGRID (*value_1*, *value_2*, ...) **REMO**ve specified groups from **GR**oup **ID**

Operands *value_1*, *value_2*: specify a group number between 1 and 8

Type

MPL Program	On-line
X	X

Binary code

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0
<i>Value</i>															

Description The command removes groups from the **group ID**. On each axis, the group ID represents a filter for a multicast transmission. When a multicast message is received, each axis compares the message group ID with its own group ID. If the two group IDs have at least one group in common, the message is accepted. For example, if an axis is member of group 1 and group 3, it will receive all the messages sent with a group ID including group 1 or group 3. The group ID is an 8-bit integer value. Each bit corresponds to one group: bit 0 – group 1, bit 1 – group 2... bit 7 – group 8.

After the execution of this command, the group ID value is modified as follows:

- Bit 0 is set to 0, if (group) 1 occurs in the parenthesis
- Bit 1 is set to 0, if (group) 2 occurs in the parenthesis
- ...
- Bit 7 is set to 0, if (group) 8 occurs in the parenthesis.

Example

```
GROUPID (8); //local axis belongs to groups 8
ADDGRID (2, 5); //local axis belongs to groups 2, 5 and 8
...
REMGRID (5, 8); //local axis belongs only to group 2
```

6.2.5.1.114. RET

Syntax

RET **RET**urn from a MPL function

Operands –

Type	MPL Program	On-line
	X	—

Binary code

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

Description Ends the execution of a MPL function and performs the return to the next MPL instruction after the function call.

Example

```
...
CALL Function1;    // Call Function1
UPD;                // Update immediate is next instruction
                    // executed after RET
...
Function1:
...
RET;                //Exit from Function1
```

6.2.5.1.115. RETI

Syntax

RETI **RET**urn from a MPL Interrupt function

Operands –

Type	MPL Program	On-line
	X	—

Binary code

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

Description Ends the execution of a MPL ISR and returns to the MPL command whose execution was postponed by the MPL interrupt. RETI globally enables the MPL interrupts which were globally disabled when the MPL interrupt was accepted and the ISR was called.

Example

```
Int5_WrapAround:    // Int5 ISR: position wraparound  
                  AXISOFF;  
                  RETI;            // return from MPL ISR
```


Description **ROUT#n** instruction sets low (0 logic) the output line number *n*. In MPL the I/O lines are numbered: #0 to #39. Each product has a specific number of inputs and outputs, therefore only a part of the 40 I/O lines is used. The I/O numbering is common for all the products; hence each product has its own list of available I/Os.

Example

```
ROUT#28;           //Reset output line #28 to 0 (set low)
```

AND/OR masks for ROUT#n							
PxDATDIR	#n	ANDrst	ORrst	PxDATDIR	#n	ANDrst	ORrst
0X7098	#0	0xFFFE	0x0000	0X709C	#20	0xFFEF	0x0000
0X07098	#1	0xFFFD	0x0000	0X709C	#21	0xFFDF	0x0000
0X7098	#2	0xFFFB	0x0000	0X709C	#22	0xFFBF	0x0000
0X7098	#3	0xFF7F	0x0000	0X709C	#23	0xFF7F	0x0000
0X7098	#4	0xFFEF	0x0000	0X709E	#24	0xFFFE	0x0000
0X7098	#5	0xFFDF	0x0000	0X7095	#25	0xFFFE	0x0000
0X7098	#6	0xFFBF	0x0000	0X7095	#26	0xFFFD	0x0000
0X7098	#7	0xFF7F	0x0000	0X7095	#27	0xFFFB	0x0000
0X709A	#8	0xFFFE	0x0000	0X7095	#28	0xFF7F	0x0000
0X709A	#9	0xFFFD	0x0000	0X7095	#29	0xFFEF	0x0000
0X709A	#10	0xFFFB	0x0000	0X7095	#30	0xFFDF	0x0000
0X709A	#11	0xFF7F	0x0000	0X7095	#31	0xFFBF	0x0000
0X709A	#12	0xFFEF	0x0000	0X7095	#32	0xFF7F	0x0000
0X709A	#13	0xFFDF	0x0000	0X7096	#33	0xFFFE	0x0000
0X709A	#14	0xFFBF	0x0000	0X7096	#34	0xFFFD	0x0000
0X709A	#15	0xFF7F	0x0000	0X7096	#35	0xFFFB	0x0000
0X709C	#16	0xFFFE	0x0000	0X7096	#36	0xFF7F	0x0000
0X709C	#17	0xFFFD	0x0000	0X7096	#37	0xFFEF	0x0000
0X709C	#18	0xFFFB	0x0000	0X7096	#38	0xFFDF	0x0000
0X709C	#19	0xFF7F	0x0000	0X7096	#39	0xFFBF	0x0000

Example

```
// Position profile. Position feedback: 500 lines incremental
// encoder (2000 counts/rev)
CACC = 0.3183;//acceleration rate = 1000[rad/s^2]
CSPD = 33.3333;//slew speed = 1000[rpm]
CPOS = 6000;//position command = 3[rot]
CPR; //position command is relative
MODE PP;
TUM1; //set Target Update Mode 1
UPD; //execute immediate
!MC; // set event when motion is completed
WAIT!;//Wait until the event occurs i.e. the motor stops
// At this point TPOS=6000, APOS = 6000-POSERR
SAP 2000; // Set actual position 1[rot]
// Now, TPOS=2000, APOS=2000-POSERR
```

6.2.5.1.119. SAVE

Syntax

SAVE Save setup data in the EEPROM

Operands –

Type	MPL Program	On-line
	X	X

Binary code

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

Description Saves the actual values of all the MPL parameters with setup data from the active data RAM memory into the EEPROM memory, in the setup table. Through this command, you can save all the setup modifications done, after the power on initialization.

Example

```
SAVE; // Save setup data in the EEPROM setup table
```

6.2.5.1.120. SCIBR

Syntax

SCIBR *value16* Set **S**erial **C**ommunication **I**nterface **B**aud **R**ate to *value16*

SCIBR *VAR16* Set **S**erial **C**ommunication **I**nterface **B**aud **R**ate to *VAR16*

Operands *value16*: 16-bit integer immediate value between 0 and 4
VAR16: integer variable

Type	MPL Program	On-line
	X	X

Binary code

SCIBR <i>value16</i>															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0
<i>Value16</i>															

SCIBR <i>VAR16</i>															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	1	0	0	1	0	0	0	0	0
<i>&VAR16</i>															

Description Sets the baud rate on the RS232/RS485 serial communication interface (SCI). The new baud rate can be provided either as an immediate value or by the value of a MPL variable. In both cases, the possible values are:

<i>Value16</i>	SCI baud rate
0	9600
1	19200
2	38400
3	56600
4	115200

The serial baud rate is set at power on using the following algorithm:

- a. With the value read from the EEPROM setup table
- b. If the setup table is invalid, with the last baud rate read from a valid setup table
- c. If there is no baud rate set by a valid setup table, with 9600.

Remarks:

- *Use this command when a drive/motor operates in AUTORUN (after power on starts to execute the MPL program from the EEPROM) and it must communicate with a host at a baud rate different from the default value. In this case, the MPL program must start with a serial baud rate change.*

-
- *An alternate solution to the above case is to set via **SCIBR** command the desired baud rate and then to save it in the EEPROM, with the command **SAVE**. After a reset, the drive/motor starts directly with the new baud rate, if the setup table was valid. Once set, the new default baud rate is preserved, even if the setup table is later on disabled, because the default serial baud rate is stored in a separate area of the EEPROM.*

Example

```
SCIBR 4;           // sets the SCI baud rate to 115200 baud
```


6.2.5.1.121. SEG

Syntax

SEG *D_time, D_ref* Define a contouring segment via immediate values

SEG *VAR16, VAR32* Define a contouring segment via MPL variables

Operands *D_time* –16-bit unsigned integer value: segment time
D_ref: 32-bit fixed immediate value: segment reference increment per time unit
VAR16 – 16-bit integer variable: segment time
VAR32 – 32-bit fixed variable: segment reference increment per time unit

Type	MPL Program	On-line
	X	—

Binary code

SEG <i>D_time, D_ref</i>															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0
<i>D_time</i>															
LOWORD(<i>D_ref</i>)															
HIWORD(<i>D_ref</i>)															

SEG <i>VAR16, VAR32</i>																	
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
0	1	1	1	1	1	X	(9LSBs of & <i>VAR16</i>)										
& <i>VAR32</i>																	

Description Defines a contouring segment. The **time** and the reference **increment** per time unit may be provided either as immediate values or via the values of 2 MPL variables. The **time** represents the segment duration expressed in [time units](#) i.e. in number of slow loop sampling periods. The reference **increment** represents the amount of reference variation per time unit i.e. per slow loop sampling period.

SEG *VAR16, VAR32* uses a 9-bit **short address** for the operand. Bit 9, value **X**, specifies the destination address range:

X	Address range of MPL data
0	from 0x0200 to 0x03FF
1	from 0x0800 to 0x09FF

Example

```
// Position contouring with position feedback on motor: 500 lines
// incremental encoder (2000 counts/rev)
MODE PC;//Set Position Contouring
TUM1;//Start from actual value of position reference
SEG 100U, 20.00000; //1st segment. At its end, TPOS increases with
// 20*100 = 2000 counts (i.e. 1 rev)
UPD; //Execute immediate
SEG 100U, 0.00000; // 2nd segment. At its end TPOS remains the same SEG
0, 0.0; //End of contouring
```

6.2.5.1.122. SEND

Syntax

SEND VAR16 **SEND** the content of VAR16

SEND VAR32 **SEND** the content of VAR32

Operands VAR16: integer variable
 VAR32: long/fixed variable

Type	MPL Program	On-line
	X	—

Binary code

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	X	(9LSBs of &VAR16)								

Description When the instruction is encountered, the content of VAR16/VAR32 is sent using “Take Data 2” message type. The instruction uses a 9-bit **short address** for the destination variable. Bit value **X** specifies the destination address range:

X	Address range of MPL data
0	from 0x0200 to 0x03FF
1	from 0x0800 to 0x09FF

Execution The value of VAR16/VAR32 is sent using “Take Data 2” message.

Example

```
MASTERID = 33; // Set host ID / address = 2
//Send SRH & SRL if motion complete or pos. trigger 1 bits change
SRH_MASK = 0x0002;
SRL_MASK = 0x0400;
MER_MASK = 0xFFFF; // send MER on any bit change
SEND CAPPOS; // Send to host contents of variable CAPPOS
```

6.2.5.1.123. SetAsInput

Syntax

SetAsInput(n1, n2, n3,...) **SetAsInput** the I/O lines numbers n1, n2, n3

Operands n1, n2, n3: IO line number. It specifies the position of the control bit associated to the I/O line in the IO_mask.

Type	MPL Program	On-line
	X	X

Binary code

SetAsInput(IO_Mask)															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	0	0	0	0	0	0	0	0
IO_Mask															

Description Some drives/motors include I/O lines that may be used either as inputs or as outputs. Before using these lines as inputs you have to use the **SetAsInput** command with the input numbers as argument. The input numbers identifies the corresponding bit from the IO_mask, i.e. input number 2 has associated bit 2 in the IO_mask. In MPL the input lines are numbered from 0 to 15.

Remarks:

- Check the drive/motor user manual to find how are set, after power-on, the I/O lines that may be used either as inputs or as outputs. Each product has a specific number of inputs, therefore only a part of the 15 input lines is used.
- You need to set an I/O line as input, only once, after power on

Example

```
SetAsInput(2,5); //Set IO line 2 and 5 as inputs  
v1 = IN(2); //Read I/O line 2 data into variable v1
```

6.2.5.1.124. SetAsOutput

Syntax

SetAsOutput(n1, n2, n3,...) **SetAsOutput** I/O lines numbered **n1, n2, n3**

Operands *n1, n2, n3*: IO line number. It specifies the position of the control bit associated to the I/O line in the IO_mask.

Type

MPL Program	On-line
X	X

Binary code

SetAsOutput(IO_Mask)															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	0	0	0	0	0	0	0	0
IO_Mask															

Description Some drives/motors include I/O lines that may be used either as inputs or as outputs. Before using these lines as outputs you have to use the **SetAsOutput** command having as argument the output lines numbers. The output lines numbers identifies the control bit from the IO_mask, i.e. output number 7 has associated bit 7 in the IO_mask, setting to 1 bit 7 the IO line will be used as output. In MPL the output lines are numbered from 0 to 15.

Remarks:

- Check the drive/motor user manual to find how are set, after power-on, the I/O lines that may be used either as inputs or as outputs. Each product has a specific number of inputs and outputs, therefore only a part of the 15 output lines is used.
- You need to set an I/O line as output, only once, after power on

Example

```
SerAsOutput(7); //Set IO line 7 as output
Out(4,7)=0x0090; //Set I/O lines 4 and 7 to High.
```

6.2.5.1.125. SETIO

Syntax

SETIO#n IN **SETIO#n** as Input port
SETIO#n OUT **SETIO#n** as OUTput port

Operands *n*: I/O number (0<=*n*<=39)

Type	MPL Program	On-line
	X	X

Binary code

SETIO#n IN															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	0	0	0	0	0	0	0	0	0	0
<i>PxDATDIR</i>															
<i>ANDin</i>															
<i>ORin</i>															

SETIO#n OUT															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	0	0	0	0	0	0	0	0	0	0
<i>PxDATDIR</i>															
<i>ANDout</i>															
<i>ORout</i>															

Description Some drives/motors include I/O lines that may be used either as inputs or as outputs. Before using these lines, you need to specify how you want to use them, with the **SETIO** commands:

```
SETIO#n OUT;      //Set the I/O line #n as an input
SETIO#n IN;      //Set the I/O line #n as an output
```

Remarks:

- *Check the drive/motor user manual to find how are set, after power-on, the I/O lines that may be used either as inputs or as outputs*
- *You need to set an I/O line as input or output, only once, after power on*

In MPL the I/O lines are numbered: #0 to #39. Each product has a specific number of inputs and outputs, therefore only a part of the 40 I/O lines is used. The I/O numbering is common for all the products; hence each product has its own list of available I/Os.

Example

```
SETIO#12 OUT;      //Set IO line 12 as output
ROUT#12;          //Reset IO line 12 level low (0 logic)
SETIO#12 IN;      //Set IO line 12 as input
v1 = IN#12;       //Read I/O line 12 data into variable v1
```

PxDATDIR	AND/OR masks for SETIO#n IN			AND/OR masks for SETIO#n OUT		
	#n	ANDin	ORin	#n	ANDout	ORout
0X7098	#0	0xFEFF	0x0000	#0	0xFFFF	0x0100
0X7098	#1	0xFDFF	0x0000	#1	0xFFFF	0x0200
0X7098	#2	0xFBFF	0x0000	#2	0xFFFF	0x0400
0X7098	#3	0xF7FF	0x0000	#3	0xFFFF	0x0800
0X7098	#4	0xEFFF	0x0000	#4	0xFFFF	0x1000
0X7098	#5	0xDFFF	0x0000	#5	0xFFFF	0x2000
0X7098	#6	0xBFFF	0x0000	#6	0xFFFF	0x4000
0X7098	#7	0x7FFF	0x0000	#7	0xFFFF	0x8000
0X709A	#8	0xFEFF	0x0000	#8	0xFFFF	0x0100
0X709A	#9	0xFDFF	0x0000	#9	0xFFFF	0x0200
0X709A	#10	0xFBFF	0x0000	#10	0xFFFF	0x0400
0X709A	#11	0xF7FF	0x0000	#11	0xFFFF	0x0800
0X709A	#12	0xEFFF	0x0000	#12	0xFFFF	0x1000
0X709A	#13	0xDFFF	0x0000	#13	0xFFFF	0x2000
0X709A	#14	0xBFFF	0x0000	#14	0xFFFF	0x4000
0X709A	#15	0x7FFF	0x0000	#15	0xFFFF	0x8000
0X709C	#16	0xFEFF	0x0000	#16	0xFFFF	0x0100
0X709C	#17	0xFDFF	0x0000	#17	0xFFFF	0x0200
0X709C	#18	0xFBFF	0x0000	#18	0xFFFF	0x0400
0X709C	#19	0xF7FF	0x0000	#19	0xFFFF	0x0800
0X709C	#20	0xEFFF	0x0000	#20	0xFFFF	0x1000
0X709C	#21	0xDFFF	0x0000	#21	0xFFFF	0x2000
0X709C	#22	0xBFFF	0x0000	#22	0xFFFF	0x4000
0X709C	#23	0x7FFF	0x0000	#23	0xFFFF	0x8000
0X709E	#24	0xFEFF	0x0000	#24	0xFFFF	0x0100
0X7095	#25	0xFEFF	0x0000	#25	0xFFFF	0x0100
0X7095	#26	0xFDFF	0x0000	#26	0xFFFF	0x0200
0X7095	#27	0xFBFF	0x0000	#27	0xFFFF	0x0400
0X7095	#28	0xF7FF	0x0000	#28	0xFFFF	0x0800
0X7095	#29	0xEFFF	0x0000	#29	0xFFFF	0x1000
0X7095	#30	0xDFFF	0x0000	#30	0xFFFF	0x2000
0X7095	#31	0xBFFF	0x0000	#31	0xFFFF	0x4000
0X7095	#32	0x7FFF	0x0000	#32	0xFFFF	0x8000
0X7096	#33	0xFEFF	0x0000	#33	0xFFFF	0x0100
0X7096	#34	0xFDFF	0x0000	#34	0xFFFF	0x0200
0X7096	#35	0xFBFF	0x0000	#35	0xFFFF	0x0400
0X7096	#36	0xF7FF	0x0000	#36	0xFFFF	0x0800
0X7096	#37	0xEFFF	0x0000	#37	0xFFFF	0x1000
0X7096	#38	0xDFFF	0x0000	#38	0xFFFF	0x2000
0X7096	#39	0xBFFF	0x0000	#39	0xFFFF	0x4000

6.2.5.1.126. SETMODE

Only available on multi-axis Motion Controller

Syntax

SETMODE *value16* **SET 2D/3D motion MODE**

Operands *value16*: 16-bit integer immediate value

Type

MPL Program	On-line
X	X

Binary code

SETMODE *value16*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	0	1	0	0	0	0	0	0	0	0
<i>Value16</i>															

Description Sets the Vector or Linear Interpolation Mode as specified by *value16*. Value16 has the following significance:

BIT	Value	Description
15	0	Nothing
	1	Clear segment buffer and initialize buffer internal variables. This option is mandatory after changing the start address of segment buffer
14	0	Enable integrity counter error checking
	1	Disable integrity counter error checking
13	0	Leave internal integrity counter unchanged
	1	Change integrity counter with the value specified in bits 0 to 6
12	0	Reserved
11..8	0..15	New level for buffer low signaling. When the number of entries in the segment buffer is equal or less than buffer low level, SEGBUFSTS.14 is set to 1
7	0	Leave buffer low level unchanged
	1	Change the buffer low level with the value specified in bits 8 to 11
6..0	0..127	New integrity counter value

Remark: after *SETMODE* execution, a copy of **value16** is saved in the MPL variable **MACOMMAND**.

Example

```
// 3D linear interpolated profile. Position feedbacks: 500 lines
//incremental encoder

SETMODE 0xCF00; //Clear buffer
LPLANE (A, B, C);
MODE LI; // Set Linear Interpolation Mode
//Increment position with (X, Y, Z) = (0.5[rot], 0.05[rot], 0.05[rot])
LPOS1 1000L, 100L, 100L; LPOS2 1000L, 100L, 100L;
UPD; //Execute immediate
//Increment position with (X, Y, Z) = (0.05[rot], 0.5[rot], 0.05[rot])
LPOS1 100L, 1000L, 100L; LPOS2 100L, 1000L, 100L;
//Increment position with (X, Y, Z) = (0.5[rot], 0.1[rot], 0.25[rot])
LPOS1 1000L, 200L, 500L; LPOS2 1000L, 200L, 500L;
//Increment position with (X, Y, Z) = (0.5[rot], 0.5[rot], 0.5[rot])
LPOS1 1000L, 1000L, 1000L; LPOS2 1000L, 1000L, 1000L;
```

6.2.5.1.127. SETPT

Syntax

SETPT *value16* SETUp PT mode operation

Operands *value16*: 16-bit integer immediate value

Type	MPL Program	On-line
	X	X

Binary code

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	0	0	1	0	0	0	0	0	0	0
<i>Value16</i>															

Description Sets the PT mode operation as specified by the *value16*. *Value16* has the following significance:

BIT	VALUE	DESCRIPTION
15	0	Nothing
	1	Clear PVT buffer and reinitialize the buffer internal variables. This command is mandatory after changing the start address of the PT buffer
14	0	Enable the integrity counter error checking
	1	Disable the integrity counter error checking. When integrity error checking is disabled, the drive stops sending messages when PT buffer becomes full, low or empty.
13	0	No change to the internal integrity counter
	1	Change internal integrity counter with the value specified in bits 0 to 6
12	0	If PT mode is set under CPA (absolute positioning), the initial position is taken from MPL parameter PVTPSO0 (default = 0). The initial position is used to compute the distance to move up to the first PT point
	1	If PT mode is set under CPA (absolute positioning), the initial position is considered the current value of MPL variable TPOS . The initial position is used to compute the distance to move up to the first PT point
11..8	0..15	New parameter for buffer low signaling. When the number of entries in the PT buffer is equal or less than buffer low value, PVTSTS.14 is set to one.
7	0	No change in the buffer low parameter
	1	Change the buffer low parameter with the value specified in bits 8 to11
6..0	0..127	New integrity counter value

Remark: after SETPT execution, a copy of **value16** is saved in the MPL variable **PVTMODE**.

Example

```
SETPT 0xE02F;      //Leave PT buffer intact; Change integrity
                  //counter value to 17
MODE PVT;         // Set PVT Mode
TUM1;//Start from actual value of position reference
CPR;
PVTP 2000L, 0.0667, 500U, 17;//PVT(1[rot], 1.9[rpm], 0.5[s])
UPD;              //Execute immediate
PVTP 0L, 0.0667, 500U, 18;//PVT(1[rot], 1.99997[rpm], 1[s])
PVTP 6000L, 0, 500U, 19;//PVT(4[rot], 0[rpm], 1.5[s])
```

6.2.5.1.128. SETPVT

Syntax

SETPVT *value16* **SETUp PVT** mode operation

Operands *value16*: 16-bit integer immediate value

Type	MPL Program	On-line
	X	X

Binary code

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	0	0	1	0	0	0	0	0	0	0
<i>Value16</i>															

Description Sets the PVT mode operation as specified by the *value16*. *Value16* has the following significance:

BIT	VALUE	DESCRIPTION
15	0	Nothing
	1	Clear PVT buffer and reinitialize the buffer internal variables. This command is mandatory after changing the start address of the PVT buffer
14	0	Enable the integrity counter error checking
	1	Disable the integrity counter error checking
13	0	No change to the internal integrity counter
	1	Change internal integrity counter with the value specified in bits 0 to 6
12	0	If PVT mode is set under CPA (absolute positioning), the initial position is taken from MPL parameter PVTPOS0 (default = 0). The initial position is used to compute the distance to move up to the first PVT point
	1	If PVT mode is set under CPA (absolute positioning), the initial position is considered the current value of MPL variable TPOS . The initial position is used to compute the distance to move up to the first PVT point
11..8	0..15	New parameter for buffer low signaling. When the number of entries in the PVT buffer is equal or less than buffer low value, PVTSTS.14 is set to one.
7	0	No change in the buffer low parameter
	1	Change the buffer low parameter with the value specified in bits 8 to 11
6..0	0..127	New integrity counter value

Remark: after *SETPVT* execution, a copy of **value16** is saved in the MPL variable **PVTMODE**.

Example

```
// PVT sequence. Position feedback: 500 lines incremental
// encoder (2000 counts/rev)
MASTERID = 4081; // Set host address to 255 (255<<4+1)
SETPVT 0xC000; //Clear PVT buffer, disable counter check
                //Don't change counter & buffer low condition

MODE PVT; // Set PVT Mode
TUM1; //Start from actual value of position reference
CPR; // Relative mode
PVTP 400L, 60, 10U, 0; //PVT(0.2[rot], 1800[rpm], 0.01[s])
UPD; //Execute immediate
PVTP 400L, 0, 10U, 0; //PVT(0.4[rot], 0[rpm], 0.02[s])
!MC; WAIT!; //wait for completion
```

6.2.5.1.129. SETSYNC

Syntax

SETSYNC *value16* **SET SYNC**hronization *value16*

Operands *value16*: 16-bit integer immediate value

Type	MPL Program	On-line
	X	X

Binary code

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	1	0	0	0	0	0	0	0	1	0	0
<i>Value16</i>															

Description Enables/disables the transmission of synchronization messages. The drive/motor, were these messages are enabled, is master for the synchronization process. This is performed in two steps. First, the master sends a synchronization message to all axes, including to itself. When this message is received, all the axes read their own internal time. Next, the master sends its internal time to all the slaves, which compare it with their own internal time. If there are differences, the slaves correct slightly their sampling periods in order to keep them synchronized with those of the master. As effect, when synchronization procedure is active, the execution of the control loops on the slaves is synchronized with those of the master within a 10 μ s time interval. Due to this powerful feature, drifts between master and slave axes are eliminated. The MPL command **SETSYNC value** activates the synchronization procedure if **value** is different from 0. Value represents the time interval in internal units between the synchronization messages sent by the master. Recommended value is 20ms. Setting value to 0 stops the master transmissions and deactivates the synchronization procedure.

Remark: *The master for synchronization procedure can be any drive/motor from the network or a host. The master for this process may or may not be the same with the overall motion application master (if present).*

Example

```
SETSYNC 20; //Send synchronization messages every 20[ms]
```


6.2.5.1.132. SRB/SRBL

Syntax

SRB *VAR16, ANDmask, ORmask* **Set/Reset Bits** of *VAR16* (short addressing)

SRBL *VAR16, ANDmask, ORmask* **Set/Reset Bits** of *VAR16* (full addressing)

Operands *VAR16*: integer variable
 ANDmask: 16-bit mask for AND operation
 ORmask: 16-bit mask for OR operation

Type

MPL Program	On-line
X	X

Binary code

SRB *VAR16, ANDmask, ORmask*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	0	X	(9LSBs of &VAR16)								
								<i>ANDm</i>							
								<i>ORm</i>							

SRBL *VAR16, ANDmask, ORmask*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	0	0	0	0	0	0	0	0	0	0
								&VAR16							
								<i>ANDm</i>							
								<i>ORm</i>							

Description Performs a logic AND between *VAR16* and the AND mask, followed by a logic OR between the result and the OR mask. The result is saved in *VAR16*. These instructions may be used to set/reset individual bits from a register or a MPL variable without affecting the other ones. **SRB** performs these operations in a safe way avoiding the interference with the other concurrent processes wanting to change the same MPL data. This is particularly useful for the MPL registers, which have bits that can be manipulated by both drive/motor and user at MPL level.

SRB uses a 9-bit **short address** for the operand. Bit 9 value **X** specifies the destination address range:

X	Address range of MPL data
0	from 0x0200 to 0x03FF
1	from 0x0800 to 0x09FF

All predefined or user-defined MPL data are inside these address ranges, hence this instruction can be used without checking the variables addresses. However, considering future developments, the MPL also includes **SRBL** instruction using a 16-bit **full address** for the operand.

Example

```
int var1;
....
SRB var1, 0xFF0F, 0x0003;    //Reset bits 4 to 7, set bits 0
                               //and 1 of var1
```

6.2.5.1.133. STARTLOG

Syntax

STARTLOG *value* **START LOGGER**

Operands *value*: integer value 1 or 2

Type	MPL Program	On-line
	X	X

Binary code

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	0	0	1	0	1	1	0	0	1	0	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0		<i>value</i>

Description Starts the acquisition of the variables selected in the **Setup Logger Variables** dialogue. *Value* may have the following values:

1 – acquire data at each current loop sampling or from *n* to *n* current loop samplings

2 – acquire data at each position/speed sampling loop or from *n* to *n* position/speed loop samplings

Where *n* is the number of samplings between two consecutive data acquisitions.

Remark: To start the data acquisition simultaneously on all the axes for multi-axis data logging send a broadcast message with the **STARTLOG** command.

Example

```
// In the Setup Logger Variables, the number of samplings between
// data acquisitions is set to 1
STARTLOG 1;            // Save data every current loop sampling
STARTLOG 2;            // Save data every position/speed loop sampling
[b]{STARTLOG 1;} // Start multi-axis logging. The data is saved at
                  // every current loop sampling
```

6.2.5.1.134. STOPLOG

Syntax

STOPLOG **STOP LOGGER**

Operands –

Type	MPL Program	On-line
	X	X

Binary code

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

Description Stops the data acquisition of the variables selected in the **Setup Logger Variables** dialogue. To upload and plot the data saved in the drive's acquisition buffer use the **Logger | Upload Data** menu command.

***Remark:** To stop the data acquisition on all the axes for multi-axis logging, send a broadcast message with the STOPLOG command.*

Example

```
STOPLOG;            // Stop the data acquisition on the current axis  
[b]{STOPLOG;}       // Stop the data acquisition on all the axes.
```

6.2.5.1.135. STA

Syntax

STA Set Target position = Actual position

Operands –

Type	MPL Program	On-line
	X	X

Binary code

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

Description Sets the value of the target position (the position reference) to the value of the actual load position i.e. **TPOS = APOS_LD**. The command may be used in closed loop systems when the load/motor is still following a hard stop, to reposition the target position to the actual load position.

Remark: The **STA** command is automatically done if the next motion mode is set without **TUM1** (i.e. using the default target update mode **TUM0**). In this case the target position and speed are both updated with the actual values of the load position and respectively load speed: **TPOS = APOS_LD** and **TSPD = ASPD_LD**.

Example

```
MODE PC;           //Set Position Contouring Mode 2
TUM1;              //Set target update mode 1
SEG 100U, 5.00000; //Set 1st motion segment. Increment
                  //position reference with 5 counts for
                  //the next 100 sampling periods
UPD;              //Update immediate
SEG 100U, 5.00000; //Set 2st motion segment.
SEG 100U, -20.00000; //Set 3st motion segment.
SEG 100U, 10.00000; //Set 4st motion segment.
SEG 0, 0.;        //End of contouring mode
STA;              //Set target position value (TPOS) equal to //the actual
                  position value (APOS_LD)
```

6.2.5.1.136. STOP

Syntax

STOP! **STOP** motion on event

Operands –

Type	MPL Program	On-line
	X	—

Binary code

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

Description Executes a **STOP** command when a programmed event occurs.

Example:

```
// Move at constant speed and stop when input 36 goes low.
// Position feedback: 500 lines encoder (2000 counts/rev)
CACC = 0.3183; //acceleration rate = 1000[rad/s^2]
CSPD = 33.3333; //jog speed = 1000[rpm]
MODE SP;
TUM1; //set Target Update Mode 1
UPD; //execute immediate
!IN#36 0; // Set event: when input #36 goes low
STOP!; //Stop the motion when event occurs
WAIT!; //Wait until the event occurs
```

6.2.5.1.137. TUM

Syntax

TUM0 Set Target Update Mode 0

TUM1 Set Target Update Mode 1

Operands -

Type	MPL Program	On-line
	X	X

Binary code

TUM0

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

TUM1

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

Description Sets the target update mode 0 or 1. The **TUM0** and **TUM1** instructions offer 2 choices for starting a new motion mode.

After a **TUM1** command, the reference generator computes the new motion mode trajectory starting from the actual values of position and speed reference.

After a **TUM0** command, the reference generator first updates the position and speed references with the actual values of the load position and speed (**TPOS=APOS_LD** and **TSPD=ASPD_LD**) and then starts to compute the new motion mode trajectory.

By default, each command setting a motion mode activates the **TUM0** mode. In order to activate the **TUM1** mode, execute the MPL instruction **TUM1** AFTER the command setting the motion mode and BEFORE the **UPD** command.

As a general rule, **TUM1** mode is recommended for normal operation. Use **TUM0** in the following situations:

- Recovery from an error setting **AXISOFF** command, where significant differences may occur between the last target position and speed values computed by the reference generator before the **AXISOFF** and the actual values of the load/motor position and speed
- Precise relative positioning from the point where the load/motor has hit a marker – to eliminate the following error
- When you start / stop your motor using only **AXISON** / **AXISOFF** commands
- If you switch from a torque control mode (where target position and speed are not computed by the reference generator) to a motion mode performing position or speed control

Remark: In open loop control of steppers, **TUM0** is ignored as there is no position and/or speed feedback

The instructions become effective at the next update command **UPD**.

Example1:

```
// Start a position profile with TUM1. Position feedback:
// 500 lines incremental encoder (2000 counts/rev)
CACC = 0.3183; //acceleration rate = 1000[rad/s^2]
CSPD = 33.3333; //slew speed = 1000[rpm]
CPOS = 6000; //position command = 3[rot]
CPR; //position command is relative
SRB ACR 0xFFFF, 0x800; // and additive
MODE PP; // set trapezoidal position profile mode
TUM1; //set Target Update Mode 1
UPD; //execute immediate
```

Example2:

```
// Start a position profile with TUM0. Position feedback:
// 500 lines incremental encoder (2000 counts/rev)
CACC = 0.3183; //acceleration rate = 1000[rad/s^2]
CSPD = 33.3333; //slew speed = 1000[rpm]
CPOS = 6000; //position command = 3[rot]
CPR; //position command is relative
SRB ACR 0xFFFF, 0x800; // and additive
MODE PP; // set trapezoidal position profile mode
// No need to set TUM0 before UPD as MODE PP does it automatically
UPD; //execute immediate
```


6.2.5.1.139. VPLANE

Only available on multi-axis Motion Controller

Syntax

VPLANE (*X_axis*, *Y_axis*, *Tangent_axis*) **Vector PLANE**

Operands *X_axis*, *Y_axis*, *Tangent_axis*: slave axes defining the coordinate system

Type	MPL Program	On-line
	X	X

Binary code

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	0	0	0	1	1	1
							<i>X_axis</i>								
							<i>Y_axis</i>								
							<i>Tangent_axis</i>								

Description Sets the 2D coordinate system for Vector Mode using the slave axes specified with *X_axis*, *Y_axis*. and *Tangent_axis*.

Example

```
// 2D linear interpolated profile. Position feedbacks: 500 lines
//incremental encoder

SETMODE 0xCF00; //Clear buffer
VPLANE (A, B, C); // X_axis = A, Y_axis = B and Tangent_axis = C
RESRATIOX=0u;
RESRATIOY=0u;
NLINESTAN=2000;
MODE VM; // Set Vector Mode

// Circular segment of radius 3.14159[mm], with initial angle 0[deg] and
angle increment 360[deg])
CIRCLE1 1L, 360.; CIRCLE2 1L, 0.;
UPD; //Execute immediate
// Insert End Segment
VSEG1 0L, 0L; VSEG2 0L, 0L;
WMC (A, B, C); // wait for motion completion
```

6.2.5.1.140. VSEG

Only available on multi-axis Motion Controller

Syntax

VSEG1 *Pos_X, Pos_Y*

Vector linear **SEG**ment

VSEG2 *Pos_X, Pos_Y*

Operands *Pos_X*: X axis position increment for 2D trajectory

Pos_Y: Y axis position increment for 2D trajectory

Type

MPL Program	On-line
X	X

Binary code

VSEG1 *Pos_X, Pos_Y*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	1	1	0	0	0	0	0	0	0	0	0
LOWORD(M)															
HIWORD(M)															
LOWORD(X_PROJ)															
HIWORD(X_PROJ)															

VSEG2 *Pos_X, Pos_Y*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0
LOWORD(Y_PROJ)															
HIWORD(Y_PROJ)															

Description VSEG1 and VSEG2 define a linear segment for 2D trajectory executed in Vector Mode.

Based on Radius, Theta_inc and Theta_start the MPL compiler from the actual parameters used by the motion controller to generate the PVT points for slave axes.

If the points are sent from a host then the following relations must be used to compute the actual parameters of a path segment:

$$M = \sqrt{Pos_X^2 + Pos_Y^2}$$

$$X_PROJ = \frac{Pos_X}{M}$$

$$Y_PROJ = \frac{Pos_Y}{M}$$

Example

```
// 2D linear interpolated profile. Position feedbacks: 500 lines
//incremental encoder

SETMODE 0xCF00; //Clear buffer
LPLANE (A, C); //Slaves A and C define the coordinate system
MODE LI; // Set Linear Interpolation Mode
// Increment position with (X, Y) = (0.5[rot], 0.05[rot])
LPOS1 1000L, 100L; LPOS2 1000L, 100L;
UPD; //Execute immediate
// Increment position with (X, Y) = (0.05[rot], 0.5[rot])
LPOS1 100L, 1000L; LPOS2 100L, 1000L;
// Increment position with (X, Y) = (0.5[rot], 0.1[rot])
LPOS1 1000L, 200L; LPOS2 1000L, 200L;
// Increment position with (X, Y) = (0.5[rot], 0.5[rot])
LPOS1 1000L, 1000L; LPOS2 1000L, 1000L;
```

6.2.5.1.141. WAIT!

Syntax

WAIT! **WAIT** motion event !

WAIT! *value32* **WAIT** motion event ! but exit if time > *value32*

Operands *value32*: 32-bit long immediate value – wait loop timeout limit

Type	MPL Program	On-line
	X	—

Binary code

WAIT!

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

WAIT! *value32*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	0	0	0	1	0	0	0
LOWORD(<i>value32</i>)															
HIWORD(<i>value32</i>)															

Description Stops the MPL program execution, until the programmed event occurs. During this period, only the MPL commands received via a communication channel are processed. You may also specify the timeout limit for the wait, by adding a time value after the **WAIT!** command: **value32**. If the monitored event doesn't occur in the time limit set, the wait loop is interrupted; the event checking is reset and the MPL program passes to the next MPL instruction. The timeout is measured in internal time units i.e. slow loop sampling periods.

Example1

```
// Unconditional wait for a motion complete event
// Position profile. Position feedback: 500 lines incremental
// encoder (2000 counts/rev)
CSPD = 10;//slew speed = 300[rpm]
CPOS = 4000;//position command = 2[rot]
CPR;//position command is relative
MODE PP;
UPD;//execute immediate
!MC; // set motion complete event
WAIT!; //wait for the programmed event to occur
// if the final position is not reached or the motion complete is
// not set because the settle band and time conditions are not met
// the MPL program will remain at this point
```

Example2

```
//Conditional wait for a limit switch event
// Speed profile. Position feedback: 500 lines incremental
// encoder (2000 counts/rev)
CACC = 0.1591;//acceleration rate = 500[rad/s^2]
CSPD = 40; //jog speed = 1200[rpm]
MODE SP;
TUM1; //set Target Update Mode 1
UPD;//execute immediate
ENLSP1; // activate LSP input to detect low->high transitions
!LSP; // set event of LSP transition
WAIT! 5000; //Wait until the event occurs but no more than 5[s]
STOP; // stop motion
```

6.2.5.1.142. WAMPU

Syntax

WAMPU (*Slave*), *value32* Wait for slave's **Absolute Motor Position Under** *value32*

WAMPU (*Slave*), *VAR32* Wait for slave's **Absolute Motor Position Under** *VAR32*

Operands *Slave*: slave axis monitored for event occurrence
 VAR32: long variable
 value32: 32-bit long immediate value

Type

MPL Program	On-line
X	X

Binary code

WAMPU (*Slave*) *value32*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	0	0	0	0	0	0	1	0
<i>Slave</i>															
LOWORD(<i>value32</i>)															
HIWORD(<i>value32</i>)															

WAMPU (*Slave*) *VAR32*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	0	0	0	1	0	0	1	0
<i>Slave</i>															
& <i>VAR32</i>															

Description Sets the event condition and halts the execution of the MPL program from motion controller until the slave's motor absolute position becomes equal or under the specified value or the value of the specified variable. After you have programmed an event, you can do the following actions:

- Change the motion mode and/or the parameters when the event occurs, with command **UPD!**
- Stop the motion of slave axes when the event occurs, with command **STOP**.

The programmed event is automatically erased when the event occurs or if the timeout for wait expires.

Execution Activates the monitoring of the event on the slave axis, when motor absolute position \leq *value32*, respectively *VAR32*. The motion controller application remains in a loop until the event on the slave axis occurs or it timeouts. This operation erases a previous programmed event that has occurred.

Example

```
//Stop slave B and C when the motor position <= -3 rev on slave A
//Position feedback: 500 lines encoder (2000 counts/rev)
// Wait for event : When axis A motor absolute position is
// equal or under value -3 rot
WAMPU (A), -6000L;
(B,C) {
    STOP; // Stop motion with acceleration / deceleration set
}
```

6.2.5.1.143. WAMPO

Syntax

WAMPO (*Slave*) *value32* Wait for slave's **Absolute Motor Position Over** *value32*

WAMPO (*Slave*) *VAR32* Wait for slave's **Absolute Motor Position Over** *VAR32*

Operands *Slave*: slave axis monitored for event occurrence
 VAR32: long variable
 value32: 32-bit long immediate value

Type	MPL Program	On-line
	X	X

Binary code

WALPO (*Slave*) *value32*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	0	1	1	0	0	0	1	0
<i>Slave</i>															
LOWORD(<i>value32</i>)															
HIWORD(<i>value32</i>)															

WALPO (*Slave*) *VAR32*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	0	1	1	1	0	0	1	0
<i>Slave</i>															
& <i>VAR32</i>															

Description Sets the event condition and halts the execution of the MPL program from motion controller until the slave's motor absolute position becomes equal or over the specified value or the value of the specified variable. After you have programmed an event, you can do the following actions:

- Change the motion mode and/or the parameters when the event occurs, with command **UPD!**
- Stop the motion of slave axes when the event occurs, with command **STOP**.

The programmed event is automatically erased when the event occurs or if the timeout for the wait expires.

Execution Activates the monitoring of the event on the slave axis, when motor absolute position \geq *value32*, respectively *VAR32*. The motion controller application remains in a loop until the event on the slave axis occurs or it timeouts. This operation erases a previous programmed event that has occurred.

Example

```
//Reverse motion on B slave when motor position >= lrev on C slave
//Position feedback: 500 lines encoder (2000 counts/rev)
(B) {
    //Speed profile on B slave
    CACC = 0.3183;//acceleration rate = 1000[rad/s^2]
    CSPD = 3.3333;//jog speed = 100[rpm]
    MODE SP;
    TUM1; //set Target Update Mode 1
    UPD; // execute immediate
}
CSPD = -40; //jog speed = -1200[rpm]
(B)CSPD = CSPD; //Send the local variable CSPD to variable CSPD of
// slaves (B)
// Wait for event : When axis C motor absolute position is equal
// or over value 1 rot
WAMPO (C), 2000L;
(B) {
    UPD; // Update immediate. Speed command is reversed
}
```

Remark: You can activate a new motion on a programmed event in 2 ways:

- Set **UPD!** command then wait for event occurrence. This will activate the new motion immediately when the event occurs
- Wait the event then update the motion with **UPD**. This will activate the new motion with a slight delay compared with the first option

6.2.5.1.144. WALPU

Syntax

WALPU (*Slave*) *value32* **Wait for slave's Absolute Load Position Under** *value32*

WALPU (*Slave*) *VAR32* **Wait for slave's Absolute Load Position Under** *VAR32*

Operands *Slave*: slave axis monitored for event occurrence
 VAR32: long variable
 value32: 32-bit long immediate value

Type

MPL Program	On-line
X	X

Binary code

WALPU (*Slave*) *value32*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	0	1	0	0	0	0	1	0
<i>Slave</i>															
LOWORD(<i>value32</i>)															
HIWORD(<i>value32</i>)															

WALPU (*Slave*) *VAR32*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	0	1	0	1	0	0	1	0
<i>Slave</i>															
& <i>VAR32</i>															

Description Sets the event condition and halts the execution of the MPL program from motion controller until the slave's load absolute position becomes equal or under the specified value or the value of the specified variable. After you have programmed an event, you can do the following actions:

- Change the motion mode and/or the parameters when the event occurs, with command **UPD!**
- Stop the motion of slave axes when the event occurs, with command **STOP**.

The programmed event is automatically erased when the event occurs or if the timeout for the wait expires.

Execution Activates the monitoring of the event on the slave axis, when load absolute position \leq *value32*, respectively *VAR32*. The motion controller application remains in a loop until the event on the slave axis occurs or it timeouts. This operation erases a previous programmed event that has occurred.

Example

```
//Set the speed command when load absolute position is <= 10 rev
//Position feedback: 500 lines encoder (2000 counts/rev)
// Wait for event : When axis B load absolute position is equal
// or under value 10 rot
CSPD = 13.3333;//new slew speed command = 500[rpm]
WALPU (B), 20000L;
(C)CSPD = CSPD; //Send the local variable CSPD to variable CSPD
           // of slaves (C)
```

6.2.5.1.145. WALPO

Syntax

WALPO (*Slave*), *value32* Wait slave's **Absolute Load Position Over** *value32*

WALPO (*Slave*), *VAR32* Wait slave's **Absolute Load Position Over** *VAR32*

Operands *Slave*: slave axis monitored for event occurrence
 VAR32: long variable
 value32: 32-bit long immediate value

Type

MPL Program	On-line
X	X

Binary code

WALPO (*Slave*) *value32*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	0	1	1	0	0	0	1	0
<i>Slave</i>															
LOWORD(<i>value32</i>)															
HIWORD(<i>value32</i>)															

WALPO (*Slave*) *VAR32*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	0	1	1	1	0	0	1	0
<i>Slave</i>															
& <i>VAR32</i>															

Description Sets the event condition and halts the execution of the MPL program from motion controller until the slave's load absolute position becomes equal or over the specified value or the value of the specified variable. After you have programmed an event, you can do the following actions:

- Change the motion mode and/or the parameters when the event occurs, with command **UPD!**
- Stop the motion of slave axes when the event occurs, with command **STOP**.

The programmed event is automatically erased when the event occurs or if the timeout for the wait expires.

Execution Activates the monitoring of the event on the slave axis, when load absolute position \geq *value32*, respectively *VAR32*. The motion controller application remains in a loop until the event on the slave axis occurs or it timeouts. This operation erases a previous programmed event that has occurred.

Example

```
//Stop all slaves when load position on slave B >= 3 rev
//Position feedback: 500 lines encoder (2000 counts/rev)
    TIMEOUT 1000L; // Set wait timeout to 1[s]
// Wait for event : When axis B motor absolute position is equal
// or over value 1 rot
    WAMPO (B), 6000L;
    STOP; // Stop the motion
```

6.2.5.1.146. SAVEERROR

Syntax

SAVEERROR VAR32 **GET** oldest **ERROR** from RAM

Operands VAR32: 32-bit long variable containing the slave error

Type

MPL Program	On-line
X	X

Binary code

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	0	0	0	0	0	1	0	0	0	1
&VAR32															

Description Saves the slave error from VAR32 in a circular buffer located in EEPROM. The buffer can hold up to 8 error codes. If the buffer is full and a new error is saved then the oldest error is overwritten. The content of VAR32 must be initialized using the GETERROR VAR32 command.

Example

```
// Retrieve oldest 3 errors and save them in the EEPROM
LONG error_code; //define variable error_code
GETERROR error_code; //Read oldest error from motion controller RAM
SAVEERROR error_code; // Save the error in the motion controller EEPROM
GETERROR error_code; //Read second error from motion controller RAM
SAVEERROR error_code; // Save the error in the motion controller EEPROM
GETERROR error_code; //Read third error from motion controller RAM
SAVEERROR error_code; // Save the error in the EEPROM
SEND error_code; // Send third error code to the host
```

6.2.5.1.147. GETERROR

Syntax

GETERROR VAR32 **GET** oldest **ERROR** from RAM
GETERROR n,VAR32 **GET** *n*-th **ERROR** from EEPROM

Operands *VAR32*: 32-bit long variable to store the error
n : error position in the circular buffer

Type	MPL Program	On-line
	X	X

Binary code

GETERROR VAR32

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	0	0	0	0	0	1	0	0	1	1
<i>&VAR32</i>															

GETERROR n, VAR32

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	0	0	0	0	0	1	0	0	0	0
<i>n</i>															
<i>&VAR32</i>															

Description The motion controller uses a circular buffer in RAM to store the slaves' errors. The buffer can hold up to 8 error codes. If an error is received and the buffer is full then the new error will overwrite the oldest one. The buffer is read with GETERROR VAR32 command which retrieves the oldest error from the motion controller RAM. The error code is saved in VAR32. Once it was read the buffer entry is released. GETERROR VAR32 returns zero when the buffer is empty.

The GETERROR *n, VAR32* retrieves *n*-th error stored in the non-volatile memory of the drive. The errors are stored in a circular buffer that can hold up to 8 error codes, *n* = 0 oldest entry and *n* = 7 newest entry. The errors can be saved in the EEPROM with the command SAVEERROR command.

Example

```
// Retrieve oldest 3 errors and save them in the EEPROM
LONG error_code; //define variable error_code
GETERROR error_code; //Read oldest error from motion controller RAM
SAVEERROR error_code; // Save the error in the motion controller EEPROM
GETERROR error_code; //Read second error from motion controller RAM
SAVEERROR error_code; // Save the error in the motion controller EEPROM
GETERROR error_code; //Read third error from motion controller RAM
SAVEERROR error_code; // Save the error in the EEPROM
GETERROR 1, error_code; // Retrieve second error from the EEPROM
SEND error_code; // Send third error code to the host
```

6.2.5.1.148. WVDU

Syntax

WVDU *value32* Wait Vector Distance Under *value32*

WVDU *VAR32* Wait Vector Distance Under *VAR32*

Operands *VAR32*: long variable
 value32: 32-bit long immediate value

Type	MPL Program	On-line
	X	X

Binary code

WVDU <i>value32</i>															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	0	0	0	0	0	1	0	1
LOWORD(<i>value32</i>)															
HIWORD(<i>value32</i>)															

WVDU <i>VAR32</i>															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	0	0	0	1	0	1	0	1
& <i>VAR32</i>															

Description Sets the event condition and halts the execution of the MPL program from motion controller until the vector distance is equal or under the specified value or the value of 32-bit variable. After you have programmed an event, you can do the following actions:

- Change the motion mode and/or the parameters when the event occurs, with command **UPD!**
- Stop the motion when the event occurs, with command **STOP**.

The programmed event is automatically erased when the event occurs or if the timeout for wait expires.

Execution Activates the monitoring of the event when vector distance \leq *value32*, respectively *VAR32*. The motion controller application remains in a loop until the event occurs or it timeouts. This operation erases a previous programmed event that has occurred.

6.2.5.1.149. WCAP

Syntax

WCAP1 (*Slave*) Wait for slave's 1st **CAP**ture input transition 0 to 1

WCAP0 (*Slave*) Wait for slave's 1st **CAP**ture input transition 1 to 0

Operands *Slave*: slave axis monitored for event occurrence

Type	MPL Program	On-line
	X	X

Binary code

WCAP1 (*Slave*)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	0	0	1	1	0	1	1	1
<i>Slave</i>															

WCAP0 (*Slave*)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	0	0	1	0	0	1	1	1
<i>Slave</i>															

Description Sets the event condition and halts the execution of the MPL program from motion controller until the transition occurs on the 1st capture/encoder index inputs on the slave axis. When the programmed transition occurs the following happens on the slave axis:

- The input capability to detect transitions is disabled. It must be enabled again to detect another transition
- Motor position **APOS_MT** is captured and memorized in the MPL variable **CAPPOS**, except the case of open-loop systems, where reference position **TPOS** is captured instead
- Master position **APOS2** or load position **APOS_LD** is captured and memorized in the MPL variable **CAPPOS2**, except the case of steppers controlled open loop with an encoder on the load, when load position is captured in **CAPPOS**.

The selection between master and load position is done as follows: load position is saved in **CAPPOS2** only for setup configurations which use different sensors for load and motor and foresee a transmission ratio between them. For all the other setup configurations, the master position is saved in **CAPPOS2**

After you have programmed an event, you can do the following actions:

- Change the motion mode and/or the parameters when the event occurs, with command **UPD!**
- Stop the motion when the event occurs, with command **STOP**.

The programmed event is automatically erased when the event occurs or if the timeout for the wait expires.

Execution Activates the monitoring of the event, when the programmed transition (low to high or high to low) occurs on the 1st capture/encoder index input. The motion controller application remains in a loop until the event on the slave axis occurs or it timeouts. This operation erases a previous programmed event that has occurred.

Example

```
//Stop motion on all slaves on next encoder index
// Wait for event : When axis A encoder index goes low->high
WCAP1 (A);
STOP; //Stop the motion
(A) { // Command slave A to move on captured position
CPOS = CAPPOS; // new command position = captured position
CPA; //position command is absolute
MODE PP;
TUM1; //set Target Update Mode 1
UPD; //execute immediate
}
WMC (A); //wait for completion
```

6.2.5.1.150. WVDO

Syntax

WVDO *value32* Wait Vector Distance Under *value32*

WVDO *VAR32* Wait Vector Distance Under *VAR32*

Operands *VAR32*: long variable
 value32: 32-bit long immediate value

Type	MPL Program	On-line
	X	X

Binary code

WVDO *value32*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	0	0	1	0	0	1	0	1
LOWORD(<i>value32</i>)															
HIWORD(<i>value32</i>)															

WVDO *VAR32*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	0	0	1	1	0	1	0	1
& <i>VAR32</i>															

Description Sets the event condition and halts the execution of the MPL program from motion controller until the vector distance is equal or over the specified value or the value of 32-bit variable. After you have programmed an event, you can do the following actions:

- Change the motion mode and/or the parameters when the event occurs, with command **UPD!**
- Stop the motion when the event occurs, with command **STOP**.

The programmed event is automatically erased when the event occurs or if the timeout for wait expires.

Execution Activates the monitoring of the event when vector distance \geq *value32*, respectively *VAR32*. The motion controller application remains in a loop until the event occurs or it timeouts. This operation erases a previous programmed event that has occurred.

6.2.5.1.151. WTR

Syntax

WTR (*Slave*) **Wait Target Reached**

Operands *Slave*: slave axis monitored for event occurrence

Type	MPL Program	On-line
	X	X

Binary code

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	1
<i>Slave</i>															

Description Sets the event condition and halts the execution of the MPL program from motion controller until the slave axis reaches the target position. After you have programmed an event, you can do the following actions:

- Change the motion mode and/or the parameters when the event occurs, with command **UPD!**
- Stop the motion when the event occurs, with command **STOP**.

The programmed event is automatically erased when the event occurs or if the timeout for wait expires.

Execution Activates the monitoring of the event when target reached. The motion controller application remains in a loop until the event occurs or it timeouts. This operation erases a previous programmed event that has occurred.

6.2.5.1.152. WPRU

Syntax

WPRU (*Slave*), *value32* **Wait** for slave's **Position Reference Under** *value32*

WPRU (*Slave*), *VAR32* **Wait** for slave's **Position Reference Under** *VAR32*

Operands *Slave*: slave axis monitored for event occurrence

VAR32: long variable

value32: 32-bit long immediate value

Type

MPL Program	On-line
X	X

Binary code

WPRU (*Slave*), *value32*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0
<i>Slave</i>															
LOWORD(<i>value32</i>)															
HIWORD(<i>value32</i>)															

WPRU (*Slave*), *VAR32*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	0	0	0	1	0	0	0	0
<i>Slave</i>															
& <i>VAR32</i>															

Description Sets the event condition and halts the execution of the MPL program from motion controller until the when the position reference is equal or under the specified value or the value of the specified variable. After you have programmed an event, you can do the following actions:

- Change the motion mode and/or the parameters when the event occurs, with command **UPD!**
- Stop the motion when the event occurs, with command **STOP**.

The programmed event is automatically erased when the event occurs or if the timeout for the **WAIT!** command expires.

Remark: After setting **UPD!** or **STOP!** you need to wait until the programmed event occurs using **WAIT!**, otherwise, the program will continue with the next instructions that may override the event monitoring.

Execution Activates the monitoring of the event, when position reference \leq *value32*, respectively *VAR32*. The motion controller application remains in a loop until the event on the slave axis occurs or it timeouts. This operation erases a previous programmed event that has occurred.

Example:

```
//Stop motion when position reference >= 3 rev  
//Position feedback: 500 lines encoder (2000 counts/rev)  
WPRU 6000; //Set event: when position reference is >= 3 rev  
STOP;//Stop the motion when the event occurs
```

6.2.5.1.153. WPRO

Syntax

WPRO (*Slave*), *value32* **Wait** for slave's **Position Reference Over** *value32*

WPRO (*Slave*), *VAR32* **Wait** for slave's **Position Reference Over** *VAR32*

Operands *Slave*: slave axis monitored for event occurrence
 VAR32: long variable
 value32: 32-bit long immediate value

Type

MPL Program	On-line
X	X

Binary code

WPRO (*Slave*), *value32*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	0	0	2	0	0	0	0	0
<i>Slave</i>															
LOWORD(<i>value32</i>)															
HIWORD(<i>value32</i>)															

WPRO (*Slave*), *VAR32*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	0	0	1	1	0	0	0	0
<i>Slave</i>															
& <i>VAR32</i>															

Description

Sets the event condition and halts the execution of the MPL program from motion controller until the slave's position reference is equal or over the specified value or the value of the specified variable. After you have programmed an event, you can do the following actions:

- Change the motion mode and/or the parameters when the event occurs, with command **UPD!**
- Stop the motion when the event occurs, with command **STOP**.

The programmed event is automatically erased when the event occurs or if the timeout for wait expires.

Execution

Activates the monitoring of the event, when position reference \geq *value32*, respectively *VAR32*. The motion controller application remains in a loop until the event on the slave axis occurs or it timeouts. This operation erases a previous programmed event that has occurred.

Example:

```
//Stop motion on all slaves when position reference on slave C >=  
// 3 rev. Position feedback: 500 lines encoder (2000 counts/rev)  
// Wait for event : When position reference on axis C is equal or  
// over value 3rot  
WPRO (C), 6000L;  
STOP; //Stop the motion
```


6.2.5.1.154. WMSU

Syntax

WMSU (*Slave*), *value32* **Wait for slave's Motor Speed**
Under *value32*

WMSU (*Slave*), *VAR32* **Wait for slave's Motor Speed**
Under *VAR32*

Operands *Slave*: slave axis monitored for event occurrence

VAR32: fixed variable

value32: 32-bit fixed immediate value

Type

MPL Program	On-line
X	X

Binary code

!MSU *value32*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	0	0	0	1	0	0	0	0	1	1	1
0	0	0	0	0	0	1	0	0	0	1	0	1	1	0	0
LOWORD(<i>value32</i>)															
HIWORD(<i>value32</i>)															

!MSU *VAR32*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	0	0	1	1	0	0	0	0	1	1	1
0	0	0	0	0	0	1	0	0	0	1	0	1	1	0	0
& <i>VAR32</i>															

Description Sets the event condition halts the execution of the MPL program from motion controller until the motor speed is equal or under the 32-bit value or the value of the specified fixed variable. After you have programmed an event, you can do the following actions:

- Change the motion mode and/or the parameters when the event occurs, with command **UPD!**
- Stop the motion when the event occurs, with command **STOP**.

The programmed event is automatically erased when the event occurs or if the timeout for wait expires.

Execution Activates the monitoring of the event when motor speed \leq *value32*, respectively *VAR32*. The motion controller application remains in a loop until the event on the slave axis occurs or it timeouts. This operation erases a previous programmed event that has occurred.

Example

```
//Motor on slave A is decelerating. Start a new position profile
// on slave A when motor speed < 600 rpm
//Position feedback: 500 lines encoder (2000 counts/rev)
WMSU (A) 20; //Set event: when motor speed is < 600 rpm
// prepare new motion mode
(A) {

    CACC = 0.3183;//acceleration rate = 1000[rad/s^2]
    CSPD = 100;//slew speed = 3000[rpm]
    CPOS = 20000;//position command = 10[rot]
    CPR; //position command is relative
    MODE PP;
    TUM1; //set Target Update Mode 1
    UPD
};
```

6.2.5.1.155. WMSO

Syntax

WMSO (*Slave*), *value32* Wait for slave's **M**otor **S**peed **O**ver *value32*

WMSO (*Slave*), *VAR32* Wait for slave's **M**otor **S**peed **O**ver *VAR32*

Operands *Slave*: slave axis monitored for event occurrence
 VAR32: fixed variable
 value32: 32-bit fixed immediate value

Type

MPL Program	On-line
X	X

Binary code

WMSO (*Slave*), *value32*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	0	0	1	0	0	1	0	0
<i>Slave</i>															
LOWORD(<i>value32</i>)															
HIWORD(<i>value32</i>)															

WMSO (*Slave*), *VAR32*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	0	0	1	1	0	1	0	0
<i>Slave</i>															
& <i>VAR32</i>															

Description Sets the event condition and halts the execution of the MPL program from motion controller until when the motor speed is equal or over the 32-bit value or the value of the specified fixed variable. After you have programmed an event, you can do the following actions:

- Change the motion mode and/or the parameters when the event occurs, with command **UPD!**
- Stop the motion when the event occurs, with command **STOP**.

The programmed event is automatically erased when the event occurs or if the timeout for the wait expires.

Execution Activates the monitoring of the event on the slave axis, when motor speed \geq *value32*, respectively *VAR32*. The motion controller application remains in a loop until the event on the slave axis occurs or it timeouts. This operation erases a previous programmed event that has occurred.

Example

```
//Motor is accelerating. Stop motion on all axes when motor
//speed > 600 rpm
//Position feedback: 500 lines encoder (2000 counts/rev)
WMSO (D) 20; //Set event: when motor speed is > 600 rpm
STOP;//Stop the motion when the event occurs
```

6.2.5.1.156. WLSP

Syntax

WLSP1 Wait for slave's Limit Switch Positive goes from 0 to 1

WLSP0 Wait for slave's Limit Switch Positive goes from 1 to 0

Operands –

Type	MPL Program	On-line
	X	X

Binary code

WLSP1 (Slave)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	0	1	1	1	0	1	1	1
<i>Slave</i>															

WLSP0 (Slave)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	0	1	1	0	0	1	1	1
<i>Slave</i>															

Description Sets the event condition when the programmed transition occurs at the positive limit switch input. After you have programmed an event, you can do the following actions:

- Change the motion mode and/or the parameters when the event occurs, with command **UPD!**
- Stop the motion when the event occurs, with command **STOP**.

The programmed event is automatically erased when the event occurs or if the timeout for the wait expires.

Execution Activates monitoring of the event when the programmed transition occurs at the positive limit switch input. This operation erases a previous programmed event that has occurred.

Example

```
//Reverse slave C when positive limit switch is reached
//Position feedback: 500 lines encoder (2000 counts/rev)
(C) {
    CACC = 0.0637;    //acceleration rate = 200[rad/s^2]
    CSPD = 16.6667;  //jog speed = 500[rpm]
    MODE SP;
    UPD;              //execute immediate
}
// Wait for event : When axis C positive limit switch goes low->high
WLSPl (C);
// Wait for event : When motion is completed on axis (C)
WMC (C); //limit switch is active -> quick stop mode active
        // wait until the motor stops because only then the
        // new motion commands are accepted
(C) {
    CSPD = -40;      //jog speed = -1200[rpm]
    MODE SP;         //after quick stop set again the motion mode
    UPD;             //execute immediate
}
```

6.2.5.1.157. WLSO

Syntax

WLSO (*Slave*), *value32* Wait for slave's Load Speed Over *value32*

WLSO (*Slave*,) *VAR32* Wait for slave's Load Speed Over *VAR32*

Operands *Slave*: slave axis monitored for event occurrence
 VAR32: fixed variable
 value32: 32-bit fixed immediate value

Type

MPL Program	On-line
X	X

Binary code

WLSO (*Slave*), *value32*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	0	1	1	0	0	1	0	0
<i>Slave</i>															
LOWORD(<i>value32</i>)															
HIWORD(<i>value32</i>)															

WLSO (*Slave*), *VAR32*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	0	1	1	1	0	1	0	0
<i>Slave</i>															
& <i>VAR32</i>															

Description Sets the event condition and halts the execution of the MPL program from motion controller until the load speed is equal or over the 32-bit value or the value of the specified fixed variable. After you have programmed an event, you can do the following actions:

- Change the motion mode and/or the parameters when the event occurs, with command **UPD!**
- Stop the motion when the event occurs, with command **STOP**.

The programmed event is automatically erased when the event occurs or if the timeout for wait expires.

Execution Activates the monitoring of the event when load speed \geq *value32*, respectively *VAR32*. The motion controller application remains in a loop until the event on the slave axis occurs or it timeouts. This operation erases a previous programmed event that has occurred.

Example

```
//Stop motion when load speed > 600 rpm  
//Load Position feedback: 500 lines encoder (2000 counts/rev)  
WLSO (A) 20; //Set event: when load speed is > 600 rpm  
STOP;//Stop motion on all axes
```


6.2.5.1.158. WLSN

Syntax

WLSN1 Wait for slave's Limit Switch Negative goes from 0 to 1
WLSN0 Wait for slave's Limit Switch Negative goes from 1 to 0

Operands –

Type	MPL Program	On-line
	X	X

Binary code

WLSN1 (Slave)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	0	0	1	1	0	1	1	1
<i>Slave</i>															

WLSN0 (Slave)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	0	0	1	0	0	1	1	1
<i>Slave</i>															

Description Sets the event condition and halts the execution of the MPL program from motion controller until the programmed transition occurs at the negative limit switch input. After you have programmed an event, you can do the following actions:

- Change the motion mode and/or the parameters when the event occurs, with command **UPD!**
- Stop the motion when the event occurs, with command **STOP**.

The programmed event is automatically erased when the event occurs or if the timeout for the wait expires.

Execution Activates monitoring of the event on the slave axis, when the programmed transition occurs at the negative limit switch input. The motion controller application remains in a loop until the event on the slave axis occurs or it timeouts. This operation erases a previous programmed event that has occurred.

Example

```
//Reverse slave C when negative limit switch is reached
//Position feedback: 500 lines encoder (2000 counts/rev)
(C) {
    CACC = 0.0637;    //acceleration rate = 200[rad/s^2]
    CSPD = -16.6667; //jog speed = -500[rpm]
    MODE SP;
    UPD;             //execute immediate
}
// Wait for event : When axis C negative limit switch goes low->high
WLSN1 (C);
CSPD = 40;          //jog speed = 1200[rpm]
MODE SP;           //after quick stop set again the motion mode
UPD;               //execute immediate
```

6.2.5.1.159. WLSU

Syntax

WLSU (*Slave*), *value32* Wait for slave's Load Speed Under *value32*

WLSU (*Slave*), *VAR32* Wait for slave's Load Speed Under *VAR32*

Operands *Slave*: slave axis monitored for event occurrence

VAR32: fixed variable

value32: 32-bit fixed immediate value

Type

MPL Program	On-line
X	X

Binary code

WLSU (*Slave*), *value32*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	0	1	0	0	0	1	0	0
<i>Slave</i>															
LOWORD(<i>value32</i>)															
HIWORD(<i>value32</i>)															

WLSU (*Slave*), *VAR32*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	0	1	0	1	0	1	0	0
<i>Slave</i>															
& <i>VAR32</i>															

Description

Sets the event condition and halts the execution of the MPL program from motion controller until the slave's load speed is equal or under the 32-bit value or the value of the specified fixed variable. After you have programmed an event, you can do the following actions:

- Change the motion mode and/or the parameters when the event occurs, with command **UPD!**
- Stop the motion when the event occurs, with command **STOP**.

The programmed event is automatically erased when the event occurs or if the timeout for wait expires.

Execution

Activates the monitoring of the event when load speed \leq *value32*, respectively *VAR32*. The motion controller application remains in a loop until the event on the slave axis occurs or it timeouts. This operation erases a previous programmed event that has occurred.

Example

```
// Start a position profile when load speed < 600 rpm
// Load Position feedback: 500 lines encoder (2000 counts/rev)
WLSU (A) 20; //Set event: when motor speed is < 600 rpm
// prepare new motion mode
(A) {
    CACC = 0.3183;//acceleration rate = 1000[rad/s^2]
    CSPD = 100;//slew speed = 3000[rpm]
    CPOS = 20000;//position command = 10[rot]
    CPR; //position command is relative
    MODE PP;
    TUM1; //set Target Update Mode 1
    UPD;
}
```

6.2.5.1.160. WIN

Syntax

WIN#n (Slave), 0 **Wait for slave's Input#n is 0**

WIN#n (Slave), 1 **Wait for slave's Input#n is 1**

Operands *Slave*: slave axis monitored for event occurrence
n: input line number (0<=*n*<=39)

Type	MPL Program	On-line
	X	X

Binary code

WIN#n (Slave), 0															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	0	0	0	0	0	1	1	1
<i>PxDATDIR</i>															
<i>Bit_mask</i>															

WIN#n (Slave), 1															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	0	0	0	1	0	1	1	1
<i>PxDATDIR</i>															
<i>Bit_mask</i>															

Description Sets the event condition and halts the execution of the MPL program from motion controller until the slave's digital input #n becomes 0, respectively 1. The slave checks the condition of the input #n is tested at each slow loop sampling period. After you have programmed an event, you can do the following actions:

- Change the motion mode and/or the parameters when the event occurs, with command **UPD!**
- Stop the motion when the event occurs, with command **STOP**.

The programmed event is automatically erased when the event occurs or if the timeout for the wait expires.

Execution Activates monitoring of the event on the slave axis, when the digital input #n becomes 0 (!IN#n 0), respectively 1 (!IN#n 1). The motion controller application remains in a loop until the event on the slave axis occurs or it timeouts. This operation erases a previous programmed event that has occurred.

Example

```
// Start motion on slave A when digital input #36 from slave C is high
// Wait for event: When axis C digital input 36/IN36 is high
WIN#36 (C), 1;
(A) {
    //Position profile
    CACC = 0.3183;//acceleration rate = 1000[rad/s^2]
    CSPD = 40.;//slew speed = 1200[rpm]
    CPOS = 12000L;//position command = 6[rot]
    CPR; //position command is relative
    MODE PP;
    TUM1; //set Target Update Mode 1
    UPD; // execute immediate
}
WMC (A); // wait for motion completion
```

6.2.5.1.161. W2CAP

Syntax

W2CAP1 (*Slave*) Wait for slave's 2nd **CAP**ture input transition 0 to 1

W2CAP0 (*Slave*) Wait for slave's 2nd **CAP**ture input transition 1 to 0

Operands *Slave*: slave axis monitored for event occurrence

Type	MPL Program	On-line
	X	X

Binary code

W2CAP1 (*Slave*)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	0	1	0	1	0	1	1	1
<i>Slave</i>															

W2CAP0 (*Slave*)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	0	1	0	0	0	1	1	1
<i>Slave</i>															

Description Sets the event condition and halts the execution of the MPL program from motion controller until the transition occurs on the 2nd capture/encoder index inputs on the slave axis. When the programmed transition occurs the following happens on the slave axis:

- The input capability to detect transitions is disabled. It must be enabled again to detect another transition
- Motor position **APOS_MT** is captured and memorized in the MPL variable **CAPPOS**, except the case of open-loop systems, where reference position **TPOS** is captured instead
- Master position **APOS2** or load position **APOS_LD** is captured and memorized in the MPL variable **CAPPOS2**, except the case of steppers controlled open loop with an encoder on the load, when load position is captured in **CAPPOS**.

The selection between master and load position is done as follows: load position is saved in **CAPPOS2** only for setup configurations which use different sensors for load and motor and foresee a transmission ratio between them. For all the other setup configurations, the master position is saved in **CAPPOS2**

After you have programmed an event, you can do the following actions:

- Change the motion mode and/or the parameters when the event occurs, with command **UPD!**
- Stop the motion when the event occurs, with command **STOP**.

The programmed event is automatically erased when the event occurs or if the timeout for the wait expires.

Execution Activates the monitoring of the event, when the programmed transition (low to high or high to low) occurs on the selected capture input. This operation erases a previous programmed event that has occurred.

Example

```
//Stop motion on all slaves on next 2nd encoder index
// Wait for event : When axis A 2nd encoder index / home input
//goes low->high
W2CAP1 (A);
STOP; // Stop the motion
(A) { // Command slave A to move on captured position
CPOS = CAPPOS; // new command position = captured position
CPA; //position command is absolute
MODE PP;
TUM1; //set Target Update Mode 1
UPD; //execute immediate
}
WMC (A); //wait for completion
```

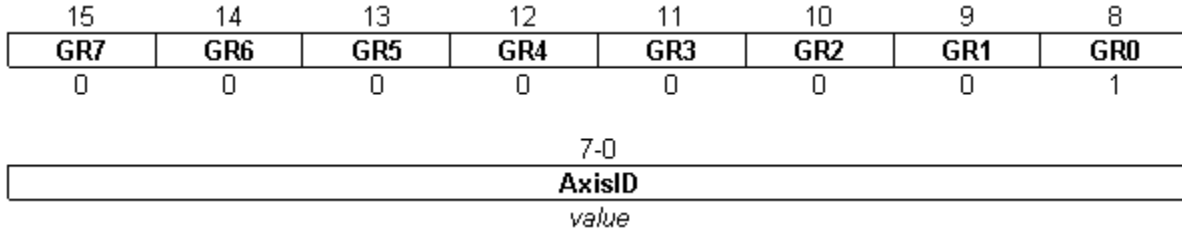
6.2.6. MPL Registers

6.2.6.1. AAR - Axis Addresses Register (status, RO)

Purpose: **AAR** is a 16-bit status register, containing information that defines the individual and group addresses of the motion axis.

MPL Address: 0x030C

Contents. **AAR** information is structured as follows:



Bits 15-8 GRn. Group n selection

- 0 = The motion axis does not belong to group n
- 1 = The motion axis belongs to group n

Bits 7-0 AXISID. Axis address

value = Individual identification address for the motion axis

Remark: The **AxisID** is initially set at power on using the following algorithm:

- 1 With the value read from the EEPROM setup table containing all the setup data.
- 2 If the setup table is invalid, with the last axis ID value read from a valid setup table
- 3 If there is no axis ID set by a valid setup table, with the value read from the hardware switches/jumpers for axis ID setting
- 4 If the drive/motor has no hardware switches/jumpers for axis ID setting, with the default axis ID value which is 255.

6.2.6.2. ACR - Auxiliary Command Register (status, R/W)

Purpose: ACR is a 16-bit status register. It defines extra settings like: the configuration for automatic start and the external reference, operation options for the S-curve and the electronic camming modes.

MPL Address: 0x0912

Contents. ACR information is structured as follows:

15-14		13	12	11	10	9	8
Reserved		SOLCTR	CAMTYPE	RPOSTYPE	POSCTR	SPDCTR	TCTR
00		0	0	0	1	0	1
7	6	5	4	3	2	1	0
DIGREF	AREF	RDAREF	Reserved	AXISEN	DIGTYPE	ASTART	STPSC
0	0	1	0	0	1	0	1

Bits 15-14 Reserved

Bit 13 SOLCTR. Control type for stepper open loop

- 0 = Position control with automatic external reference analogue
- 1 = Speed control with automatic external reference analogue

Bit 12 CAMTYPE. Electronic camming type

- 0 = Relative
- 1 = Absolute

Bit 11 RPOSTYPE. Relative positioning type

- 0 = Standard
- 1 = Additive

Bit 10 POSCTR. Position control

- 0 = Disable
- 1 = Enable

Bit 9 SPDCTR. Speed control

- 0 = Disable
- 1 = Enable

Bit 8 TCTR. Torque control

- 0 = Disable
- 1 = Enable

Bit 7 DIGREF. Digital external reference

- 0 = Disable

1 = Enable

Bit 6 AREF. Analogue external reference

0 = Disable

1 = Enable

Bits 5 RDAREF. Read analogue external reference for torque mode when “Automatically activated after Power On” is enabled

0 = In slow loop

1 = In fast loop

Bit 4 Reserved

Bit 3 AXISEN. Behavior at ENABLE input transitions from low to high

0 = Don't execute AXISON

1 = Execute AXISON

Bit 2 DIGTYPE. Digital external reference type

0 = 2nd encoder

1 = Pulse & Direction

Bit 1 ASTART. Start automatically after power on

0 = Disable

1 = Enable

Bit 0 STPSC. Stop profile for S-curve motion mode

0 = An S-curve profile

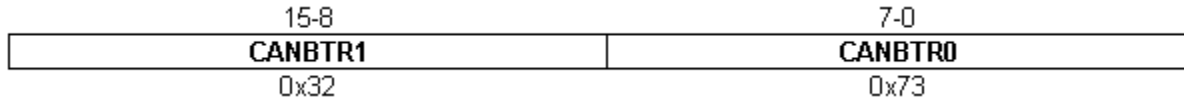
1 = A trapezoidal profile

6.2.6.3. CBR - CAN baud rate Register (status, R/W)

CBR is a 16-bit status register, containing information to setup the communication baud rate parameters for CAN controller.

MPL Address: 0x030D

Contents. **CBR** information is structured as follows:



Bit 15-8 CANBTR1. CAN bus Timing Register 1 (BTR1)

xx = CAN controller bus timing register 1

Bit 7-0 CANBTR0. CAN bus Timing Register 0 (BTR0)

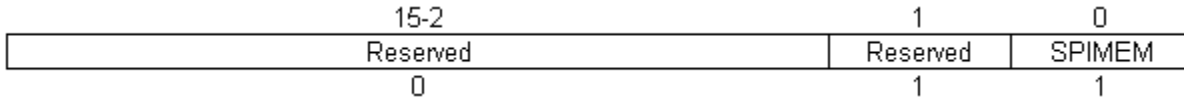
xx = CAN controller bus timing register 0

6.2.6.4. CCR - Communication Control Register (command, R/W)

CCR is a 16-bit status register, containing settings for the SPI link with the EEPROM memory.

MPL Address: 0x030A

Contents. **CCR** information is structured as follows:



Bit 15-2 Reserved

Bit 1 Reserved

Bit 0 SPIMEM. EEPROM memory

0 = Not installed

1 = Installed

6.2.6.5. CER - Communication Error Register (status, RO)

CER is a 16-bit status register, containing status information about communication errors on CAN, SPI and SCI communication channels.

MPL Address: 0x0301

Contents. **CER** information is structured as follows:

15-8				11	10	9	8
Reserved				OFWRER	OFRDER	ALRSER	ALRDER
0				0	0	0	0
7	6	5	4	3	2	1	0
SPITTO	CANBER	CANTER	CANRER	Reserved	SCIRTO	SCITTO	SCIRER
0	0	0	0	0	0	0	0

Bit 15-12 Reserved

Bit 11 [OFWRER](#). EnDat encoder offset write error

- 0 = No SPI timeout
- 1 = SPI timeout

Bit 10 [OFRDER](#). EnDat encoder offset read error

- 0 = No SPI timeout
- 1 = SPI timeout

Bit 9 [ALRSER](#). EnDat encoder alarm reset error

- 0 = No SPI timeout
- 1 = SPI timeout

Bit 8 [ALRDER](#). EnDat encoder alarm read error

- 0 = No SPI timeout
- 1 = SPI timeout

Bit 7 [SPITTO](#). SPI timeout on write operation

- 0 = No SPI timeout
- 1 = SPI timeout

Bit 6 [CANBER](#). CAN bus off error

- 0 = No CAN bus off error
- 1 = Error

Bit 5 [CANTER](#). CAN Tx overrun error

- 0 = No CAN transmission overrun error

1 = CAN transmission overrun error

Bit 4 CANRER. CAN Rx overrun error

0 = No CAN reception overrun error

1 = CAN reception overrun error

Bit 3 Reserved

Bit 2 SCIRTO. SCI Rx timeout error

0 = No SCI reception timeout error

1 = SCI reception timeout error

Bit 1 SCITTO. SCI Tx timeout error

0 = No SCI transmission timeout error

1 = SCI transmission timeout error

Bit 0 SCIRER. SCI Rx error

0 = No SCI reception error

1 = SCI reception error

6.2.6.6. CSR - Communication Status Register (status, RO)

CSR is a 16-bit status register, containing status information about the communication channels of the system.

MPL Address: 0x030B

Contents. CSR information is structured as follows:

15	14	13-11	10	9-8	7-1	0
ELGEAR	AXISDP	SCIBD	Reserved	SPIBD	Reserved	SCITYPE
0	0	100	0	10	0	0

Bit 15 ELGEAR. Electronic gearing/camming master flag

- 0 = No data to send
- 1 = Data to send

Bit 14 AXISDSTP. Axis ID setup flag

- 0 = Initial Axis ID set by software
- 1 = Initial Axis ID set by hardware

Bit 13-11 SCIBD. SCI baud rate

- 000 = 9600
- 001 = 19200
- 010 = 38400
- 011 = 56600
- 100 = 115200
- 101 = Reserved
- 110 = Reserved
- 111 = Reserved

Bit 10 Reserved

Bit 9-8 SPIBD. SPI baud rate

00 = 1 MHz

01 = 2 MHz

10 = 5 MHz

11 = Reserved

Bit 7-1 Reserved

Bit 0 SCITYPE. Serial communication driver type

0 = RS-232

1 = RS485

6.2.6.7. ICR - Interrupts Control Register (command, R/W)

ICR is a 16-bit command register, enabling/disabling the MPL interrupts. All the unmasked bits of this register will allow the generation of a MPL interrupt at the occurrence of the associated specific situation.

MPL Address: 0x0304

Contents. ICR information is structured as follows:

15	14-12			11	10	9	8
GIM				EVNIM	TPIM	MOTIM	PCAPIM
1	000			0	0	0	0
7	6	5	4	3	2	1	0
LSWNIM	LSWPIM	WRPIM	CMERIM	CTRERIM	SWPRIM	PDPIM	DLSBIM
0	0	0	0	1	1	1	1

Bit 15 GIM. Globally enable/disable MPL interrupts

0 = Disable

1 = Enable

Bit 14-12 Reserved

Bit 11 EVNIM. Enable/disable interrupt 11 – “Event set has occurred”

0 = Disable

1 = Enable

Bit 10 TPIM. Enable/disable interrupt 10 – “Time period has elapsed”

0 = Disable

1 = Enable

Bit 9 MOTIM. Enable/disable interrupt 9 – “Motion is complete”

0 = Disable

1 = Enable

Bit 8 PCAPIM. Enable/disable interrupt 8 – “Capture input transition detected”

0 = Disable

1 = Enable

Bit 7 LSWNIM. Enable/disable interrupt 7 – “LSN programmed transition detected”

0 = Disable

1 = Enable

Bit 6 LSWPIM. Enable/disable interrupt 6 – “LSP programmed transition detected”

0 = Disable

1 = Enable

Bit 5 WRPIM. Enable/disable interrupt 5 – “Position wrap around”

0 = Disable

1 = Enable

Bit 4 CMERIM. Enable/disable interrupt 4 – “Communication error”

0 = Disable

1 = Enable

Bit 3 CTRERIM. Enable/disable interrupt 3 – “Control error”

0 = Disable

1 = Enable

Bit 2 SWPRIM. Enable/disable interrupt 2 – “Software protection”

0 = Disable

1 = Enable

Bit 1 PDPIM. Enable/disable interrupt 1 – “Short-circuit”

0 = Disable

1 = Enable

Bit 0 DLSBIM. Enable/disable interrupt 0 – “Enable input has changed”

0 = Disable

1 = Enable

6.2.6.8. ISR - Interrupts Status Register (status, RO)

ISR is a 16-bit status register, containing the interrupt flags for MPL interrupts. Only unmasked MPL interrupts (see Interrupt Control Register - ICR) will generate a MPL interrupt request.

MPL Address: 0x0306

Contents. **ISR** information is structured as follows:

15-12 Reserved 0000				11 EVNIF 0	10 TPIF 0	9 MOTIF 0	8 PCAPIF 0
7 LSWNIF 0	6 LSWPIF 0	5 WRPIF 0	4 CMERIF 0	3 CTRERIF 0	2 SWPRIF 0	1 PDPIF 0	0 DSLBIIF 0

Bit 15-12 Reserved

Bit 11 EVNIF. Flag for interrupt 11 – “Event set has occurred”

0 = Not triggered

1 = Triggered

Bit 10 TPIF. Flag for interrupt 10 – “Time period has elapsed”

0 = Not triggered

1 = Triggered

Bit 9 MOTIF. Flag for interrupt 9 – “Motion is complete”

0 = Not triggered

1 = Triggered

Bit 8 PCAPIF. Flag for interrupt 8 – “Capture input transition detected”

0 = Not triggered

1 = Triggered

Bit 7 LSWNIF. Flag for interrupt 7 – “LSN programmed transition detected”

0 = Not triggered

1 = Triggered

Bit 6 LSWPIF. Flag for interrupt 6 – “LSP programmed transition detected”

0 = Not triggered

1 = Triggered

Bit 5 WRPIF. Flag for interrupt 5 – “Position wraparound”

0 = Not triggered

1 = Triggered

Bit 4 CMERIF. Flag for interrupt 4 – “Communication error”

0 = Not triggered

1 = Triggered

Bit 3 CTRERIF. Flag for interrupt 3 – “Control error”

0 = Not triggered

1 = Triggered

Bit 2 SWPRIF. Flag for interrupt 2 – “Software protections”

0 = Not triggered

1 = Triggered

Bit 1 PDPIF. Flag for interrupt 1 – “Short-circuit”

0 = Not triggered

1 = Triggered

Bit 0 DSLBIF. Flag for interrupt 0 – “Enable input has changed”

0 = Not triggered

1 = Triggered

6.2.6.9. MCR - Motion Command Register (status, RO)

MCR is a 16-bit status register containing information about the motion modes, reference mode, active control loops, positioning type - absolute or relative, etc.

MPL Address: 0x0309

Contents. MCR information is structured as follows:

15	14	13	12	11	10	9	8
MMODE	MODECHG	POSTYPE	REGMODE	ELGEAR	POSLP	SPDLP	CRTLP
1	0	0	0	0	0	0	0
7-6	6	4-0					
EXTREF	REFLOC	REFTYPE					
0	0	10000					

Bit 15 MMODE. Motion mode

- 0 = Same motion mode
- 1 = New motion mode

Bit 14 MODECHG. When motion mode is changed

- 0 = Update the reference
- 1 = Keep the reference

Bit 13 POSTYPE. Positioning type

- 0 = Relative
- 1 = Absolute

Bit 12 REGMODE. Motion superposition

- 0 = Disable the superposition of the electronic gearing mode with a second motion mode
- 1 = Enable the superposition of the electronic gearing mode with a second motion mode

Bit 11 ELGEAR. Electronic gearing master

- 0 = Disable the axis as master
- 1 = Enable the axis as master

Bit 10 POSLP. Position loop status

- 0 = Disabled
- 1 = Enabled

6.2.7. Bit 9 SPDLP. Speed loop status

- 0 = Disabled
- 1 = Enabled

Bit 8 CRTLP. Current loop status

- 0 = Disabled
- 1 = Enabled

Bit 7-6 EXTREF. External reference type

- 00 = On-line reference
- 01 = Analogue reference
- 10 = Digital reference
- 11 = Reserved

Bit 5 REFLOC. Analogue external reference for torque/voltage mode update

- 0 = Update in slow control loop
- 1 = Update in fast control loop

Bit 4-0 REFTYPE. Reference type

- 00000 = External reference
- 00001 = Trapezoidal reference
- 00010 = Contouring position/speed
- 00011 = Contouring torque/voltage
- 00100 = Pulse & direction
- 00101 = Electronic gearing slave
- 00110 = Electronic camming slave
- 00111 = S-curve reference
- 01000 = Test mode
- 01001 = PVT
- 01010 = PT
- 10000 = Stop 0/1/2
- 10001 = Stop using trapezoidal profile
- 10100 = Stop using S-curve profile
- 10101 = Quickstop

6.2.7.1. MER - Motion Error Register (status, RO)

Purpose: MER is a 16-bit status register. It groups together all the errors conditions. Most of the error conditions trigger the [FAULT status](#).

MPL Address: 0x08FC

Contents. MER information is structured as follows:

15	14	13	12	11	10	9	8
ENST	CMDER	OVER	UVER	OTERD	OTERM	I2TER	OCER
0	0	0	0	0	0	0	0
7	6	5	4	3	2	1	0
LSNST	LSPST	WRPSE	SCIER	CTRER	STPTBL	SCER	CANBER
0	0	0	0	0	0	0	0

Bit 15 ENST. Enable status of drive/motor

- 0 = Enabled
- 1 = Disabled

Bit 14 CMDER. Command error

- 0 = No command error
- 1 = Command error. The bit is set in 2 cases:

	MER.14	SRL.7
An UPD command is received during AXISON command execution	1	0
A cancelable call is received while a MPL function is active following a previous cancelable call	1	1

Bit 13 UVER. Under voltage error

- 0 = No under voltage error
- 1 = Under voltage error

Bit 12 OVER. Over voltage error

- 0 = No over voltage error
- 1 = Over voltage error

Bit 11 OTERD. Drive over temperature error

- 0 = No drive over temperature error
- 1 = Drive over temperature error

Bit 10 OTERM. Motor over temperature error

- 0 = No motor over temperature error
- 1 = Motor temperature error

Bit 9 I2TER. I2T protection error

- 0 = No drive or motor I2T error
- 1 = Drive or motor I2T error

Bit 8 OCER. Over-current error

- 0 = No over-current error
- 1 = Over-current error

Bit 7 LSNST. Negative limit switch status

- 0 = LSN in not active
- 1 = LSN is active

Bit 6 LSPST. Positive limit switch status

- 0 = LSP is not active
- 1 = LSP is active

Bit 5 WRPSER. Hall sensor missing /Resolver error /BiSS error /Position wrap around error

- 0 = No error
- 1 = Error

Bit 4 SCIER. Communication error

- 0 = No serial or internal communication error
- 1 = Serial or internal communication error

Bit 3 CTRER. Control error

- 0 = No control error
- 1 = Control error

Bit 2 STPTBL. Setup table status

- 0 = The drive/motor has a valid setup table
- 1 = The drive/motor has an invalid setup table

Bit 1 SCER. Short-circuit protection status

- 0 = No short-circuit error
- 1 = Short-circuit error

Bit 0 CANBER. CAN bus status

- 0 = No CAN bus error
- 1 = CAN bus error

6.2.7.2. MSR - Motion Status Register (status, RO)

MSR is a 16-bit status register, containing information about motion system status and some specific events like: control error condition, position wrap-around, limit switches and captures triggered by programmed transitions, etc.

MPL Address: 0x0308

Contents. MSR information is structured as follows:

15	14	13	12	11	10	9	8
UPDATE	EVNRS	AXISST	Reserved	EVNS	CNTSGS	MOTS	PCAPS
0	0	0	0	0	0	1	0
7	6	5	4	3	2	1	0
LSWNS	LSWPS	WRPS	Reserved	CTRERS	SWPRS	SCUPD	ENDINIT
0	0	0	0	0	0	0	0

Bit 15 UPDATE. Update the motion mode

- 0 = No update
- 1 = Update

Bit 14 EVNRS. Event status

- 0 = Reset after update
- 1 = Set of update

Bit 13 AXISST. Axis status

- 0 = Axis Off
- 1 = Axis On

Bit 12 Reserved

Bit 11 EVNS. Events

- 0 = No event set, or programmed event not occurred yet
- 1 = Last event reached

Bit 10 CNTSGS. Contour segment

- 0 = Don't update
- 1 = Update

Bit 9 MOTS. Motion status

- 0 = In motion
- 1 = Motion complete

Bit 8 PCAPS. Position capture

- 0 = Not triggered
- 1 = Triggered

Bit 7 LSWNS. Negative limit switch

- 0 = Not triggered
- 1 = Triggered

Bit 6 LSWPS. Positive limit switch

- 0 = Not triggered
- 1 = Triggered

Bit 5 WRPS. Position wrap around

- 0 = Not triggered
- 1 = Triggered

Bit 4 Reserved

Bit 3 CTRERS. Control error status

- 0 = Not triggered
- 1 = Triggered

Bit 2 SWPRS. Software protections status

- 0 = Not triggered
- 1 = Triggered

Bit 1 SCUPD. S-Curve update status

- 0 = S-curve updated successfully
- 1 = S-curve update denied (UPD instruction received when motion was not complete)

Bit 0 ENDINIT. Drive/motor initialization status

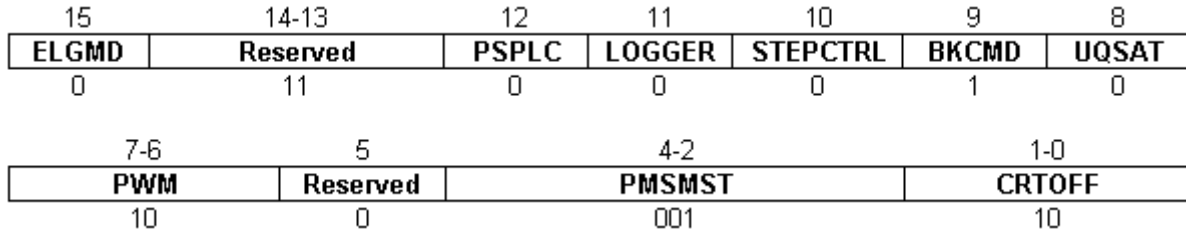
- 0 = Not performed
- 1 = Performed

6.2.7.3. OSR - Operating Settings Register (configuration, R/W)

OSR is a 16-bit configuration register, defines some specific operating settings regarding motor control and data acquisition

MPL Address: 0x0302

Contents. **OSR** information is structured as follows:



Bit 15 ELGMD. Electronic gearing master mode

- 0 = Send actual position to slave axes
- 1 = Send target position to slave axes

Bit 14-13 Reserved

Bit 12 PSPLC. Position sensor mounting place

- 0 = Position sensor on motor
- 1 = Position sensor on load

Bit 11 LOGGER. PMSM start logging

- 0 = No data logging during PMSM motor start
- 1 = Data logging during PMSM motor start

Bit 10 STEPCTRL. Stepper motor control type

- 0 = Open loop
- 1 = Closed loop

Bit 9 BKCMD. Brake command

- 0 = Disabled
- 1 = Enabled

Bit 8 UDQSAT. Ud,q command saturation method

- 0 = Use independently saturated commands on d and q axes
- 1 = Compute Uq from Ud. $Uq = f(Ud)$

Bit 7-6 PWM. PWM command method

- 00 = Standard symmetric PWM
- 01 = Dead-time and Vdc compensation
- 10 = Dead-time, Vdc compensation and third harmonic injection
- 11 = Reserved

Bit 5 Reserved

Bit 4-2 PMSMST. PMSM motor start method

- 000 = Reserved
- 001 = a/b, voltage mode, incremental encoder
- 010 = Start using digital Hall sensors
- 011 = Reserved
- 100 = Reserved
- 101 = Motionless start (encoder only) *
- 110 = Reserved
- 111 = Direct start using absolute encoder

Bit 1-0 CRTOFF. Current offset detection

- 00 = No current offset detection
- 01 = Detection without PWM activated
- 10 = Detection with PWM activated
- 11 = Reserved

*On select firmware versions only

6.2.7.4. PCR - Motion Status Register (command/status, R/W)

PCR is a 16-bit command and status register, containing both masks and status information for MPL protections.

MPL Address: 0x0303

Contents. PCR information is structured as follows:

15	14	13	12	11	10	9	8
I2DPRS	Reserved	UVPRS	OVPRS	OT2PRS	OT1PRS	I2TMPRS	IMXPRS
0	0	0	0	0	0	1	0
7	6	5	4	3	2	1	0
I2DPRM	Reserved	UVPRM	OVPRM	OT2PRM	OT2PRM	I2TMPRS	IMXPRM
0	0	0	0	0	0	0	0

Bit 15 I2DPRS. Status of drive i2t protection

- 0 = Not triggered
- 1 = Triggered

Bit 14 Reserved

Bit 13 UVPRS. Status of under voltage protection

- 0 = Not triggered
- 1 = Triggered

Bit 12 OVPRS. Status of over voltage protection

- 0 = Not triggered
- 1 = Triggered

Bit 11 OT2PRS. Status of drive over temperature protection

- 0 = Not triggered
- 1 = Triggered

Bit 10 OT1PRS. Status of motor over temperature protection

- 0 = Not triggered
- 1 = Triggered

Bit 9 I2TMPRS. Status of motor i2t protection

- 0 = Not triggered
- 1 = Triggered

Bit 8 IMAXP. Status of over current protection

- 0 = Not triggered

1 = Triggered

Bit 7 I2DPRM. Mask for drive I2t protection

0 = Disable

1 = Enable

Bit 6 Reserved

Bit 5 UVPRM. Mask for under voltage protection

0 = Disable

1 = Enable

Bit 4 OVPRM. Mask for over voltage protection

0 = Disable

1 = Enable

Bit 3 OT2PRM. Mask for drive over temperature protection

0 = Disable

1 = Enable

Bit 2 OT1PRM. Mask for motor over temperature protection

0 = Disable

1 = Enable

Bit 1 I2TPRM. Mask for motor I2t protection

0 = Disable

1 = Enable

Bit 0 IMXPRM. Mask for maximum current protection

0 = Disable

1 = Enable

6.2.7.5. SCR - System Configuration Register (configuration, R/W)

SCR is a 16-bit configuration register, defines the basic application configuration regarding the motor type and the feedback sensors used

MPL Address: 0x0300

Contents. **SCR** information is structured as follows:

15	14-12	11-9	8	7	6	5-3	2-0
Reserved	MOTOR	Reserved	TSNS2	TSNS1	Reserved	SSNS	PSNS
0	000	000	0	0	0	000	000

Bit 15 Reserved

Bit 14-12 MOTOR. Motor type

- 000 = Brushless DC
- 001 = Brushed DC
- 010 = Brushless AC
- 011 = Reserved
- 100 = Stepper
- 101 = Tri-phases stepper
- 110 = Reserved
- 111 = Reserved

Bit 11-9 Reserved

Bit 8 TSNS2. Drive temperature sensor

- 0 = Disabled
- 1 = Enabled

Bit 7 TSNS1. Motor temperature sensor

- 0 = Disabled
- 1 = Enabled

Bit 6 Reserved

Bit 5-3 SSNS. Speed sensor

- 000 = Position difference
- 001 = Tachogenerator
- 010 = Pulse length from Hall sensor
- 011 = Reserved
- 100 = Reserved
- 101 = Reserved
- 110 = Reserved
- 111 = None

Bit 2-0 PSNS. Position sensor

- 000 = Quadrature encoder
- 001 = Resolver
- 010 = Sin-cos with/without EnDat
- 011 = SSI
- 100 = Linear Hall
- 101 = BiSS encoder
- 110 = Reserved
- 111 = None

6.2.7.6. SRH - Status Register High part (status, RO)

Purpose: SRH is the high part of a the status register grouping together all the key status information concerning the drive/motor

MPL Address: 0x090F

Contents. SRH information is structured as follows:

15	14	13	12	11	10	9	8
FAULT	INCAM	Reserved	INGEAR	I2TWRGD	I2TWRGM	TRGR	PCAPS
0	0	0	0	0	0	1	0
7	6	5	4	3	2	1	0
LSWNS	LSWPS	AUTORUN	PTRG4	PTRG3	PTRG2	PTRG1	ENDINIT
0	0	0	0	0	0	0	0

Bit 15 FAULT. Fault status

- 0 = No fault
- 1 = Drive/motor in fault status

Bit 14 INCAM. Reference position in absolute electronic camming mode

- 0 = Not reached
- 1 = Reached

Bit 13 Reserved

Bit 12 INGEAR Gear ratio in electronic gearing mode

- 0 = Not reached
- 1 = Reached

Bit 11 I2TWRGD. Drive I2T protection warning

- 0 = Drive I2T warning limit not reached
- 1 = Drive I2T warning limit reached

Bit 10 I2TWRGM. Motor I2T protection warning

- 0 = Motor I2T warning limit not reached
- 1 = Motor I2T warning limit reached

Bit 9 TRGR. Target command

- 0 = Not reached
- 1 = Reached

Bit 8 PCAPS. Capture event/interrupt

- 0 = Not triggered
- 1 = Triggered

Bit 7 LSWNS. Limit switch negative event/interrupt

- 0 = Not triggered
- 1 = Triggered

Bit 6 LSWPS. Limit switch positive event/interrupt

- 0 = Not triggered
- 1 = Triggered

Bit 5 AUTORUN. AUTORUN mode status

- 0 = Disabled
- 1 = Enabled

Bit 4 PTRG4. Position trigger 4

- 0 = Not reached
- 1 = Reached

Bit 3 PTRG3. Position trigger 3

- 0 = Not reached
- 1 = Reached

Bit 2 PTRG2. Position Trigger 2

- 0 = Not reached
- 1 = Reached

Bit 1 PTRG1. Position Trigger 1

- 0 = Not triggered
- 1 = Triggered

Bit 0 ENDINIT. Drive/motor initialization status

- 0 = Not performed
- 1 = Performed

See also:

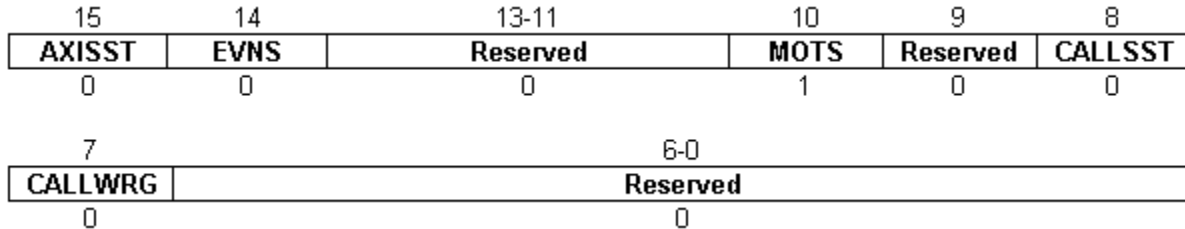
[Status register low part – SRL](#)

6.2.7.7. SRL - Status Register Low part (status, RO)

Purpose: SRL is the low part of a status register grouping together all the key status information concerning the drive/motor

MPL Address: 0x090E

Contents. SRL information is structured as follows:



Bit 15 AXISST. Axis status

- 0 = Axis Off
- 1 = Axis On

Bit 14 EVNS. Events

- 0 = No event set, or programmed event not occurred yet
- 1 = Last programmed event reached

Bits 13-11 Reserved.

Bit 10 MOTS. Motion status

- 0 = In motion
- 1 = Motion complete

Bit 9 Reserved.

Bit 8 CALLSST. Cancelable call status

- 0 = No function in execution following a cancelable call
- 1 = A function in execution following a cancelable call

Bit 7 CALLWRG. Cancelable call warning

- 0 = No warning
- 1 = Warning – a cancelable call is issued while another cancelable function is in execution

Bits 6-0 Reserved

See also:

[Status register high part – SRH](#)

6.2.7.8. SSR - Slave Status Register (status, RO)

SSR is a 32-bit status register containing information about slave axes initialization status, setup table status, firmware compatibility and slave presence in the CAN network.

MPL Address: 0x07F2

Contents. **SSR** information is structured as follows:

31	30	29	28	27	26	25	24
HIERR	HIFW	HISTP	HDET	GIERR	GIFW	GISTP	GDET
0	0	0	0	0	0	1	0
23	22	21	20	19	18	17	16
FIERR	FIFW	FISTP	FDET	EIERR	EIFW	EISTP	EDET
0	0	0	0	0	0	0	0
15	14	13	12	11	10	9	8
DIERR	DIFW	DISTP	DDET	CIERR	CIFW	CISTP	CDET
0	0	0	0	0	0	1	0
7	6	5	4	3	2	1	0
BIERR	BIFW	BISTP	BDET	AIERR	AIFW	AISTP	ADET
0	0	0	0	0	0	0	0

Bit 31 HIERR. H slave initialization status

- 0 = Initialization successful
- 1 = Initialization error

Bit 30 HIFW. H slave firmware compatibility with motion controller

- 0 = Firmware compatible
- 1 = Firmware incompatible

Bit 29 HISTP. H slave invalid setup table

- 0 = Setup table valid
- 1 = Invalid setup table

Bit 28 HDET. H slave detection

- 0 = Detected successfully
- 1 = Not detected

Bit 27 GIERR. G slave initialization status

- 0 = Initialization successful
- 1 = Initialization error

Bit 26 GIFW. G slave firmware compatibility with motion controller

- 0 = Firmware compatible
- 1 = Firmware incompatible

Bit 25 GISTP. G slave invalid setup table

- 0 = Setup table valid
- 1 = Invalid setup table

Bit 24 GDET. G slave detection

- 0 = Detected successfully
- 1 = Not detected

Bit 23 FIERR. F slave initialization status

- 0 = Initialization successful
- 1 = Initialization error

Bit 22 FIFW. F slave firmware compatibility with motion controller

- 0 = Firmware compatible
- 1 = Firmware incompatible

Bit 21 FISTP. F slave invalid setup table

- 0 = Setup table valid
- 1 = Invalid setup table

Bit 20 FDET. F slave detection

- 0 = Detected successfully
- 1 = Not detected

Bit 19 EIERR. E slave initialization status

- 0 = Initialization successful
- 1 = Initialization error

Bit 18 EIFW. E slave firmware compatibility with motion controller

- 0 = Firmware compatible
- 1 = Firmware incompatible

Bit 17 EISTP. E slave invalid setup table

- 0 = Setup table valid
- 1 = Invalid setup table

Bit 16 EDET. E slave detection

- 0 = Detected successfully
- 1 = Not detected

Bit 15 DIERR. D slave initialization status

- 0 = Initialization successful
- 1 = Initialization error

Bit 14 DIFW. D slave firmware compatibility with motion controller

- 0 = Firmware compatible
- 1 = Firmware incompatible

Bit 13 DISTP. D slave invalid setup table

- 0 = Setup table valid
- 1 = Invalid setup table

Bit 12 DDET. D slave detection

- 0 = Detected successfully
- 1 = Not detected

Bit 11 CIERR. C slave initialization status

- 0 = Initialization successful
- 1 = Initialization error

Bit 10 CIFW. C slave firmware compatibility with motion controller

- 0 = Firmware compatible
- 1 = Firmware incompatible

Bit 9 CISTP. C slave invalid setup table

- 0 = Setup table valid
- 1 = Invalid setup table

Bit 8 CDET. C slave detection

- 0 = Detected successfully
- 1 = Not detected

Bit 7 BIERR. B slave initialization status

- 0 = Initialization successful
- 1 = Initialization error

Bit 6 BIFW. B slave firmware compatibility with motion controller

- 0 = Firmware compatible
- 1 = Firmware incompatible

Bit 5 BISTP. B slave invalid setup table

- 0 = Setup table valid
- 1 = Invalid setup table

Bit 4 BDET. B slave detection

- 0 = Detected successfully
- 1 = Not detected

Bit 3 AIERR. A slave initialization status

0 = Initialization successful

1 = Initialization error

Bit 2 AIFW. A slave firmware compatibility with motion controller

0 = Firmware compatible

1 = Firmware incompatible

Bit 1 AISTP. A slave invalid setup table

0 = Setup table valid

1 = Invalid setup table

Bit 0 ADET. A slave detection

0 = Detected successfully

1 = Not detected

6.2.7.9. UPGRADE - Upgrade Register (configuration, R/W)

UPGRADE is a 16-bit status register, defining new options and extended features that are activated when their corresponding bits are set.

MPL Address: 0x0857

Contents. UPGRADE information is structured as follows:

15	14	13	12	11	10	9	8
STPTBL	Reserved	TXBUFF	TINTQSTP	MCM	I2TPRT	IPOS	IORW
0	0	0	0	0	0	0	0
7	6	5	4	3	2	1	0
ATIME	FSTSLW	STBCRT	SPDCTR	REG	LMTSPDACC	STPMD	AREFLMT
0	0	0	0	0	0	0	0

Bit 15 STPTBL. Setup table

- 0 = Valid setup table is not required
- 1 = Valid setup table is required

Bit 14 Reserved

Bit 13 TXBUFF. CAN-bus transmit buffer length

- 0 = The length of CAN-bus transmit buffer is 1 messages
- 1 = The length of CAN-bus transmit buffer is 5 messages

Bit 12 TINTQSTP. MPL time interrupt/quickstop

- 0 = Disable
- 1 = Enable MPL time interrupt and quickstop mode when a limit switch is reached

Bit 11 MCM. Motion complete mode

- 0 = Motion complete set when the position reference arrives at the commanded position
- 1 = Motion complete set when the position feedback arrives at the commanded position and remains in a settle band for a preset stabilize time interval

Bit 10 I2TPRT. I2T protection

- 0 = One I2T protection common for drive and motor
- 1 = Two I2T protections, one for drive and the other for the motor

Bit 9 IPOS. Initial positioning mode

- 0 = Standard – wait time per phase up to 1s
- 1 = Extended – wait time per phase up to 635s

Bit 8 IORW. I/O lines read/write

- 0 = Simultaneous read /write of 4 general purpose inputs/outputs
- 1 = Simultaneous read 4 general-purpose inputs and 3 dedicated inputs: Enable, LSP and LSN. Simultaneous set 4 general-purpose outputs and 2 dedicated outputs: Ready and Error.

Bit 7 ATIME. Absolute time start

- 0 = After instruction ENDINIT
- 1 = After power on

Bit 6 FSTSLW. Position/speed control mode

- 0 = Position/speed control in slow loop
- 1 = Position/speed control in fast loop

Bit 5 STBCRT. Standby current for step motors

- 0 = Disable
- 1 = Enable

Bit 4 SPDCTR. Speed control error protection

- 0 = Common with position control error protection
- 1 = Separate control error protection for position and speed

Bit 3 REG. Registration mode

- 0 = Disabled
- 1 = Enabled

Bit 2 LMTSPDACC. Maximal speed/acceleration in motion modes: external, electronic gearing and electronic camming

- 0 = Unlimited
- 1 = Limited

Bit 1 STPMD. Stop mode for steppers

- 0 = Disabled
- 1 = Enabled

Bit 0 AREFLMT. Analogue reference

- 0 = Symmetrical, only positive or only negative
- 1 = Separately programmable upper and lower limits

6.3. Internal Units and Scaling Factors

ElectroCraft drives/motors work with parameters and variables represented in internal units (IU). The parameters and variables may represent various signals: position, speed, current, voltage, etc. Each type of signal has its own internal representation in IU and a specific scaling factor. In order to easily identify each type of IU, these have been named after the associated signals. For example the **position units** are the internal units for position, the **speed units** are the internal units for speed, etc.

The scaling factor of each internal unit shows the correspondence with the international standard units (SI). The scaling factors are dependent on the product, motor and sensor type. Put in other words, the scaling factors depend on the setup configuration.

In order to find the internal units and the scaling factors for a specific case, select the application in the project window and then execute menu command **Help | Application Programming | Internal Units and Scaling Factors**.

***Important:** The **Help | Application Programming | Internal Units and Scaling Factors** command provides customized information function of the application setup. If you change the drive, the motor technology or the feedback device, check again the scaling factors with this command. It may show you other relations!*

6.4. PRO EEPROM Programmer

6.4.1. PRO EEPROM Programmer

All ElectroCraft drives/motors include a non-volatile EEPROM memory. Its role is to:

- Keep the setup data in a dedicated area named **setup table** together with a user programmable **application ID**, which helps you quickly identify the setup data uploaded from a drive/motor.
- Store the **MPL motion programs** and their associated data like the **cam tables** needed for electronic camming applications.
- Keep the **product ID** of each drive/motor and the required **firmware ID** for the programmed application.

***Remark:** The required firmware ID indicates that the actual drive/motor firmware ID must have the same number and a revision letter equal or higher. For example if the required firmware ID is: F000H, the actual drive/motor firmware ID must be F000H or F000I, or F000J, etc.*

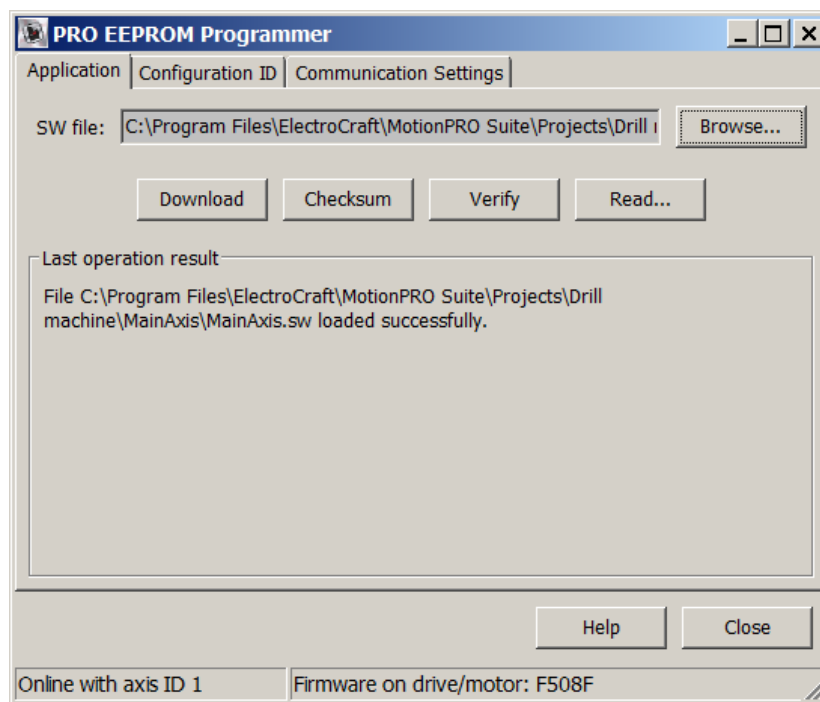
The Drive/Motor PRO EEPROM Programmer is a tool specifically designed for production, through which you can:

- Program fast and easy the EEPROM memory of any ElectroCraft drive/motor with all the data needed to run a specific application. These data are grouped into a unique file named **software file** (with extension **.sw**)
- Check EEPROM data integrity by comparing the information read from the drive/motor memory with that read from a **.sw** software file
- Write protect a part or the entire EEPROM memory.
- Get information about the drive/motor configuration ID including the product ID, the firmware ID and the application ID

The Drive/Motor PRO EEPROM Programmer is included in both the PROconfig and MotionPRO Developer installation packages and is automatically installed with them. However, it may also be installed separately. To launch the Drive/Motor EEPROM from Windows Start menu execute: “Start | Programs | PROconfig | Drive/Motor PRO EEPROM Programmer” or “Start | Programs | MotionPRO Developer | Drive/Motor PRO EEPROM Programmer” depending on which installation package you have used. You can also start the Drive/Motor PRO EEPROM Programmer from the main folder of the PROconfig / MotionPRO Developer by executing “**PRO EEPROMprog.exe**”.

The Drive/Motor PRO EEPROM Programmer has 3 tabs: **Application**, **Configuration ID** and **Communication Settings**

In the **Application** tab you select a **.sw** software file. Use the **Download** button to program it into the drive/motor EEPROM memory. Use the **Verify** button to check if the information stored in the drive/motor EEPROM is identical with that from the selected **.sw** file. Press the **Checksum** button to compute the sum modulo 65536 of all the data from a **.sw** file. The checksum result may be used by a master during the application initialization to validate that data from a drive/motor EEPROM memory is correct and complete. For example, the host can ask a drive/motor to return the checksum for each block of continuous data from the EEPROM, according with the **.sw** file. By adding the results returned by the drive/motor, the host obtains a global checksum which must match with the value got when the **Checksum** button is pressed. Use the **Read...** button to save the contents of the whole EEPROM memory in a **.sw** file.

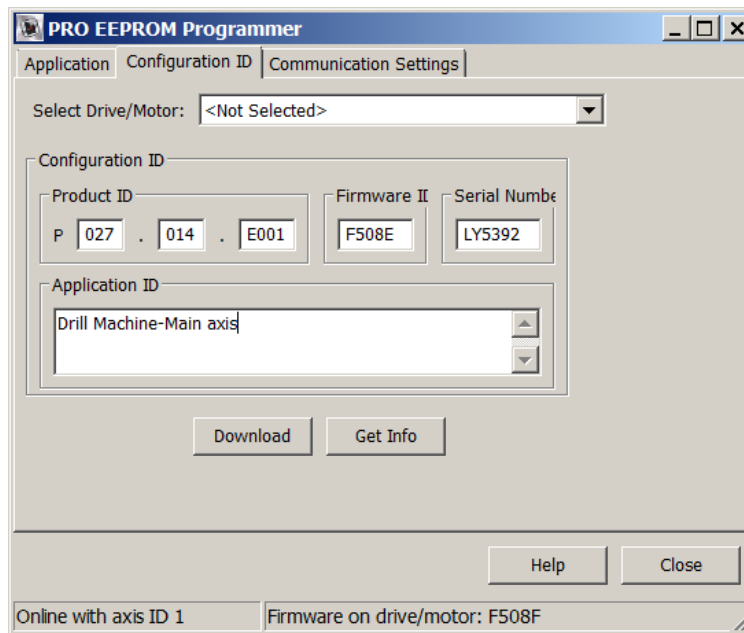


The Drive/Motor PRO EEPROM Programmer signals with an **OK** inside a green disk that the:

- **Download** operation is correctly executed
- **Verify** operation, gives a match between the **.sw** file and the drive/motor EEPROM contents
- **Read** operation is successfully ended and the **.sw** file is created

Otherwise, an **ERROR** inside a red disk is displayed together with a message explaining the error type.

In the **Configuration ID** tab, by pressing the **Get Info** button, you get the drive/motor configuration ID including the product ID, the firmware ID, the EEPROM size and the application ID.



The **product ID** uniquely identifies the drive/motor execution. This information is written by ElectroCraft in the last EEPROM memory locations in especially reserved locations. In these locations ElectroCraft also puts the product **EEPROM size** in 16-bit Kwords and the required **firmware ID**. The main goal of this information is to protect against accidental wrong programming of the EEPROM memory or in the case of very big MPL programs against bypassing the EEPROM capacity. Both PROconfig and MotionPRO Developer perform the following verifications every time a setup data or a motion application has to be downloaded.

- 1) The product ID of the application/setup data to download matches with the product ID stored in the drive/motor EEPROM, or is set as being compatible
- 2) The required firmware of the application/setup data to download has the same number as the drive/motor actual firmware and either the same or a lower revision

The download is performed only if both conditions are true. The **application ID** is a space reserved for a text of up to 40 characters which you can program. Its main goal is to help you quickly identify the setup data uploaded from a drive/motor. In order to program an application ID, edit your text in the Application ID field and press the **Download** button.

The configuration ID tab may also be used to reprogram the product ID, the required firmware ID and the EEPROM size, if by mistake, the area reserved for this information in the EEPROM memory is erased. In this case, select the product name from the list, add your application ID (if it is the case) and press the **Download** button.

In the **Communication Settings** tab, you can set the communication type and parameters as well as the EEPROM write protection degree. When you launch the PRO EEPROM Programmer, it tries to communicate with your drive/motor using your last communication settings. If the communication attempt fails, the PRO EEPROM Programmer opens automatically the Communication Settings tab, where you can setup the communication parameters(HELP_COMMUNICATION_SETUP@PROconfig.hlp). Each time when you'll try to switch to the other tabs, the communication is checked and the other tabs are opened ONLY if the communication is established.

If your application does not require storing data in the drive/motor EEPROM at runtime and you don't intend to change the setup parameters from your host and then to save the changes in the drive/motor EEPROM, you can write protect the entire EEPROM after you download the **.sw** file. This is an extra protection against accidental wrong commands that may modify EEPROM locations. If your application requires to store data at runtime but you don't and you don't intend to change the setup parameters and maybe cam tables (if present) you can write protect only the last quarter or last half of the EEPROM and allow the write operation for the rest.

See also:

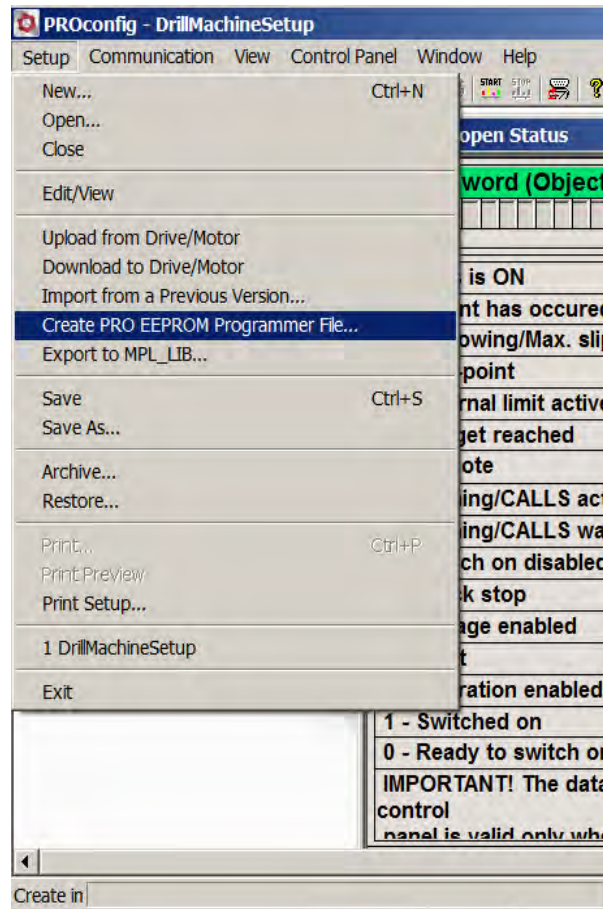
[Software Files Creation and Format](#)

[Communication Setup](#)

6.4.2. PRO EEPROM Programmer File Format

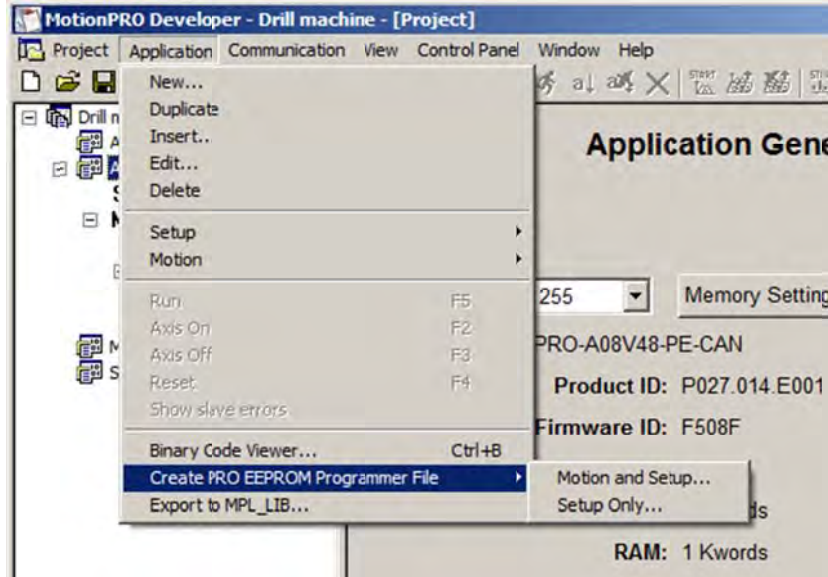
The **.sw** software files can be generated either from PROconfig or from MotionPRO Developer.

In PROconfig you create a **.sw** file with the command **Setup | PRO EEPROM Programmer File...**



The software file generated, includes the setup data and the drive/motor configuration ID with the user programmable application ID. Typically, this type of **.sw** file is used in applications where the motion programming is done from the host using for example one of the MPL_LIB motion libraries offered by ElectroCraft for: PC applications (written in C/C++, Visual Basic, Delphi Pascal, Labview) or for PLCs according with the PLCOpen standard.

In MotionPRO Developer you create a **.sw** file with one of the commands: **Application | PRO EEPROM Programmer File | Motion and Setup** or **Setup Only**. The option **Motion and Setup** creates a **.sw** file with complete information including setup data, MPL programs, cam tables (if present) and the drive/motor configuration ID. The option **Setup Only** produces a **.sw** file identical with that produced by PROconfig i.e. having only the setup data and the configuration ID.



Software File Format

A software file (with extension **.sw**) is a text file that can be read with any text editor. It contains blocks of data separated by an empty row. Each block of data starts with the block start address, followed by data values to place in ascending order at consecutive addresses: first data – to write at start address, second data – to write at start address + 1, etc. All the data are hexadecimal 16-bit values (maximum 4 hexadecimal digits). Each row contains a single data value. When less than 4 hexadecimal digits are shown, the value must be right justified. For example 92 represent 0x0092.

See also:

[Drive/Motor PRO EEPROM Programmer](#)

Appendix A : MPL Instructions List

[A/G] { MPL Instr}	Send MPL instruction to [A/G]
[A/G] V16D = V16S	[A/G] V16D = local V16S
[A/G] V16D, dm = V16S	[A/G] V16D = local V16S (fa)
[A/G] (V16D), TM = V16S	[A/G] (V16D), TM = local V16S
[A/G] (V16D+), TM = V16S	[A/G] (V16D), TM = local V16S, then V16D += 1
[A/G] V32D = V32S	[A/G] V32D = local V32S
[A/G] V32D, dm = V32S	[A/G] V32D = local V32S (fa)
[A/G] (V16D), TM = V32S	[A/G] (V16D), TM = local V32S
[A/G] (V16D+), TM = V32S	[A/G] (V16D), TM = local V32S, then V16D += 2
(?)GiveMeData	Ask one axis to return a 16/32 bit data from memory
(??)GiveMeData2	Ask a group of axes to return each a 16/32 bit data from memory
!ALPO	Set event when absolute load position is over a value
!ALPU	Set event when absolute load position is under a value
!AMPO	Set event when absolute motor position over a value
!AMPU	Set event when absolute motor position under a value
!CAP	Set event when a capture input goes low or high
!IN#n	Set event when digital input #n goes low or high
!LSN	Set event when the negative limit switch (LSN) goes low or high
!LSP	Set event when positive limit switch (LSP) goes low or high
!LSO	Set event when load speed is over a value
!LSU	Set event when load speed is under a value
!MC	Set event when the actual motion is completed
!MSO	Set event when motor speed is over a value
!MSU	Set event when motor speed is under a value
!PRO	Set event when position reference is over a value
!PRU	Set event when position reference is under a value
!RPO	Set event when relative load position is over a value

<u>!RPU</u>	Set event when relative load position is under a value
<u>!RT</u>	Set event after a wait time
<u>!SRO</u>	Set event if speed reference is over a value
<u>!SRU</u>	Set event if speed reference is under a value
<u>!TRO</u>	Set event if torque reference is over a value
<u>!TRU</u>	Set event if torque reference is under a value
<u>!VO</u>	Set event if a long/fixe d variable is over a value
<u>!VU</u>	Set event if a long/fixe d variable is under a value
<u>ABORT</u>	Abort the execution of a function called with CALLS
<u>ADDGRID (value16_1, value16_2,...)</u>	Add groups to the Group ID
<u>AXISID</u>	Set Axis ID
<u>AXISOFF</u>	AXIS is OFF (deactivate control)
<u>AXISON</u>	AXIS is ON (activate control)
<u>BEGIN</u>	BEGIN of a MPL program
<u>CALL</u>	Call a MPL function
<u>CALLS</u>	Cancelable CALL of a MPL function
<u>CANBR val16</u>	Set CAN bus baud rate
<u>CHECKSUM, TM Start, Stop, V16D</u>	V16D=Checksum between Start and Stop addresses from TM
<u>CIRCLE</u>	Define circular segment for vector mode
<u>CPA</u>	Command Position is Absolute
<u>CPR</u>	Command Position is Relative
<u>DINT</u>	Disable globally all MPL interrupts
<u>DISCAPI</u>	Disable 1st capture/encoder index input to detect transitions
<u>DIS2CAPI</u>	Disable 2nd capture/encoder index input to detect transitions
<u>DISLSN</u>	Disable negative limit switch (LSN) input to detect transitions
<u>DISLSP</u>	Disable positive limit switch (LSP) input to detect transitions
<u>EINT</u>	Enable globally all MPL interrupts
<u>EN2CAPI0</u>	Enable 2nd capture/encoder index input to detect a high to low transition
<u>EN2CAPI1</u>	Enable 2nd capture/encoder index input to detect a low to high transition
<u>ENCAPI0</u>	Enable 1st capture/encoder index input to detect a high to low transition
<u>ENCAPI1</u>	Enable 1st capture/encoder index input to detect a low to high transition

END	END of a MPL program
ENDINIT	END of Initialization
ENEEPROM	Enables EEPROM usage after it was disabled by the initialization of SSI or ENDat encoders
ENLSN0	Enable negative limit switch (LSN) input to detect a high to low transition
ENLSN1	Enable negative limit switch (LSN) input to detect a low to high transition
ENLSP0	Enable positive limit switch (LSP) input to detect a low to high transition
ENLSP1	Enable positive limit switch (LSP) input to detect a high to low transition
EXTREF	Set external reference type
FAULTR	Reset FAULT status. Return to normal operation
Get checksum	Ask one axis to return the checksum between 2 addresses from its MPL memory
GETERROR	Get last error reported by slaves
GetMPLData	Ask one axis to return a MPL data
GetVersion	Ask one axis the firmware version
GOTO	Jump
GROUPID (value16 1, value16 2,...)	Set GROUP ID
V16D = IN#n	Read input #n. V16D = input #n status
INITCAM addrS, addrD	Copy CAM table from EEPROM (addrS address) to RAM (addrD address)
V16D = INPUT1 , ANDm	V16D = logical AND between inputs IN#25 to IN#32 status and ANDm mask
V16D = INPUT2 , ANDm	V16D = logical AND between inputs IN#33 to IN#39 status and ANDm mask
V16D = INPORT , ANDm	V16D = status of inputs Enable, LSP, LSN plus IN#36 to IN#39
LOCKEEPROM	Locks or unlocks the EEPROM write protection
LPLANE	Define coordinate system for linear interpolation mode
MODE_CS	Set MODE Cam Slave
MODE_GS	Set MODE Gear Slave
MODE_LI	Set MODE Linear Interpolation
MODE_PC	Set MODE Position Contouring
MODE_PE	Set MODE Position External
MODE_PP	Set MODE Position Profile
MODE_PSC	Set MODE Position S-Curve
MODE_PT	Set MODE PT
MODE_PVT	Set MODE PVT
MODE_SC	Set MODE Speed Contouring

<u>MODE SE</u>	Set MODE Speed External
<u>MODE SP</u>	Set MODE Speed Profile
<u>MODE TC</u>	Set MODE Torque Contouring
<u>MODE TEF</u>	Set MODE Torque External Fast
<u>MODE TES</u>	Set MODE Torque External Slow
<u>MODE TT</u>	Set MODE Torque Test
<u>MODE VC</u>	Set MODE Voltage Contouring
<u>MODE VEF</u>	Set MODE Voltage External Fast
<u>MODE VES</u>	Set MODE Voltage External Slow
<u>MODE VM</u>	Set MODE Vector Mode
<u>MODE VT</u>	Set MODE Voltage Test
<u>NOP</u>	No Operation
<u>OUT(n) =value16</u>	Set the output line as specified by value16
<u>OUT(n1, n2, n3, ...) =value16</u>	Set the output lines n1 n2, n3 as specified by value16
<u>OUTPORT</u>	Set Enable, LSP, LSN and general purpose outputs OUT#28-31
<u>PING</u>	Ask a group of axes to return their axis ID
<u>PONG</u>	Answer to a PING request
<u>PROD <<= N</u>	Left shift 48-bit product register by N
<u>PROD >>= N</u>	Right shift 48-bit product register by N
<u>PTP</u>	Define a PT point
<u>PVTP</u>	Define a PVT point
<u>REG_OFF</u>	Disable superposed mode
<u>REG_ON</u>	Enable superposed mode
<u>REMGRID (value16_1, value16_2,...)</u>	Remove groups from the Group ID
<u>RESET</u>	RESET drive / motor
<u>RET</u>	Return from a MPL function
<u>RETI</u>	Return from a MPL Interrupt Service Routine
<u>RGM</u>	Reset electronic gearing/camming master mode
<u>ROUT#n</u>	Set low the output line #n
<u>SAP</u>	Set Actual Position
<u>SAVE</u>	Save setup data in the EEPROM memory
<u>SAVEERROR</u>	Save slave error in EEPROM
<u>SCIBR V16</u>	Set RS-232/Rs485 serial communication interface (SCI) baud rate

SEG	Define a contouring segment
SEND	Send to host the contents of a MPL variable
SetAsInput(n)	Set the I/O line #n as an input
SetAsOutput(n)	Set the I/O line #n as an output
SETIO#n	Set IO line #n as input or as output
SETPT	Setup PT mode operation
SETPVT	Setup PVT mod operation
SETSUNC	Enable/disable synchronization between axes
SGM	Set electronic gearing/camming master mode
SOUT#n	Set high the output line #n
SRB V16, ANDm, ORm	Set / Reset Bits from V16
SRBL V16, ANDm, ORm	Set / Reset Bits from V16 (fa)
STA	Set Target position = Actual position
STARTLOG V16	Start the data acquisition
STOP	STOP motion
STOP!	STOP motion when the programmed event occurs
STOPLOG	Stop the data acquisition
Take checksum	Answer to Get checksum request
TakeData	Answer to GiveMeData request
TakeData	Answer to Get MPL Data request
TakeData2	Answer to GiveMeData2 request
TakeVersion	Answer to Get version request
TUM0	Target update mode 0
TUM1	Target update mode 1
UPD	Update motion mode and parameters. Start motion
UPD!	Update motion mode and parameters when the programmed event occurs
VPLANE	Define coordinate system for Vector Mode
V16D = [A] V16S	Local V16D = [A] V16S
V16D = [A] V16S, dm	Local V16D = [A] V16S, dm (fa)
V16D = [A] (V16S), TM	Local V16D = [A] (V16S), dm
V16D = [A] (V16S+), TM	Local V16D = [A] (V16S), dm, then V16S += 1
V32D = [A] V32S	Local V32D = [A] V32S
V32D = [A] V32S, dm	Local V32D = [A] V32S, dm (fa)

V32D = [A] (V16S), TM	Local V32D = [A] (V16S), TM
V32D = [A] (V16S+), TM	Local V32D = [A] (V16S), TM, then V16S += 2
V16 = label	V16 = &label
V16D = V16S	V16D = V16S
V16 = val16	V16 = val16
V16D = V32S(H)	V16D = V32S(H)
V16D = V32S(L)	V16D = V32S(L)
V16D, dm = V16S	V16D = V16S (fa)
V16D, dm = val16	V16D = val16 (fa)
V16D = (V16S), TM	V16D = (V16S) from TM memory
V16D = (V16S+), TM	V16D = (V16S) from TM memory, then V16S += 1
(V16D), TM = V16S	(V16D) from TM memory = V16S
(V16D), TM = val16	(V16D) from TM memory = val16
(V16D+), TM = V16S	(V16D) from TM memory = V16S, then V16D += 1
(V16D+), TM = val16	(V16D) from TM memory = val16, then V16D += 1
V32(H) = val16	V32(H) = val16
V32(L) = val16	V32(L) = val16
V32D(H) = V16S	V32D(H) = V16
V32D(L) = V16S	V32D(L) = V16
V16D = -V16S	V16D = -V16S
V32D = V32S	V32D = V32S
V32 = val32	V32 = val32
V32D = V16S << N	V32D = V16S left-shifted by N
V32D, dm = V32S	V32D from dm = V32S (fa)
V32D, dm = val32	V32 from dm = val32 (fa)
V32D = (V16S), TM	V32D = (V16S) from TM memory
V32D = (V16S+), TM	V32D = (V16S) from TM memory, then V16S += 2
(V16D), TM = V32S	(V16D) from TM memory = V32S
(V16D), TM = val32	(V16D) from TM memory = val32
(V16D+), TM = V32S	(V16D) from TM memory = V32S, then V16D += 2
(V16D+), TM = val32	(V16D) from TM memory = val32, then V16D += 2
V32D = -V32S	V32D = -V32S
V16 += val16	V16 = V16 + val16

V16D += V16S	V16D = V16D + V16S
V32 += val32	V32 = V32 + val32
V32D += V32S	V32D = V32D + V32S
V16 -= val16	V16 = V16 - val16
V16D -= V16S	V16D = V16D - V16S
V32 -= val32	V32 = V32 - val32
V32D -= V32S	V32D = V32D - V32S
V16 * val16 << N	48-bit product register = (V16 * val16) >> N
V16 * val16 >> N	48-bit product register = (V16 * val16) >> N
V16A * V16B << N	48-bit product register = (V16A * V16B) << N
V16A * V16B >> N	48-bit product register = (V16A * V16B) >> N
V32 * V16 << N	48-bit product register = (V32 * V16) << N
V32 * V16 >> N	48-bit product register = (V32 * V16) >> N
V32 * val16 << N	48-bit product register = (V32 * val16) << N
V32 * val16 >> N	48-bit product register = (V32 * val16) >> N
V32=V16	Divide V32 to V16
V16 <<= N	Left shift V16 by N
V32 <<= N	Left shift V32 by N
V16 >>= N	Right shift V16 by N
V32 >>= N	Right shift V32 by N
VSEG	Define linear segment for vector mode
WAIT!	Wait until the programmed event occurs
WALPO	Set and wait event when slave's absolute load position is over a value
WALPU	Set and wait event when slave's absolute load position is under a value
WAMPO	Set and wait event when slave's absolute motor position over a value
WAMPU	Set and wait event when absolute motor position under a value
WCAP	Set and wait event when slave's 1st capture/encoder index input goes low or high
W2CAP	Set and wait event when slave's 2nd capture/encoder index input goes low or high
WIN#n	Set and wait event when slave's digital input #n goes low or high
WLSN	Set event when slave's negative limit switch (LSN) goes low or high
WLSP	Set event when slave's positive limit switch (LSP) goes low or high
WLSQ	Set event when slave's load speed is over a value

<u>WLSU</u>	Set event when slave's load speed is under a value
<u>WMC</u>	Set and wait event when the actual motion is completed on one or more slave axes
<u>WMSO</u>	Set and wait event when slave's motor speed is over a value
<u>WMSU</u>	Set and wait event when slave's motor speed is under a value
<u>WPRO</u>	Set and wait event when slave's position reference is over a value
<u>WPRU</u>	Set and wait event when slave's position reference is under a value
<u>WRPO</u>	Set and wait event when slave's relative load position is over a value
<u>WRPU</u>	Set and wait event when slave's relative load position is under a value
<u>WRT</u>	Set event after a wait time
<u>WVDU</u>	Set and wait event when the vector distance is under a value
<u>WVDO</u>	Set and wait event when the vector distance is over a value
<u>WTR</u>	Set and wait event when the slave's target is reached

7. Appendix B : MPL Data List

AAR	Type: UINT Address: 0x030C
ACR	Type: UINT Address: 0x0912
AD5	Type: UINT Address: 0x0241
AD5 OFF	Type: INT Address: 0x0249
APOS	Type: LONG Address: 0x0228
APOS_MT	Type: LONG Address: 0x0988
APOS2	Type: LONG Address: 0x081C
ASPD	Type: FIXED Address: 0x022C
ASPD_LD	Type: FIXED Address: 0x098A
ASPD_MT	Type: FIXED Address: 0x098A
ATIME	Type: LONG Address: 0x02C0
BRAKELIM	Type: UINT Address: 0x028A
CACC	Type: FIXED Address: 0x02A2
CADIN	Type: INT Address: 0x025C
CAMINPUT	Type: LONG Address: 0x0901
CAMOFF	Type: LONG Address: 0x03AD

CAMSTART	Type: INT Address: 0x03AC
CAMX	Type: FIXED Address: 0x0903
CAMY	Type: FIXED Address: 0x0905
CAPPOS	Type: LONG Address: 0x02BC
CAPPOS2	Type: LONG Address: 0x081E
CDEC	Type: FIXED Address: 0x0859
CLPER	Type: INT Address: 0x0250
CPOS	Type: LONG Address: 0x029E
CSPD	Type: FIXED Address: 0x02A0
DBT	Type: UINT Address: 0x0253
DIGIN_ACTIVE_LEVEL	Type: UINT Address: 0x090C
DIGIN_INVERSION_MASK	Type: UINT Address: 0x090A
DIGOUT_INVERSION_MASK	Type: UINT Address: 0x090B
E_LEVEL_AD5	Type: INT Address: 0x0870
ELRESL	Type: LONG Address: 0x0875
ENC2THL	Type: LONG Address: 0x024C
EREFP	Type: LONG Address: 0x02A8
EREFS	Type: FIXED Address: 0x02A8

EREFT	Type: INT Address: 0x02A9
EREFV	Type: INT Address: 0x02A9
ERRMAX	Type: INT Address: 0x02C5
FILTER1	Type: INT Address: 0x029D
FILTERQ	Type: INT Address: 0x0982
GEAR	Type: FIXED Address: 0x02AC
GEARMASTER	Type: INT Address: 0x0255
GEARSLAVE	Type: INT Address: 0x0256
HALL30	Type: INT Address: 0x0877
HALLCASE	Type: INT Address: 0x0259
HOMEPOS	Type: LONG Address: 0x0992
HOMESPD	Type: FIXED Address: 0x0994
I2TINTLIM_D	Type: ULONG Address: 0x0980
I2TINTLIM_M	Type: ULONG Address: 0x0815
I2TWARLIM_M	Type: ULONG Address: x097E
ICR	Type: UINT Address: 0x0304
INSTATUS	Type: UINT Address: 0x0908
INTTABLE	Type: INT Address: 0x0307

IQ	Type: INT Address: 0x0230
IQREF	Type: INT Address: 0x022F
KFFA	Type: INT Address: 0x026E
KII	Type: INT Address: 0x0273
KISPDEST	Type: INT Address: 0x095B
KPI	Type: INT Address: 0x0271
KPSPDEST	Type: INT Address: 0x095C
LEVEL_AD5	Type: INT Address: 0x086F
LS_ACTIVE	Type: INT Address: 0x0832
MACOMMAND	Type: ULONG Address: 0x02F2
MASTERID	Type: INT Address: 0x0927
MASTERRES	Type: LONG Address: 0x081A
MECRESL	Type: LONG Address: 0x024E
MER	Type: UINT Address: 0x08FC
MER_MASK	Type: UINT Address: 0x0965
MPOS0	Type: LONG Address: 0x025E
MREF	Type: LONG Address: 0x02AA
MSPD	Type: INT Address: 0x0820

MTSTYPE	Type: INT Address: 0x028C
NLINES	Type: ULONG Address: 0x0984
NLINESTAN	Type: ULONG Address: 0x0984
OSR	Type: UINT Address: 0x0302
PCR	Type: UINT Address: 0x0303
PHASEADV	Type: INT Address: 0x0257
POS0	Type: LONG Address: 0x02B8
POSERR	Type: INT Address: 0x022A
POSINIT	Type: ULONG Address: 0x02F2
POSOKLIM	Type: UINT Address: 0x036A
POSTRIGG1	Type: LONG Address: 0x091A
POSTRIGG2	Type: LONG Address: 0x091C
POSTRIGG3	Type: LONG Address: 0x091E
POSTRIGG4	Type: LONG Address: 0x0920
PVTBUFBEGIN	Type: INT Address: 0x0864
PVTBUFLEN	Type: INT Address: 0x0865
PVTMODE	Type: UINT Address: 0x086B
PVTPOS0	Type: LONG Address: 0x0869

PVTSENOFF	Type: INT Address: 0x092B
PVTSTS	Type: INT Address: 0x0863
PWMPER	Type: UINT Address: 0x0252
REFTST	Type: INT Address: 0x0281
REFTST_A	Type: INT Address: 0x0281
REFTST_V	Type: INT Address: 0x0281
RESRATIOX	Type: ULONG Address: 0x0880
RESRATIOY	Type: ULONG Address: 0x0882
RESRATIOZ	Type: ULONG Address: 0x0884
RINCTST	Type: INT Address: 0x0280
RINCTST_A	Type: INT Address: 0x0280
RINSTST_V	Type: INT Address: 0x0280
RPOS	Type: FIXED Address: 0x02BA
RTIME	Type: LONG Address: 0x02C2
SATPWM	Type: INT Address: 0x0254
SCR	Type: UINT Address: 0x0300
SEGBUFBEGIN	Type: ULONG Address: 0x0864
SEGBUFLEN	Type: ULONG Address: 0x0865

SEGBUFSTS	Type: ULONG Address: 0x0711
SERRMAX	Type: INT Address: 0x0879
SFI2T_D	Type: INT Address: 0x098C
SFI2T_M	Type: INT Address: 0x0819
SFTADIN	Type: INT Address: 0x025D
SFTKII	Type: INT Address: 0x0274
SFTKPI	Type: INT Address: 0x0272
SLAVEID	Type: INT Address: 0x0311
SLPER	Type: INT Address: 0x0251
SRH	Type: UINT Address: 0x090F
SRHMASK	Type: UINT Address: 0x0963
SRL	Type: UINT Address: 0x090E
SRL_MASK	Type: UINT Address: 0x0962
T1MAXPROT	Type: UINT Address: 0x0298
T1ONA	Type: UINT Address: 0x0284
T1ONB	Type: UINT Address: 0x0285
T2MAXPROT	Type: UINT Address: 0x0299
TACC	Type: FIXED Address: 0x02B6

TERRMAX	Type: UINT Address: 0x02C6
THTST	Type: INT Address: 0x0282
TIMAXPROT	Type: UINT Address: 0x02C4
TIME0	Type: LONG Address: 0x02BE
TINCTST	Type: INT Address: 0x0283
TJERK	Type: LONG Address: 0x08D1
TMLINPER	Type: UINT Address: 0x0983
TONPOSOK	Type: UINT Address: 0x036B
TPOS	Type: LONG Address: 0x02B2
TREF	Type: LONG Address: 0x02AE
TSERRMAX	Type: UINT Address: 0x087A
TSPD	Type: FIXED Address: 0x02B4
UMSXPOR	Type: UINT Address: 0x029A
IMINPROT	Type: UINT Address: 0x029B
UPGRADE	Type: UINT Address: 0x0857
UQREF	Type: INT Address: 0x0232

ELECTROCRAFT
PRO SERIES

The logo consists of the text 'ELECTROCRAFT' in a grey, sans-serif font at the top. Below it, the word 'PRO' is written in a large, bold, red font, and 'SERIES' is written in a smaller, grey, sans-serif font to its right. The text is integrated with a stylized circuit board design. Grey lines representing traces extend from the text, with small circles at the end of the lines, suggesting connection points or components.