



ETHERNET/IP SET UP GUIDE

Riccardo Piccinini
Mini Motor S.P.A.
FBS-EIP-EN
Firmware Revision: 5.004

Version 7, 09-2026

Table of Contents

1. DS402 Getting Started	1
1.1. What is DS402	1
1.2. The state machine	3
1.3. Enabling the motor in code	5
1.4. Profile Velocity	6
1.5. Profile Position	7
1.6. Homing	8
1.7. From here on	9
1.8. EtherNet/IP specifics: fixed assemblies and acyclic messaging	9
2. EtherNet/IP	10
2.1. CIP Objects	10
2.2. Process Data	14
3. Application Example	23
3.1. Tools	23
3.2. Overview	23
3.3. DBS Basic Application	23
3.4. The Two AOI Families	32
3.5. AOI CanOpenDS402 Explanation	33
3.6. AOI Rockwell-Like Explanation	38
3.7. Writing Parameters with Async Messaging	49
4. Micro850 Application Example	52
4.1. Class 1 on Micro800: What Is Possible and On Which CPU	52
4.2. Add the DBS as a Generic Device	52
4.3. Process Data Seen From CCW	55
4.4. Structured Text Example	56
4.5. Reading and Writing Parameters from CCW	60
4.6. Commissioning and Diagnostics	61

1. DS402 Getting Started



This chapter is a hands-on introduction to controlling a Mini Motor servo drive over a fieldbus using the **CANopen DS402** (CiA 402) device profile. It explains the protocol, walks through the drive state machine and shows, with ready-to-read code, how to enable the motor and command it in *Profile Velocity* and *Profile Position*. Once these concepts are clear, the vendor-specific Application Example that follows will be much easier to read: it is the same logic applied to a concrete PLC.

1.1. What is DS402

CiA 402 (also known as **DS402** / **DSP402**, standardised as *IEC 61800-7-201/301*) is the device profile for drives and motion control. It defines, independently from the underlying fieldbus:

- an **object dictionary**: every quantity exchanged with the drive (commands, references, state, feedback) is an object addressed by a 16-bit hexadecimal index, e.g. **6040h**;
- a **state machine** that governs when power is applied to the motor;
- a set of **modes of operation** (Profile Position, Profile Velocity, Profile Torque, Homing, ...);
- the meaning of the two central objects, **Control Word** (**6040h**) and **Status Word** (**6041h**).

The value of the profile is interoperability: the same objects and the same control logic apply on **CANopen**, **EtherCAT**, **EtherNet/IP**, **PROFINET** and **POWERLINK**. Only the transport layer changes; the code you write to talk to the motor stays the same. Modbus is the only exception — it does not use DS402 but a set of dedicated registers.

1.1.1. The master/slave dialogue

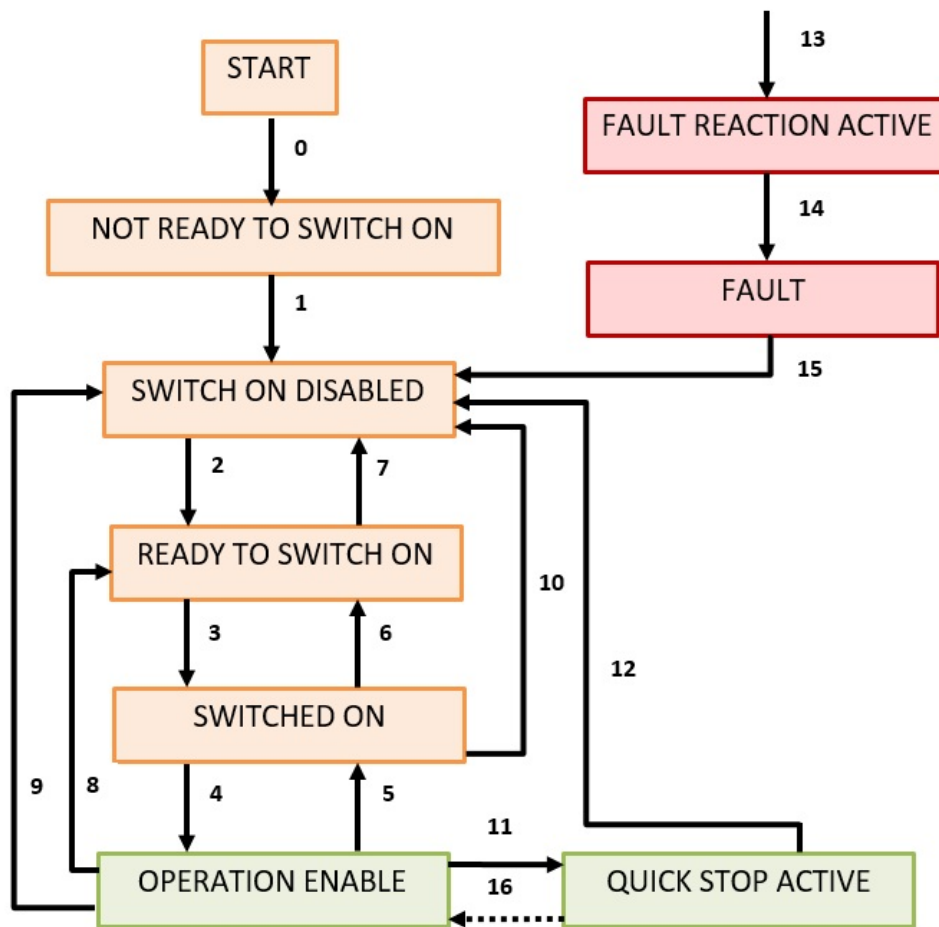
Whatever the mode, the interaction between the master (the controller) and the slave (the servo drive) always follows the same pattern:

1. The master selects the **mode of operation** by writing object *Modes of Operation* (**6060h**); the drive confirms it in *Modes of Operation Display* (**6061h**).
2. The master brings the drive to the *Operation Enabled* state by writing the **Control Word** (**6040h**), following the state machine described below.
3. The master writes the **reference** for the active mode (target velocity **60FFh**, target position **607Ah**, target torque **6071h**, ...) and, where required, triggers the motion.
4. The master monitors the drive through the **Status Word** (**6041h**) and the actual-value objects (position **6064h**, velocity **606Ch**, ...).

On cyclic fieldbuses these objects are mapped into process data (PDO) and exchanged every cycle; on CANopen they are also reachable via SDO. From the application point of view, you are always reading and writing the same objects.

1.2. The state machine

The motor produces torque only in the **Operation Enabled** state. To get there, the master walks the DS402 finite state machine by writing the Control Word and observing the Status Word.



States in **orange** have power disabled on the motor, states in **green** have power enabled, states in **red** are alarm states.

State	Meaning
Switch On Disabled	Drive ready to be prepared; parameters can be transferred, power can be enabled.
Ready To Switch On	Parameters can be transferred, power can be enabled, the motor cannot run yet.
Switched On	Same as above, one transition away from running.
Operation Enabled	No fault present, motor functions and power enabled: the motor obeys the active mode.
Quick Stop Active	The drive stopped on the emergency ramp; power still enabled, functions disabled.
Fault	A fault is active, the drive is stopped and disabled.

1.2.1. Control Word and Status Word

The transitions are driven by a few bit patterns of the **Control Word** (6040h). In practice you only need these commands:

Command	Control Word	Resulting transition
Shutdown	16#0006	to <i>Ready To Switch On</i>
Switch On	16#0007	to <i>Switched On</i>
Enable Operation	16#000F	to <i>Operation Enabled</i> (motor runs)
Disable Voltage	16#0000	to <i>Switch On Disabled</i>
Quick Stop	16#0002	to <i>Quick Stop Active</i>
Fault Reset	16#0080	clears a fault (acts on the rising edge of bit 7)

The current state is read back from the **Status Word** (6041h). By masking the low bits you can tell exactly where the drive is:

Status Word (binary)	State
xxxx xxxx x1xx 0000	Switch On Disabled
xxxx xxxx x01x 0001	Ready To Switch On
xxxx xxxx x01x 0011	Switched On
xxxx xxxx x01x 0111	Operation Enabled
xxxx xxxx x00x 0111	Quick Stop Active
xxxx xxxx x0xx 1000	Fault

Two more Status Word bits are used constantly in the examples below:

- **bit 10 – Target Reached:** set when the drive has reached the commanded position/velocity.
- **bit 12 – Set Point Acknowledge** (Profile Position): the drive acknowledges that it has latched a new position setpoint.

1.3. Enabling the motor in code

The following Structured Text (IEC 61131-3) snippets are vendor-neutral: they use `M1_Cw` for the Control Word written to the motor, `M1_Sw` for the Status Word read from it, `M1_Mode0f0p` / `M1_Mode0f0pDisplay` for objects `6060h` / `6061h`, and the target/actual objects by name. Map these to the tags of your own PLC.

The recommended pattern is a small state machine that reproduces the DS402 transitions. Every cycle it looks at the Status Word and issues the next Control Word until *Operation Enabled* is reached:

```
(* --- Bring the drive to Operation Enabled --- *)
IF (M1_Sw AND 16#004F) = 16#0008 THEN
    (* Fault: request a fault reset on the rising edge of bit 7 *)
    M1_Cw := 16#0080;
ELSIF (M1_Sw AND 16#004F) = 16#0040 THEN
    (* Switch On Disabled -> Ready To Switch On *)
    M1_Cw := 16#0006;
ELSIF (M1_Sw AND 16#006F) = 16#0021 THEN
    (* Ready To Switch On -> Switched On *)
    M1_Cw := 16#0007;
ELSIF (M1_Sw AND 16#006F) = 16#0023 THEN
    (* Switched On -> Operation Enabled *)
    M1_Cw := 16#000F;
END_IF;
```

The masks (`16#004F`, `16#006F`) isolate the state bits of the Status Word so each pattern matches exactly one state; the constant on the right is the state you are leaving. Once the Status Word settles on `...x01x 0111` (Operation Enabled) the motor obeys the active mode.



The **Fault Reset** command (`16#0080`) acts on the **rising edge** of bit 7. Make sure the Control Word returns to a value with bit 7 low before requesting another reset, otherwise the edge is lost.

1.4. Profile Velocity

In **Profile Velocity** (mode 3) the drive keeps a target speed, reaching it through the configured acceleration/deceleration ramps. The reference is *Target Velocity* (60FFh), in rpm.

The logic is:

1. select the mode (`M1_ModeOfOp := 3`) and wait for the drive to confirm it (`M1_ModeOfOpDisplay = 3`);
2. write the target velocity;
3. drive the state machine to Operation Enabled (`16#000F`) to make the motor run.

```
(* --- Profile Velocity --- *)
M1_ModeOfOp := 3;                (* 6060h: Profile Velocity *)
M1_TargetVelocity := GUI_TargetRpm; (* 60FFh: rpm, signed = direction *)

IF (M1_ModeOfOpDisplay = 3) THEN
    M1_Cw := 16#000F;             (* Enable Operation: the motor runs *)
ELSE
    M1_Cw := 16#0006;             (* mode not confirmed yet: stay ready *)
END_IF;
```

The sign of `M1_TargetVelocity` sets the direction. Writing a new value changes the speed on the fly; there is no start trigger to send, unlike Profile Position. To stop, either write 0 or drop the Control Word back to `16#0006` (Shutdown).

1.5. Profile Position

In **Profile Position** (mode 1) the drive builds a trapezoidal trajectory from the current position to *Target Position* (607Ah), using *Profile Velocity* (6081h) and the acceleration/deceleration objects. Unlike Profile Velocity, a target only becomes active when the master toggles the **New Setpoint** bit (bit 4) of the Control Word; the drive confirms it with **Set Point Acknowledge** (bit 12 of the Status Word). This handshake lets you load a new target at any time without the drive picking it up prematurely.

The sequence for a single move is:

1. select the mode (M1_ModeOfOp := 1) and wait for confirmation (M1_ModeOfOpDisplay = 1);
2. write target position and profile velocity;
3. hold the drive in Operation Enabled (16#000F);
4. pulse bit 4 (16#001F) to latch the target — the drive raises Status Word bit 12;
5. clear bit 4 (back to 16#000F) so the handshake can be repeated for the next move.

```
(* --- Profile Position (single move with New Setpoint handshake) --- *)
M1_ModeOfOp := 1;                (* 6060h: Profile Position *)
M1_TargetPos := GUI_TargetPos;    (* 607Ah *)
M1_ProfileVel := GUI_ProfileRpm;  (* 6081h, rpm *)

IF (M1_ModeOfOpDisplay = 1) THEN
  IF NewSetReq THEN
    (* raise New Setpoint (bit 4) to latch the target *)
    M1_Cw := 16#001F;
    (* the drive acknowledges with Status Word bit 12 (Set Point Ack) *)
    IF (M1_Sw AND 16#1000) = 16#1000 THEN
      M1_Cw := 16#000F;          (* drop bit 4: handshake done *)
      NewSetReq := FALSE;
    END_IF;
  ELSE
    M1_Cw := 16#000F;           (* Operation Enabled, no new target *)
  END_IF;
ELSE
  M1_Cw := 16#0006;
END_IF;

(* the move is complete when Target Reached (bit 10) is set *)
Arrived := (M1_Sw AND 16#0400) = 16#0400;
```

A few notes on the handshake:

- Set **NewSetReq** (from your logic or the HMI) whenever you want a new positioning to start; the code above turns it into the required bit-4 pulse.
- **Set Point Acknowledge** (bit 12) tells you the drive latched the target. Only then is it safe to clear bit 4.
- **Target Reached** (bit 10) tells you the motion is finished.
- Bit 6 of the Control Word selects **absolute** (0) or **relative** (1) positioning; the examples use absolute targets.

For continuous or alternating moves (e.g. shuttling between two positions), keep bit 4 low between moves and pulse it again for each new target, waiting for bit 12 each time.

1.6. Homing

Homing (mode 6) establishes the absolute position reference of the axis. The drive runs the procedure selected by *Homing Method* (6098h); the search speeds and acceleration are set by 6099h and 609Ah. The Mini Motor implementation supports several methods (limit/home switches, index, and the simple "set current position as zero"); see the Homing chapter of the product manual for the full list.

Like Profile Position, homing is triggered by the **New Setpoint / Homing Start** bit (bit 4) of the Control Word, but here the completion is reported by **Homing Done** – Status Word **bit 10** – while a failed homing is flagged by **Homing Error** – Status Word **bit 13**.

The sequence is:

1. select the mode (M1_ModeOfOp := 6) and the homing method, then wait for confirmation (M1_ModeOfOpDisplay = 6);
2. hold the drive in Operation Enabled (16#000F);
3. pulse bit 4 (16#001F) to start homing;
4. wait for Homing Done (bit 10), then clear bit 4.

```
(* --- Homing --- *)
M1_ModeOfOp := 6;                (* 6060h: Homing *)
M1_HomingMethod := 37;           (* 6098h, e.g. 37 = set current position as zero *)

IF (M1_ModeOfOpDisplay = 6) THEN
  IF HomeReq THEN
    M1_Cw := 16#001F;             (* Homing Start (bit 4) *)
  ELSE
    M1_Cw := 16#000F;             (* Operation Enabled, idle *)
  END_IF;
ELSE
  M1_Cw := 16#0006;
END_IF;

HomingDone := (M1_Sw AND 16#0400) = 16#0400;  (* bit 10 *)
HomingError := (M1_Sw AND 16#2000) = 16#2000;  (* bit 13 *)

IF HomingDone THEN
  HomeReq := FALSE;              (* drop bit 4: homing complete *)
END_IF;
```

Method 37 (set the current position as zero) needs no motion and completes immediately; the switch/index-based methods make the motor move to search for the reference and only then raise Homing Done. Always check **Homing Error** (bit 13) so your logic can react to a homing that could not complete.

1.7. From here on

The rest of this guide applies exactly this logic to a specific controller and development environment: installing the device description, mapping the process data, and the ready-made example project. When you read the vendor code, look for the same three ingredients – the state-machine walk to *Operation Enabled*, the mode selection, and the reference plus (for Profile Position) the New Setpoint handshake.

1.8. EtherNet/IP specifics: fixed assemblies and acyclic messaging

On EtherNet/IP the DS402 objects are exchanged through **assembly instances**. Two points make it different from CANopen/EtherCAT and are worth keeping in mind while reading the code above:

- **The cyclic process data is not freely composable.** You cannot build a custom PDO object by object; you must pick one of the five **fixed assemblies** – **Default (DFL)**, **A**, **B**, **C** and **D** – described in the *Process Data* section further down. *Default* carries the standard Profile Velocity/Torque/Position/Homing data; *A* adds digital/analog inputs and fieldbus-commanded digital outputs; *B* adds the CiA 402 touch probe; *C* the manufacturer touch probe; *D* the three-axis RMS acceleration values.
- **There is no SDO channel.** Objects that are not part of the cyclic assembly cannot be reached with an SDO as on CANopen. To read or write them you use **acyclic (explicit) messaging**, described later in the application example.

2. EtherNet/IP



Note: available only for drivers equipped with EtherNet/IP optional board (---/---/EIP).

This document outlines the conformance of the EtherNet/IP and CIP stack implemented in the servodrive; the following documents apply unless otherwise stated:

- CIP Network library – Volume 1 (edition 3.3, november 2007)
- CIP Network library – Volume 2; EtherNet/IP adaptation of CIP (edition 1.4, november 2007)
- CiA DS402 v3.0.1.15

Furthermore the CiA DS402 adaptation to CIP are described in the relevant section.

2.1. CIP Objects

2.1.1. Overview

The servodrive implements the following objects of the CIP object model:

- CIP common objects
 - Identity object – 01h
 - Assembly object – 04h
 - TCP object – F5h
 - Ethernet object – F6h;
- Manufacturer specific objects
 - COM object – 0x64
 - PAR object – 0x65
 - MISC object – 0x66
 - CIA402 objects – 0x67

Furthermore the following ASSEMBLIES contain process data:

- O>>T: Assembly 0x64

- T>>O: Assembly 0x65
- CFG: Assembly 0x66

2.1.2. COM object - 0x64

Instance	Attribute ID	Name	Data Type	Unit meas.	Access
0x0	0x01	Revision	UINT		RO
0x1	0x01	Software edition	UINT		RO
	0x02	Software CRC	UDINT		RO
	0x10	Save parameters	UDINT		RW



Saving parameters to non-volatile memory

Writing an attribute of the PAR object only alters the RAM image of the parameter: at the next power-up the drive reloads the values stored previously.

To make a change permanent, perform a 32 bit write of the value **65766173h** (the ASCII string "SAVE") to the *Save parameters* attribute of the COM object: class **0x64**, instance **0x1**, attribute **0x10**.

2.1.3. PAR object - 0x65

Instance	Attribute ID	Name	Data Type	Unit meas.	Access
0x0	0x01	Revision	UINT		RO
0x1	0x00–0xFF	P000–P255	UINT		RW
0x2	0x00–0xFF	P256–P511	UINT		RW

Special parameters All 32 Bit parameter are divided in two 16 bits parameters of contiguous numbering for the most significant and least significant bits.

These parameters are:

- P170 - Ip Addr
- P172 - Ip Mask
- P174 - Ip Gateway
- P120-135 - Multipos Ref 1 - 8

2.1.4. Misc objects - 0x66

Instance	Attribute ID	Name	Data Type	Unit meas.	Access
0x0	0x01	Revision	UINT		RO

Instance	Attribute ID	Name	Data Type	Unit meas.	Access
0x30 (48)	0x00	Analog input 0	UINT		RO
	0x01	Digital inputs	UINT		RO
	0x02	Digital outputs	UINT		RW
	0x03	Heatsink temperature	UINT	0.1 °C	RO
	0x05	Board temperature	UINT	0.1 °C	RO
	0x08	Actual Alarm	UINT	N/A	RO
	0x10	Dclink voltage	UINT	0.1 V	RO
	0x11	Current actual value	UINT	mA	RO
	0x12	Current limit for PP/PV/PT	UINT	mA	RO

2.1.5. CiA 402 Objects - 0x67

Instance	Attr. ID	Name	Data Type	Unit meas.	Access
0x0	0x01	Revision	UINT		RO
0x60 (96)	0x40	Control word	UINT		RW
	0x41	Status word	UINT		RO
	0x5A	Quick-stop option code	INT		RW
	0x5C	Disable operation option code	INT		RW
	0x5E	Fault reaction code	INT		RW
	0x60	Mode of operation	SINT		RW
	0x61	Mode of operation display	SINT		RO
	0x64	Position actual value	DINT	pulses	RO
	0x65	Following error window	DINT	pulses	RW
	0x66	Following error time	INT	ms	RW
	0x67	Position window	DINT	pulses	RW
	0x68	Position window time	INT	ms	RW
	0x6C	Velocity actual value	DINT	rpm	RO
	0x6D	Velocity window	DINT	rpm	RW
	0x6E	Velocity window time	INT	ms	RW
	0x6F	Velocity threshold	DINT	rpm	RW
	0x70	Velocity threshold time	INT	ms	RW
	0x71	Target torque	INT	thousand-th of rated	RW

Instance	Attr. ID	Name	Data Type	Unit meas.	Access
	0x72	Max torque	INT	thousand-th of rated	RW
	0x73	Max current	INT	thousand-th of rated	RW
	0x75	Motor rated current	DINT	milliA	RO
	0x76	Motor rated torque	DINT	milliNm	RO
	0x77	Torque actual value	INT	thousand-th of rated	RO
	0x78	Current actual value	INT	thousand-th of rated	RO
	0x7A	Target position	DINT	pulses	RW
	0x7B	Position Range index1: min limit index2: max limit	ARRAY[1..2] OF DINT	pulses	RW
	0x7C	Home offset	DINT	pulses	RW
	0x7D	Position limits index1: min limit index2: max limit	ARRAY[1..2] OF DINT	pulses	RW
	0x7E	Polarity	USINT		RW
	0x80	Max motor speed	DINT	rpm	RO
	0x81	Profile velocity	DINT	rpm	RW
	0x83	Profile acceleration	DINT	rpm/s	RW
	0x84	Profile deceleration	DINT	rpm/s	RW
	0x85	Quick stop deceleration	DINT	rpm/s	RW
	0x87	Torque slope	INT	thousandth of rated / s	RW
	0x98	Homing method	SINT		RW
	0x99	Homing speed index1: switch speed index2: index speed	ARRAY[1..2] OF DINT	rpm	RW
	0x9A	Homing acceleration	DINT	rpm/s	RW
	0xF2	Profile Option Code	INT		RW
	0xFF	Target velocity	DINT	rpm	RW

2.2. Process Data

_Valid from Firmware ≥ v4.036 _ There are four possible configurations of the EtherNet/IP process data.

- **Default (DFL):** the standard process data implementing the profile velocity/torque/position/homing
- **A:** Standard process data with the possibility to read Digital and Analog inputs and command via fieldbus the Digital Out. The latter needs the parameter P015 on 10-Fieldbus to receive commands.

- **B:** A process data with the CiA 402 Touch probe functionality
- **C:** A process data with our manufacturer custom Touch Probe implementation
- **D:** A process data with acceleration three axis rms values

You must select the corresponding device description and set the parameter P176 to the desired value.

2.2.1. Layout Default (DFL)

The configuration for process data is the following:

Assembly 0x64 (Originator to Target, DBS/FC inputs - PLC outputs)

Assembly Length : 50 Bytes

Offset	Type	Attribute	Class	Instance	Attr. ID	Default	Meaning
0x00	U16	ControlWord	0x67	0x60	0x40	0000h	see DSP402
0x02	S32	TargetPosition	0x67	0x60	0x7A	+0	position units
0x06	U16	TorqueLimit	0x67	0x60	0x72	1000	× 0.1% Trq
0x08	S32	TargetVelocity	0x67	0x60	0xFF	+0	rpm
0x0C	S16	TargetTorque	0x67	0x60	0x71	+0	× 0.1% Trq
0x0E	U8	ModeOfOperation	0x67	0x60	0x60	1	Profile ^[1]
0x0F	U8	Padding Byte					
0x10	U32	ProfileVelocity	0x67	0x60	0x81	+3000	rpm
0x14	U32	ProfileAccel	0x67	0x60	0x83	+3000	rpm/s
0x18	U32	ProfileDecel	0x67	0x60	0x84	+3000	rpm/s
0x1C	S32	PositionLimit.Neg	0x67	0x60	0x7D.1	-2147483648	position units
0x20	S32	PositionLimit.Pos	0x67	0x60	0x7D.2	+2147483647	position units
0x24	S32	HomingOffset	0x67	0x60	0x7C	+0	position units
0x28	S8	HomingMethod	0x67	0x60	0x98	37	SetQuota ^[1]

Assembly 0x65 (Target to Originator, DBS/FC outputs - PLC inputs)

Assembly Length : 20 Bytes

Offset	Type	Attribute	Class	Instance	Attr. ID	Default	Meaning
0x00	U16	StatusWord	0x67	0x60	0x41	0250h	Motor Status ^[1]

Offset	Type	Attribute	Class	Instance	Attr. ID	Default	Meaning
0x02	S32	PositionActualValue	0x67	0x60	0x64	+0	position units
0x06	S16	TorqueActualValue	0x67	0x60	0x77	+0	× 0.1% Trq
0x08	S32	VelocityActualValue	0x67	0x60	0x6C	+0	rpm
0x0C	U16	VDclink	0x67	0x30	0x10	+0	× 0.1 V
0x0E	U8	ModeOfOperationDisplay	0x67	0x60	0x61	1	Profile ^[1]
0x0F	U8	Padding Byte					
0x10	U16	ActualAlarm	0x67	0x30	0x08	0	see DBS Manual

2.2.2. Layout A

The configuration for process data is the following:

Assembly 0x64 (Originator to Target, DBS/FC inputs - PLC outputs)

Assembly Length : 50 Bytes

Offset	Type	Attribute	Class	Instance	Attr. ID	Default	Meaning
0x00	U16	ControlWord	0x67	0x60	0x40	0000h	see DSP402
0x02	S32	TargetPosition	0x67	0x60	0x7A	+0	position units
0x06	U16	TorqueLimit	0x67	0x60	0x72	1000	× 0.1% Trq
0x08	S32	TargetVelocity	0x67	0x60	0xFF	+0	rpm
0x0C	S16	TargetTorque	0x67	0x60	0x71	+0	× 0.1% Trq
0x0E	U8	ModeOfOperation	0x67	0x60	0x60	1	Profile ^[1]
0x0F	U8	Padding Byte					
0x10	U32	ProfileVelocity	0x67	0x60	0x81	+3000	rpm
0x14	U32	ProfileAccel	0x67	0x60	0x83	+3000	rpm/s
0x18	U32	ProfileDecel	0x67	0x60	0x84	+3000	rpm/s
0x1C	S32	PositionLimit.Neg	0x67	0x60	0x7D.1	-2147483648	position units
0x20	S32	PositionLimit.Pos	0x67	0x60	0x7D.2	+2147483647	position units
0x24	S32	HomingOffset	0x67	0x60	0x7C	+0	position units
0x28	S8	HomingMethod	0x67	0x60	0x98	37	SetQuota ^[1]

Offset	Type	Attribute	Class	Instance	Attr. ID	Default	Meaning
0x29	U8	Padding Byte					
0x2A	U16	Digital output	0x67	0x30	0x02	0	bit0: DOUT1 bit1..15: not used ^[2]

Assembly 0x65 (Target to Originator, DBS/FC outputs - PLC inputs)

Assembly Length : 30 Bytes

Offset	Type	Attribute	Class	Instance	Attr. ID	Default	Meaning
0x00	U16	StatusWord	0x67	0x60	0x41	0250h	Motor Status ^[1]
0x02	S32	PositionActualValue	0x67	0x60	0x64	+0	position units
0x06	S16	TorqueActualValue	0x67	0x60	0x77	+0	× 0.1% Trq
0x08	S32	VelocityActualValue	0x67	0x60	0x6C	+0	rpm
0x0C	U16	VDclink	0x67	0x30	0x10	+0	× 0.1 V
0x0E	U8	ModeOfOperationDisplay	0x67	0x60	0x61	1	Profile ^[1]
0x0F	U8	Padding Byte					
0x10	U16	ActualAlarm	0x67	0x30	0x08	0	see DBS Manual
0x12	S16	Analog Input	0x67	0x30	0x00	0	-32768 ... +32767
0x14	U16	Digital Input	0x67	0x30	0x01	0	Bit0: DIN1 Bit1: DIN2 Bit2: DIN3 Bit3: DIN4 Bit4: DIN5 Bit5..15: not used

2.2.3. Layout B

The configuration for process data is the following:

Assembly 0x64 (Originator to Target, DBS/FC inputs - PLC outputs)

Assembly Length : 50 Bytes

Offset	Type	Attribute	Class	Instance	Attr. ID	Default	Meaning
0x00	U16	ControlWord	0x67	0x60	0x40	0000h	see DSP402
0x02	S32	TargetPosition	0x67	0x60	0x7A	+0	position units
0x06	U16	TorqueLimit	0x67	0x60	0x72	1000	× 0.1% Trq
0x08	S32	TargetVelocity	0x67	0x60	0xFF	+0	rpm
0x0C	S16	TargetTorque	0x67	0x60	0x71	+0	× 0.1% Trq

Offset	Type	Attribute	Class	Instance	Attr. ID	Default	Meaning
0x0E	U8	ModeOfOperation	0x67	0x60	0x60	1	Profile ^[1]
0x0F	U8	Padding Byte					
0x10	U32	ProfileVelocity	0x67	0x60	0x81	+3000	rpm
0x14	U32	ProfileAccel	0x67	0x60	0x83	+3000	rpm/s
0x18	U32	ProfileDecel	0x67	0x60	0x84	+3000	rpm/s
0x1C	S32	PositionLimit.Neg	0x67	0x60	0x7D.1	-2147483648	position units
0x20	S32	PositionLimit.Pos	0x67	0x60	0x7D.2	+2147483647	position units
0x24	S32	HomingOffset	0x67	0x60	0x7C	+0	position units
0x28	S8	HomingMethod	0x67	0x60	0x98	37	SetQuota ^[1]
0x29	U8	Padding Byte					
0x2A	U16	Digital output	0x67	0x30	0x02	0	bit0: DOUT1 bit1..15: not used ^[2]
0x2C	U16	Touch Probe Function	0x67	0x60	0xB8	0	^[1]

Assembly 0x65 (Target to Originator, DBS/FC outputs - PLC inputs)

Assembly Length : 40 Bytes

Offset	Type	Attribute	Class	Instance	Attr. ID	Default	Meaning
0x00	U16	StatusWord	0x67	0x60	0x41	0250h	Motor Status ^[1]
0x02	S32	PositionActualValue	0x67	0x60	0x64	+0	position units
0x06	S16	TorqueActualValue	0x67	0x60	0x77	+0	× 0.1% Trq
0x08	S32	VelocityActualValue	0x67	0x60	0x6C	+0	rpm
0x0C	U16	VDclink	0x67	0x30	0x10	+0	× 0.1 V
0x0E	U8	ModeOfOperationDisplay	0x67	0x60	0x61	1	Profile ^[1]
0x0F	U8	Padding Byte					
0x10	U16	ActualAlarm	0x67	0x30	0x08	0	see DBS Manual
0x12	S16	Analog Input	0x67	0x30	0x00	0	-32768 ... +32767

Offset	Type	Attribute	Class	Instance	Attr. ID	Default	Meaning
0x14	U16	Digital Input	0x67	0x30	0x01	0	Bit0: DIN1 Bit1: DIN2 Bit2: DIN3 Bit3: DIN4 Bit4: DIN5 Bit5..15: not used
0x16	U16	Touch Probe Status	0x67	0x60	0xB9	0	[1]
0x18	S32	Touch Probe Rising Edge position	0x67	0x60	0xBA	0	position units
0x1C	S32	Touch Probe Falling Edge position	0x67	0x60	0xBB	0	position units

2.2.4. Layout C

The configuration for process data is the following:

Assembly 0x64 (Originator to Target, DBS/FC inputs - PLC outputs)

Assembly Length : 60 Bytes

Offset	Type	Attribute	Class	Instance	Attr. ID	Default	Meaning
0x00	U16	ControlWord	0x67	0x60	0x40	0000h	see DSP402
0x02	S32	TargetPosition	0x67	0x60	0x7A	+0	position units
0x06	U16	TorqueLimit	0x67	0x60	0x72	1000	× 0.1% Trq
0x08	S32	TargetVelocity	0x67	0x60	0xFF	+0	rpm
0x0C	S16	TargetTorque	0x67	0x60	0x71	+0	× 0.1% Trq
0x0E	U8	ModeOfOperation	0x67	0x60	0x60	1	Profile [1]
0x0F	U8	Padding Byte					
0x10	U32	ProfileVelocity	0x67	0x60	0x81	+3000	rpm
0x14	U32	ProfileAccel	0x67	0x60	0x83	+3000	rpm/s
0x18	U32	ProfileDecel	0x67	0x60	0x84	+3000	rpm/s
0x1C	S32	PositionLimit.Neg	0x67	0x60	0x7D.1	-2147483648	position units
0x20	S32	PositionLimit.Pos	0x67	0x60	0x7D.2	+2147483647	position units
0x24	S32	HomingOffset	0x67	0x60	0x7C	+0	position units

Offset	Type	Attribute	Class	Instance	Attr. ID	Default	Meaning
0x28	S8	HomingMethod	0x67	0x60	0x98	37	SetQuota ^[1]
0x29	U8	Padding Byte					
0x2A	U16	Digital output	0x67	0x30	0x02	0	bit0: DOUT1 bit1..15: not used ^[2]
0x2C	S32	Touch Probe positioning offset	0x67	0x30	0x20	0	Position Units
0x2C	S32	Touch Probe positioning modulo	0x67	0x30	0x21	0	Position Units

Assembly 0x65 (Target to Originator, DBS/FC outputs - PLC inputs)

Assembly Length : 30 Bytes

Offset	Type	Attribute	Class	Instance	Attr. ID	Default	Meaning
0x00	U16	StatusWord	0x67	0x60	0x41	0250h	Motor Status ^[1]
0x02	S32	PositionActualValue	0x67	0x60	0x64	+0	position units
0x06	S16	TorqueActualValue	0x67	0x60	0x77	+0	× 0.1% Trq
0x08	S32	VelocityActualValue	0x67	0x60	0x6C	+0	rpm
0x0C	U16	VDclink	0x67	0x30	0x10	+0	× 0.1 V
0x0E	U8	ModeOfOperationDisplay	0x67	0x60	0x61	1	Profile ^[1]
0x0F	U8	Padding Byte					
0x10	U16	ActualAlarm	0x67	0x30	0x08	0	see DBS Manual
0x12	S16	Analog Input	0x67	0x30	0x00	0	-32768 ... +32767
0x14	U16	Digital Input	0x67	0x30	0x01	0	Bit0: DIN1 Bit1: DIN2 Bit2: DIN3 Bit3: DIN4 Bit4: DIN5 Bit5..15: not used

2.2.5. Layout D

The configuration for process data is the following:

Assembly 0x64 (Originator to Target, DBS/FC inputs - PLC outputs)

Assembly Length : 50 Bytes

Offset	Type	Attribute	Class	Instance	Attr. ID	Default	Meaning
0x00	U16	ControlWord	0x67	0x60	0x40	0000h	see DSP402
0x02	S32	TargetPosition	0x67	0x60	0x7A	+0	position units
0x06	U16	TorqueLimit	0x67	0x60	0x72	1000	× 0.1% Trq
0x08	S32	TargetVelocity	0x67	0x60	0xFF	+0	rpm
0x0C	S16	TargetTorque	0x67	0x60	0x71	+0	× 0.1% Trq
0x0E	U8	ModeOfOperation	0x67	0x60	0x60	1	Profile ^[1]
0x0F	U8	Padding Byte					
0x10	U32	ProfileVelocity	0x67	0x60	0x81	+3000	rpm
0x14	U32	ProfileAccel	0x67	0x60	0x83	+3000	rpm/s
0x18	U32	ProfileDecel	0x67	0x60	0x84	+3000	rpm/s
0x1C	S32	PositionLimit.Neg	0x67	0x60	0x7D.1	-2147483648	position units
0x20	S32	PositionLimit.Pos	0x67	0x60	0x7D.2	+2147483647	position units
0x24	S32	HomingOffset	0x67	0x60	0x7C	+0	position units
0x28	S8	HomingMethod	0x67	0x60	0x98	37	SetQuota ^[1]
0x29	U8	Padding Byte					
0x2A	U16	Digital output	0x67	0x30	0x02	0	bit0: DOUT1 bit1..15: not used ^[2]

Assembly 0x65 (Target to Originator, DBS/FC outputs - PLC inputs)

Assembly Length : 30 Bytes

Offset	Type	Attribute	Class	Instance	Attr. ID	Default	Meaning
0x00	U16	StatusWord	0x67	0x60	0x41	0250h	Motor Status ^[1]
0x02	S32	PositionActualValue	0x67	0x60	0x64	+0	position units
0x06	S16	TorqueActualValue	0x67	0x60	0x77	+0	× 0.1% Trq
0x08	S32	VelocityActualValue	0x67	0x60	0x6C	+0	rpm
0x0C	U16	VDclink	0x67	0x30	0x10	+0	× 0.1 V
0x0E	U8	ModeOfOperationDisplay	0x67	0x60	0x61	1	Profile ^[1]

Offset	Type	Attribute	Class	Instance	Attr. ID	Default	Meaning
0x0F	U8	Padding Byte					
0x10	U16	ActualAlarm	0x67	0x30	0x08	0	see DBS Manual
0x12	S16	Analog Input	0x67	0x30	0x00	0	-32768 ... +32767
0x14	U16	Digital Input	0x67	0x30	0x01	0	Bit0: DIN1 Bit1: DIN2 Bit2: DIN3 Bit3: DIN4 Bit4: DIN5 Bit5..15: not used
0x16	U16	Acceleration rms	0x67	0x30	0x31		
0x19	U16	Acceleration rms	0x67	0x30	0x32		
0x1A	U16	Acceleration rms	0x67	0x30	0x33		

[1] see DSP402

[2] P015=10 required for fieldbus operation

3. Application Example

3.1. Tools

Program Example (no AOI)

Program Example (AOI, DS402)

AOI Download (DS402)

AOI + Program Download (Rockwell-like) V1.1

DBS/FC BSI Interface Software

3.2. Overview

This document shows the creation of a Mini Motor Integrated Drive application using the EtherNet/IP communication interface; a Rockwell CompactLogix controller is configured and an application is written into the Studio 5000 development environment.

References:

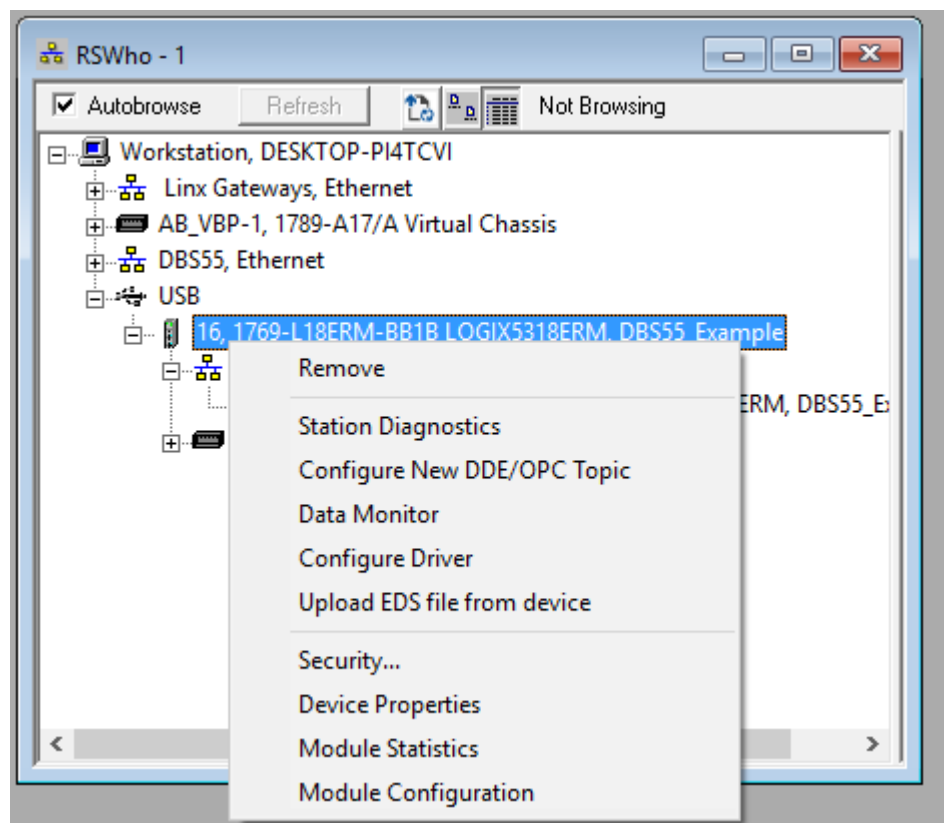
- CompactLogix L18ERM
- DBS/FC firmware version 5.004
- Studio 5000, version 31.01.00



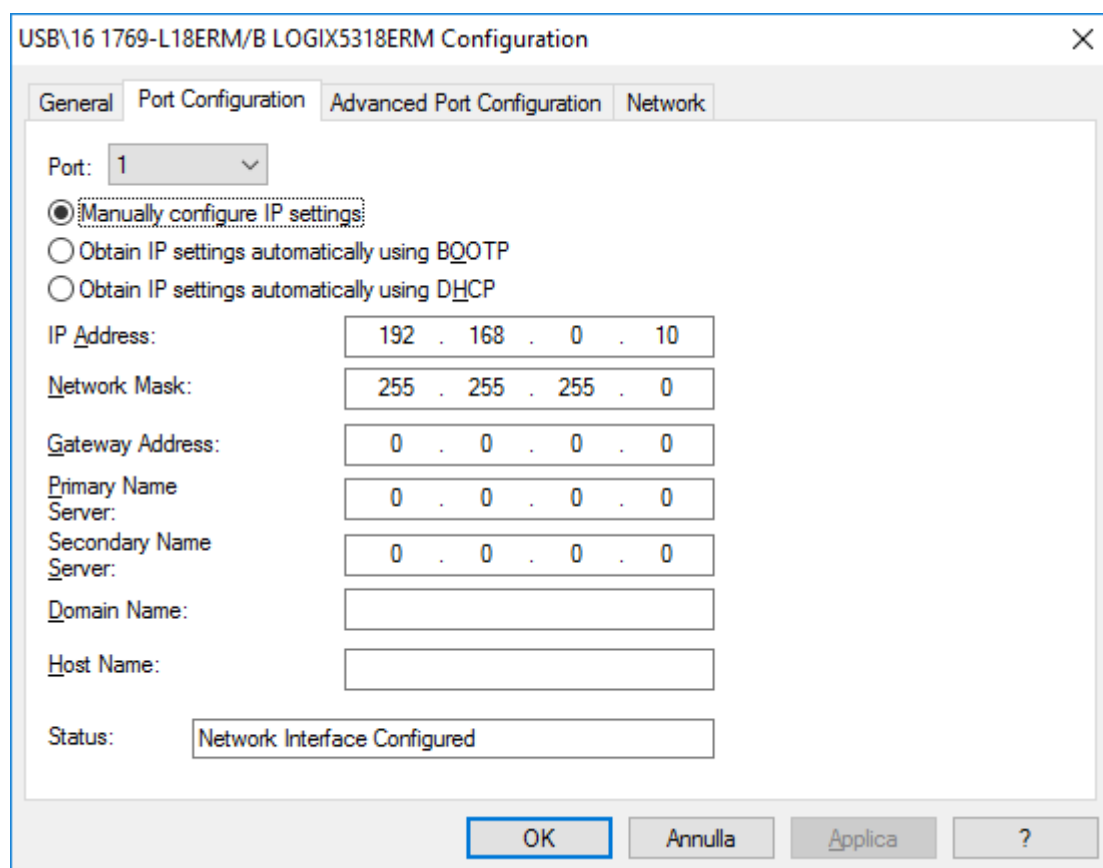
3.3. DBS Basic Application

3.3.1. Configure Controller Address

- To set the controller address, start RSLinx
- Select Communication → RSWho and open the controller node (in the example the controller is connected via USB); then right click and select *Module Configuration*



- In the configuration dialog, select *Port Configuration* (1), then *Manually configure IP settings* (2) and insert address and mask (3); in the example 192.168.0.10 is set
- Confirm with OK.



3.3.2. Configure the Motor Address

The drive leaves the factory with a **static** address, which is the one you will find on a brand new unit:

Parameter	Name and meaning	Factory value
P170	<i>IpAddr</i> – IP address of the drive	192.168.250.100
P172	<i>IpMask</i> – subnet mask	255.255.0.0
P174	<i>IpGateway</i> – default gateway	0.0.0.0
P215	<i>IpMode</i> – how the address is assigned	0 – Static

Two addressing modes are available, selected with P215.

Static (P215 = 0). Write address, mask and gateway into P170, P172 and P174 with the BSI interface software, connected to the drive through its service port. These are 32-bit parameters: BSI lets you type them as a normal dotted address, while over the fieldbus each one is exposed as two contiguous 16-bit parameters (see the PAR object table in the previous chapter). BSI stores them in non-volatile memory for you, so nothing else has to be saved: just power cycle the drive, and the new address is taken at the next start-up.

DHCP (P215 = 1). The drive asks a DHCP server for its address every time it powers up, and the content of P170, P172 and P174 is ignored. This is the mode to use if you prefer the usual Rockwell commissioning flow: with the **BOOTP/DHCP EtherNet/IP Commissioning Tool** the drive shows up in the list with its MAC address as soon as it is powered, and you can hand it the address you want.



P215 is available from firmware 4.056. On earlier firmware the drive is always static and the address can only be changed with the BSI software.



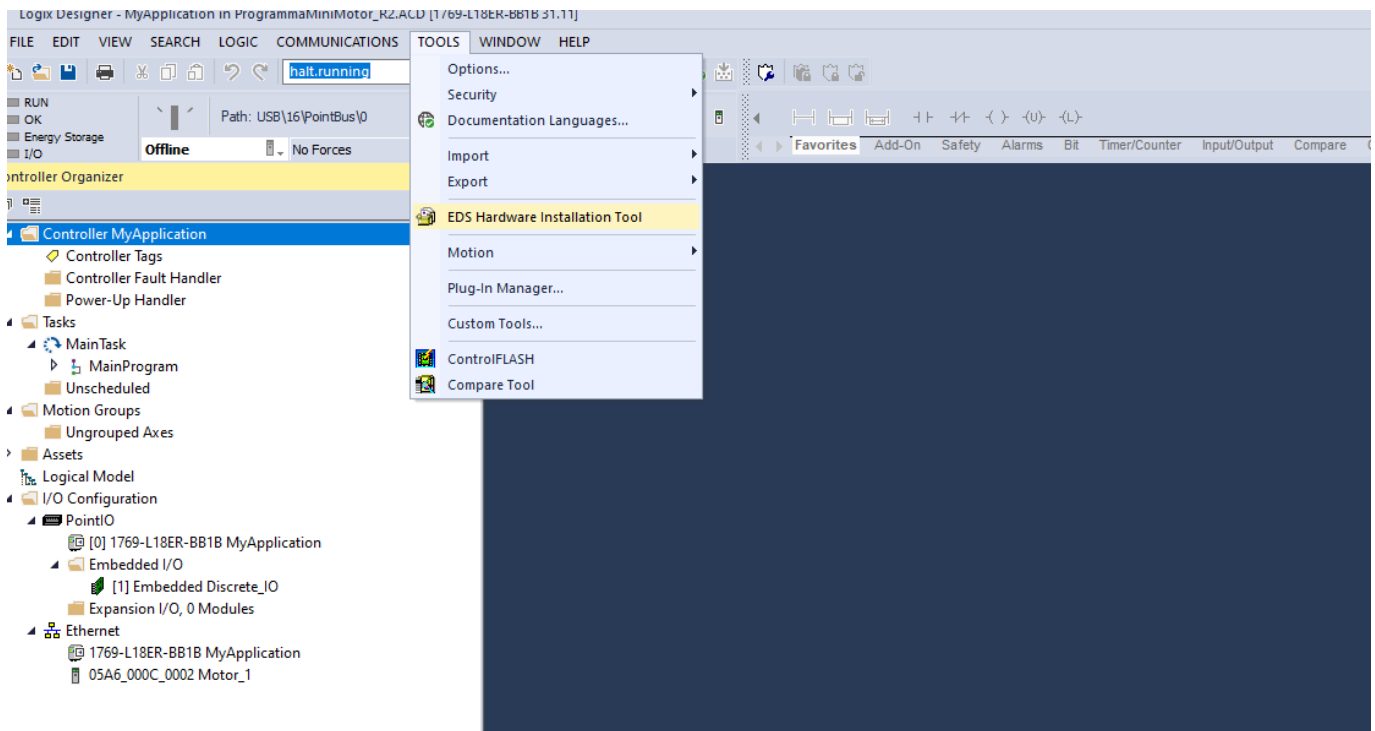
The module you are about to add in Studio 5000 points to one fixed IP address. If you leave the drive in DHCP, make sure the server always gives it the same address by reserving the lease for the MAC address of the drive; otherwise the controller will lose the connection at the next reboot. If you do not need dynamic addressing, going back to P215 = 0 with the address you like is the simplest and safest choice.



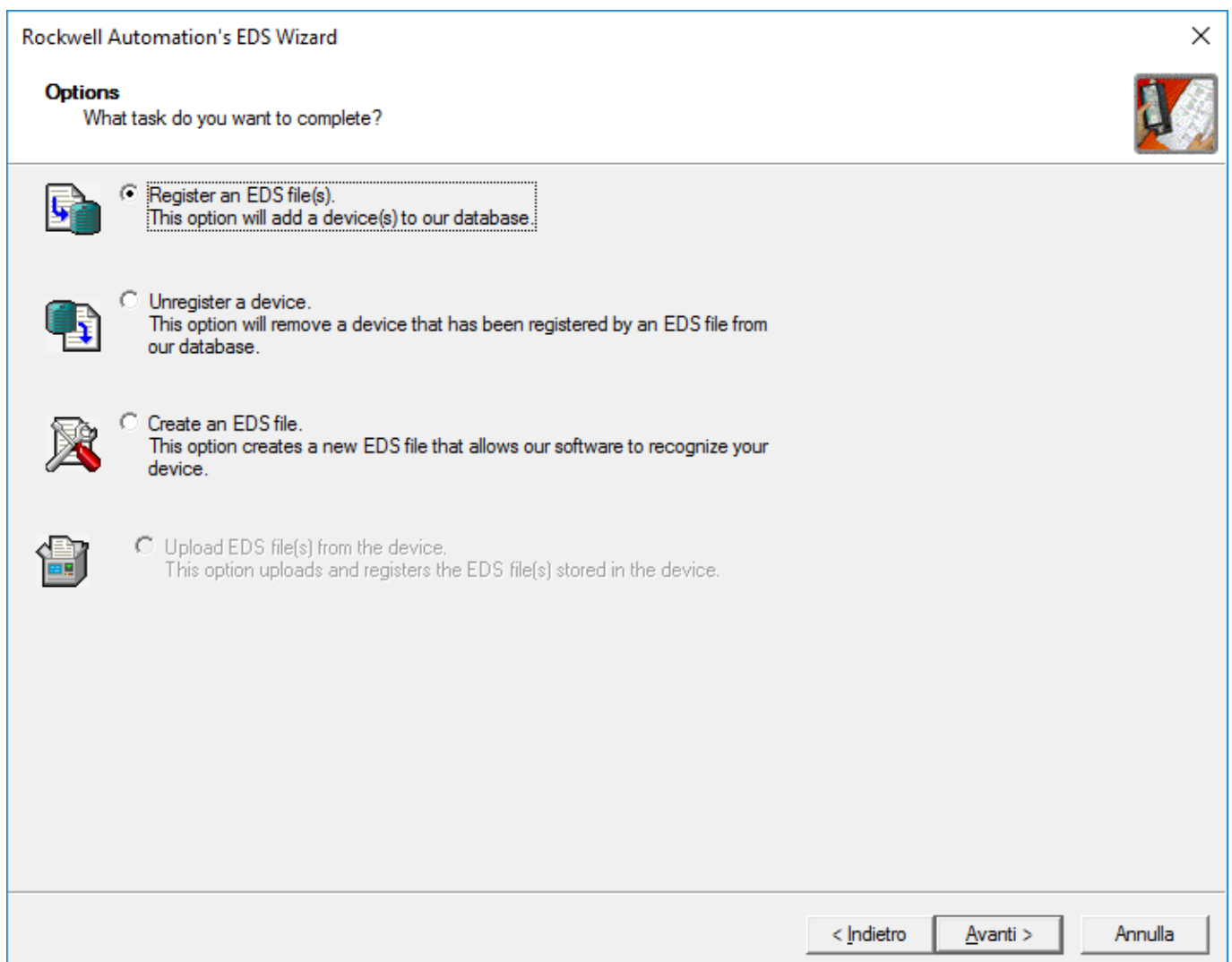
Whichever mode you use, the drive address must end up in the same subnet as the controller address set in the previous step.

3.3.3. Install EDS

- In Studio 5000 click on Tools → EDS Hardware Installation Tool



- Register an EDS file



- Browse
- Open the BSI software revision\DeviceDesc\EthernetIP folder and select the EIP layout you desire. For this example we will select DBS55_CIP_dfl.eds. Install only the one you want to use.



The EDS file must match the process data layout selected on the drive with P176. All the example programs and all the AOI shown in this guide use the **Default (DFL)** layout, so **DBS55_CIP_dfl.eds** with P176 = 0.

Condividi Visualizza

> MM-DBS_v4.041 > DeviceDesc > EthernetIP				
Nome	Ultima modifica	Tipo	Dimensione	
DBS55_CIP.eds	21/11/2022 16:31	File EDS	6 KB	
DBS55_CIP_dfl.eds	21/11/2022 16:31	File EDS	21 KB	
DBS55_CIP_ext-a.eds	21/11/2022 16:31	File EDS	23 KB	
DBS55_CIP_ext-b.eds	21/11/2022 16:31	File EDS	26 KB	
DBS55_CIP_ext-c.eds	21/11/2022 16:31	File EDS	27 KB	
DBS55_CIP_ext-d.eds	21/11/2022 16:31	File EDS	30 KB	
Readme.txt	21/11/2022 16:31	Documento di testo	1 KB	

Rockwell Automation's EDS Wizard



Registration

Electronic Data Sheet file(s) will be added to your system for use in Rockwell Automation applications.


☒ Register a single file

☐ Register a directory of EDS files

☐ Look in subfolders

Named:



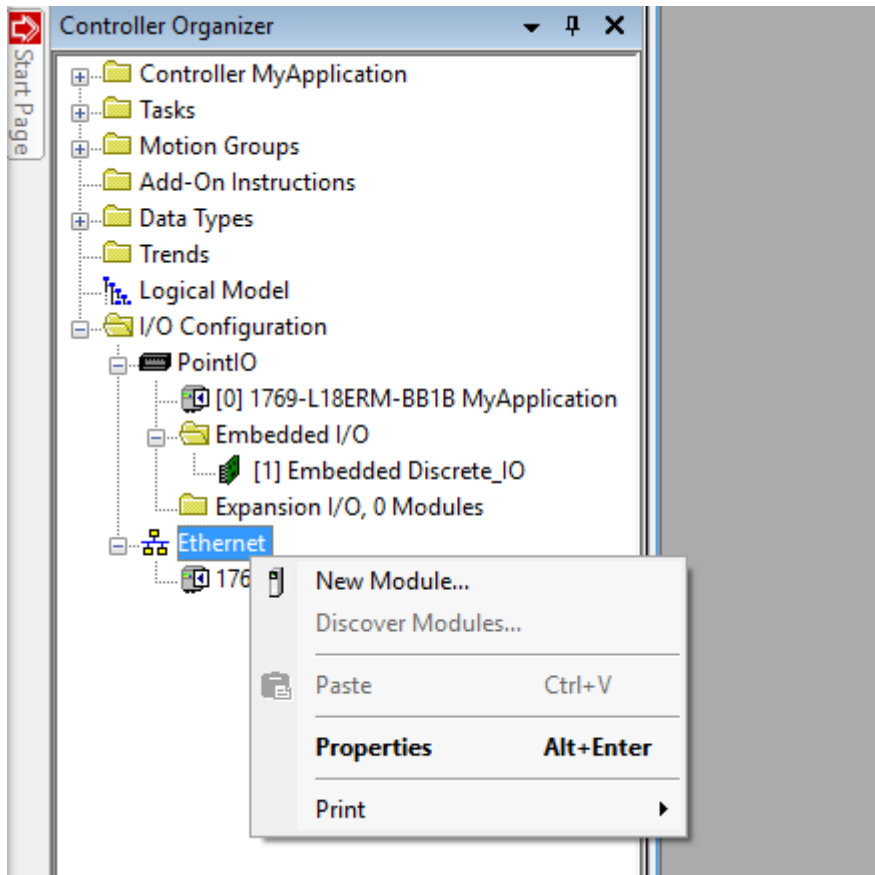
* If there is an icon file (.ico) with the same name as the file(s) you are registering then this image will be associated with the device.

To perform an installation test on the file(s), click Next

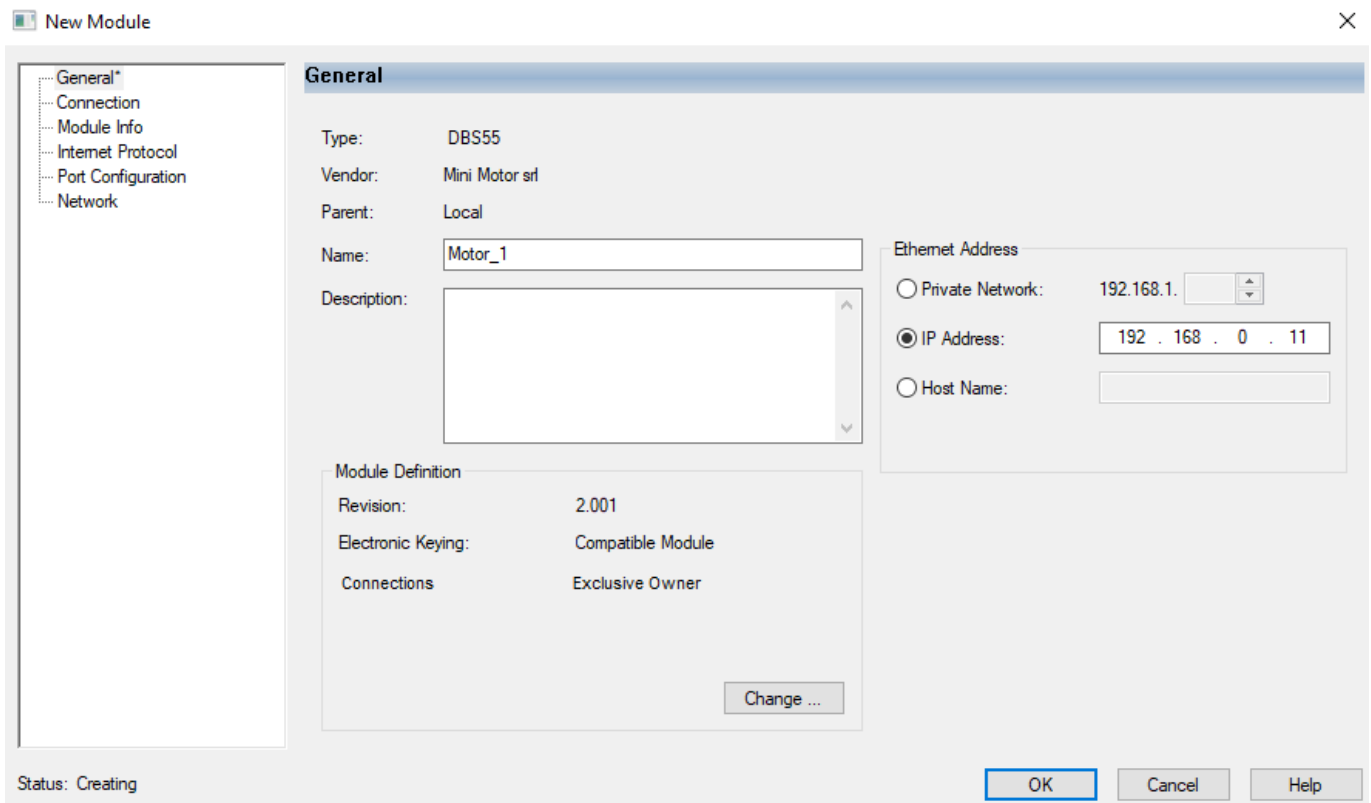
- Next →, Next →, Next →; click on Finish to complete the insertion.

3.3.4. Add DBS Slave to a Project

- Right click on *I/O Configuration > Ethernet* and select *New Module*



- Type *db*s in the filter and select *05A6_000C_0002 – DBS55*, then click on Create



New Module

General

Type: DBS55
Vendor: Mini Motor srl
Parent: Local
Name: Motor_1
Description:

Ethernet Address

☐ Private Network: 192.168.1.
☒ IP Address: 192 . 168 . 0 . 11
☐ Host Name:

Module Definition

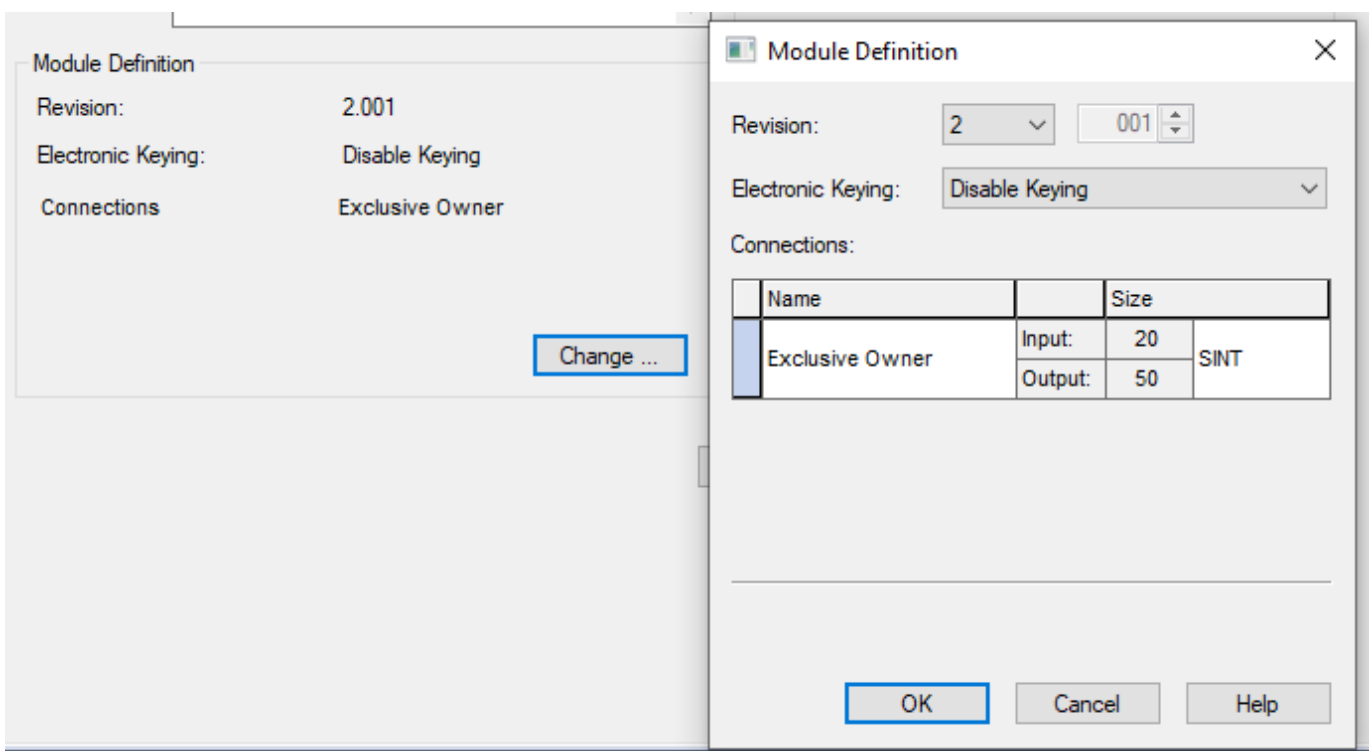
Revision: 2.001
Electronic Keying: Compatible Module
Connections: Exclusive Owner

Change ...

Status: Creating

OK Cancel Help

- In the module definition window, in the *Electronic Keying* field select *Disable Keying*
- Confirm the module definition dialog and finally click on Create, then close the dialog



Module Definition

Revision: 2 001
Electronic Keying: Disable Keying
Connections: Exclusive Owner

Change ...

Module Definition

Revision: 2 001
Electronic Keying: Disable Keying
Connections:

Name	Size
Exclusive Owner	Input: 20 Output: 50

SINT

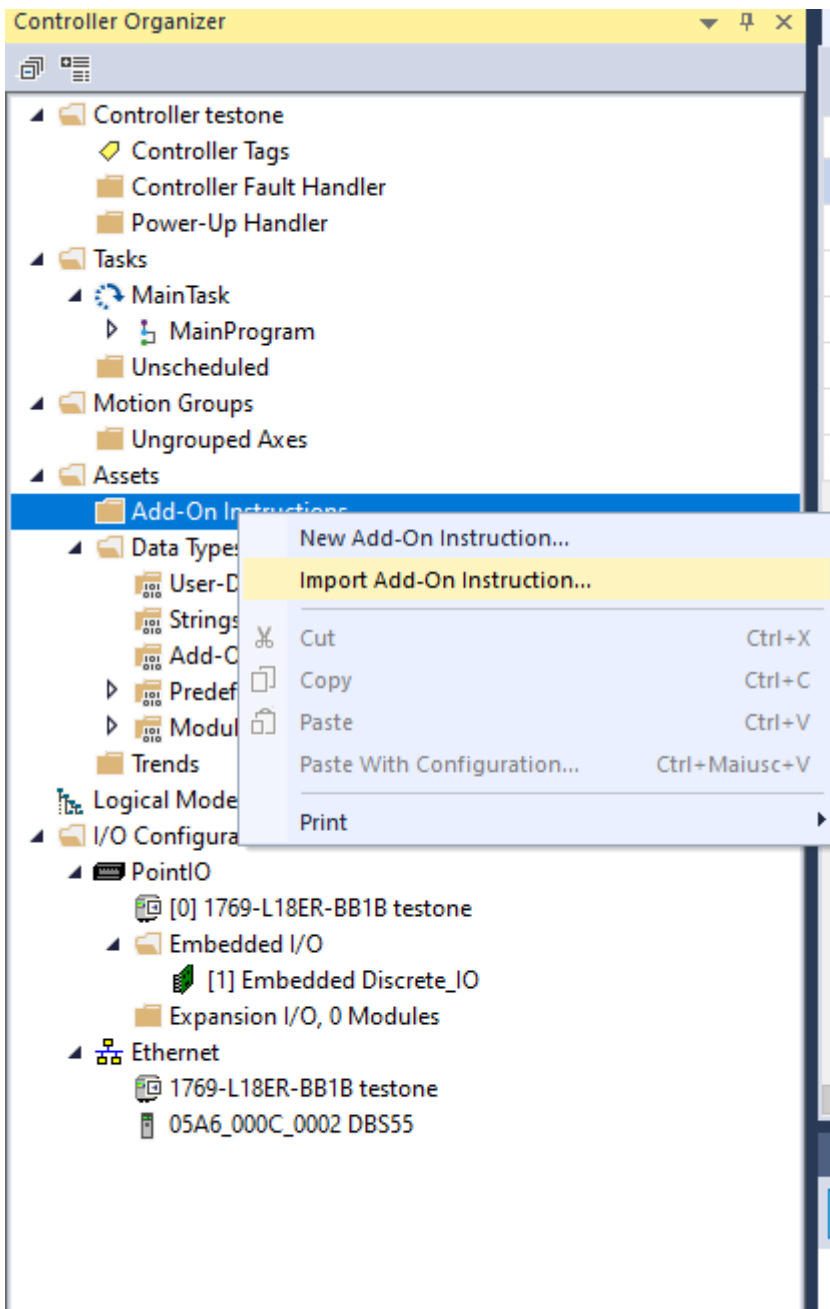
OK Cancel Help

3.3.5. AOI Import

- Import the DBS AOI you need for your project. Go to *Assets* → *Add-On Instructions* → *Right Click* → select which AOI to import.



Each **.L5X** file contains one single Add-On Instruction, so importing one block never touches the others. Both families also carry the **DBS55_In** and **DBS55_Out** definitions with them: the first import creates the two data types, the following ones find them already there.



- After having imported an AOI you should find two User Defined Data Types.
- Create for each motor a variable of both types.

Controller Tags - testone(controller) x						
Scope:  testone		Show: All Tags				
Name	Alias For	Base Tag	Data Type	Description		
▶ Local:1:O			AB:Embedded_DiscreteIO:O:0			R
▶ Local:1:I			AB:Embedded_DiscreteIO:I:0			R
▶ Local:1:C			AB:Embedded_DiscreteIO:C:0			R
▶ DBS1_In			DBS55_In	PdoRX		R
▶ DBS1_out			DBS55_Out 	PdoTX		R

3.4. The Two AOI Families

Two sets of Add-On Instructions are available, and they do the same job seen from two different angles:

- the **CanOpenDS402** set keeps the CiA 402 vocabulary – profile position, profile velocity, profile torque, homing, control word and status word;
- the **Rockwell-like** set wraps the same sequences into instructions that look and behave like the Logix motion ones (MSO, MSF, MAM, MAJ, ...), for those who are more at home with a Rockwell drive than with a CiA 402 state machine.

They are not alternatives you have to choose once and for all: both families work on the same two User Defined Data Types, **DBS55_In** (what the PLC sends to the drive) and **DBS55_Out** (what the drive returns), so they can live in the same project and even on the same axis.



All the AOI described below – both families – are built on the **Default (DFL)** process data layout. Install **DBS55_CIP_dfl.eds** and leave the drive on P176 = 0 (Default). With any other layout (A, B, C, D) the assemblies have a different length and content, the AOI will not line up with the data and the example project will not work.

Unless stated otherwise, the AOI use the units of the DS402 process data:

- speeds in rpm, accelerations and decelerations in rpm/s;
- torques in thousandths of the motor rated torque, so 1000 = 100 % of rated torque;
- positions in encoder pulses, or in the user unit if you have scaled it with P160, P161 and P162 (see the main DBS/FC manual).

3.4.1. Which Revision This Guide Describes

Every Add-On Instruction carries a revision number, visible in Studio 5000 under *Assets → Add-On Instructions* and in the properties of each instance. This guide describes the revisions in the table below: if the blocks installed on your project carry a lower number, some of the parameters and behaviours described here are not there.

Block	Revision	Notes
MM_MSO, MM_MSOF, MM_MAFR, MM_MAM, MM_MAJ, MM_MAH, MM_MAS	1.1	Rockwell-like command blocks. Revision 1.1 adds <i>Timeout</i> and <i>ErrorCode</i> and corrects the defects listed in Updating from Revision 1.0
MM_MSO_ST ... MM_MAS_ST	1.1	Optional variant of the same seven blocks, written in Structured Text. Same pins, same behaviour
MM_MAU, MM_MAUO	1.2	Rockwell-like process data blocks. No functional change from 1.1
MM_MSL	1.0	New Rockwell-like block that writes the software position limits, see MM_MSL
MM_ReadData_dfl, MM_SendData_dfl, MM_Profile_*_dfl, MM_Status_dfl	1.1	DS402 family, unchanged
MM_Reset_Error_dfl	1.0	DS402 family, unchanged



Revision 1.1 of the Rockwell-like command blocks is not a drop-in replacement for revision 1.0. Two changes break existing applications: the *Direction* input of MM_MAM, renamed *MoveType* and with its polarity corrected, and the renaming of the *Deceleration* input. Read [Updating from Revision 1.0](#) before updating a machine that is already running.

3.4.2. The Language Inside the Blocks

The Rockwell-like blocks are written in **ladder**, like the rest of the package, and that is the set to install unless you have a reason not to. It opens on any Studio 5000 seat.

A second set of the same seven blocks is available in **Structured Text**, named with an *_ST* suffix: MM_MSO_ST, MM_MAM_ST and so on. Pins, defaults, behaviour and revision are identical, only the code inside differs. The names differ as well, so the two sets can live in the same project and you can put one against the other on the same axis. MM_MAU, MM_MAUO and MM_MSL have no ST variant, being data copies with no sequence inside: the ladder ones serve both sets.

The ST variant exists because these blocks are state machines, and a **CASE** statement says what a chain of **EQU** contacts on a state tag only implies. If you maintain the code yourself, it is the easier one to read.



Structured Text is a licensed language in Studio 5000 and the lower-tier editions do not include it. On a seat without the ST licence an *_ST* block cannot be opened, verified or downloaded. For anything you hand to a customer who edits their own program, ship the ladder set.

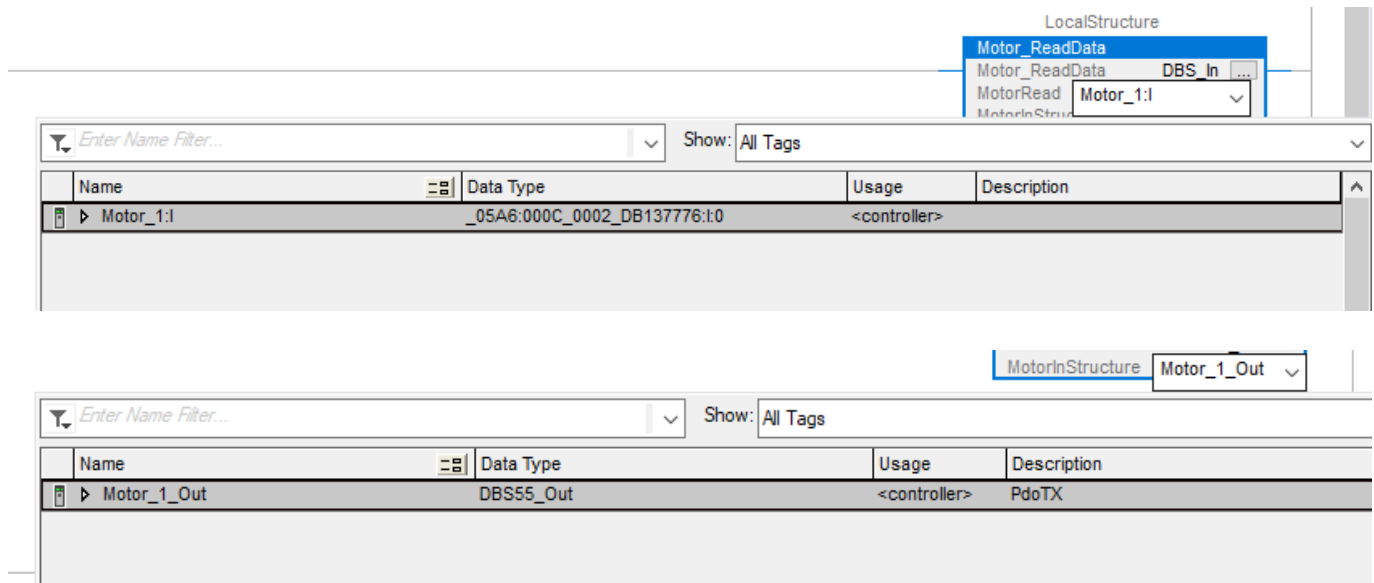
3.5. AOI CanOpenDS402 Explanation

There are 8 AOI that can be used for DBS and FC systems. We will look at them here.

3.5.1. MM_ReadData_dfl

This instruction should be the first and its job is to take the motor assembly data and assign it to a **DBS55_Out** structure. Each Mini Motor Integrated Drive system creates an input tag **Motor_x:I** of type

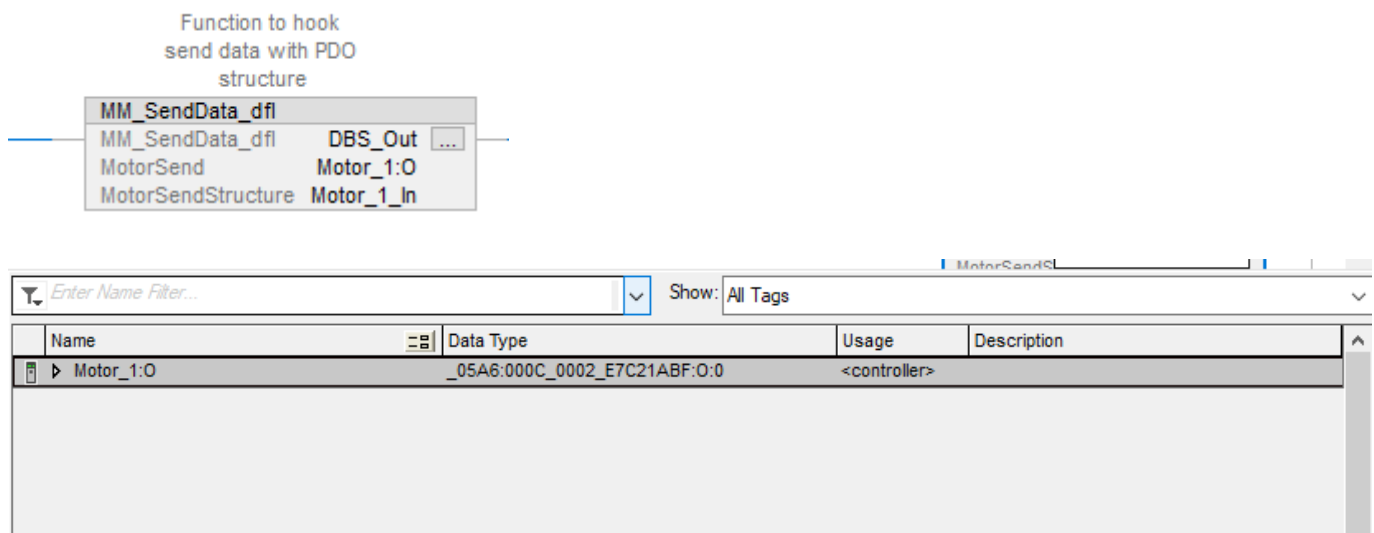
`_05A6:000C_0002_DB137776:I:0`, which must be assigned to a specific `MM_ReadData_dfl` AOI. Then each data is passed to a specific `DBS55_Out` variable. This structure will be used in all operative blocks.



The screenshot displays two tables from a software interface. The top table, titled 'LocalStructure', has columns 'Name', 'Data Type', 'Usage', and 'Description'. It contains one entry: 'Motor_1:I' with data type '_05A6:000C_0002_DB137776:I:0' and usage '<controller>'. The bottom table, titled 'MotorInStructure', also has columns 'Name', 'Data Type', 'Usage', and 'Description'. It contains one entry: 'Motor_1_Out' with data type 'DBS55_Out', usage '<controller>', and description 'PdoTX'.

3.5.2. MM_SendData_dfl

This instruction should be the last and its job is to move data from a `DBS55_In` data type to the motor data inlet. Each Mini Motor Integrated Drive system creates an output tag `Motor_x:0` of type `_05A6:000C_0002_E7C21ABF:0:0`, which must be assigned to a specific `MM_SendData_dfl` AOI. In this way the data manipulated by the rest of the blocks is passed from the `DBS55_In` to the motor.

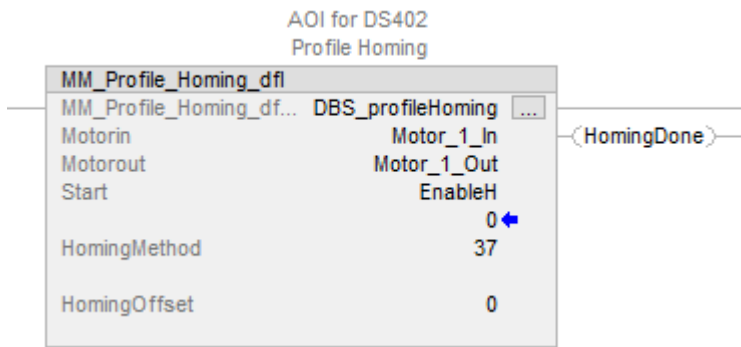


The screenshot displays a function block 'MM_SendData_dfl' with inputs 'MM_SendData_dfl', 'DBS_Out', 'MotorSend', 'Motor_1:O', and 'MotorSendStructure', 'Motor_1_In'. Below the function block is a table titled 'MotorSendS' with columns 'Name', 'Data Type', 'Usage', and 'Description'. It contains one entry: 'Motor_1:O' with data type '_05A6:000C_0002_E7C21ABF:0:0' and usage '<controller>'.

3.5.3. MM_Profile_Homing_dfl

With the `MM_Profile_Homing_dfl` you can home the Integrated Drive family following the DS402 homing procedure as explained in the main DBS/FC manual.

The *Motorin* and *Motorout* pins are hooked to the `DBS55_In` and `DBS55_Out` data types of the specific motor, while *HomingMethod* is the homing procedure selector and *HomingOffset* is the offset to add once the procedure is done. The *HomingDone* bit signals the completion of the homing procedure. This procedure can be started with a *Start* BOOLEAN value.



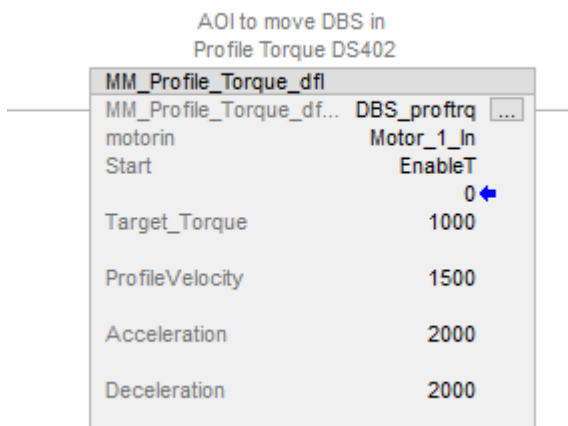
3.5.4. MM_Profile_Torque_dfl

The motor can be used in DS402 Profile Torque mode, useful when you want to have better torque control in applications where speed is secondary.

The AOI must be hooked to a **DBS55_In** data type of the motor to send commands. A BOOL *Start* makes the motor turn. *Target_Torque* is the torque set point, in thousandths of the rated torque, while *ProfileVelocity* limits the speed the motor is allowed to reach while it delivers that torque; acceleration and deceleration shape the ramp.



The help text embedded in this AOI also describes two position limit parameters, *PoslimPos* and *PoslimNeg*. They are not there: the interface of the block is only *Start*, *Target_Torque*, *ProfileVelocity*, *Acceleration* and *Deceleration*. The embedded text was copied from *MM_Profile_Velocity_dfl* and has not been corrected because the DS402 blocks are distributed encrypted; the list above is the one to trust.



3.5.5. MM_Profile_Velocity_dfl

The motor can be used in DS402 Profile Velocity mode, enabling the motor to reach and maintain a specific target speed.

The AOI must be hooked to a **DBS55_In** data type of the motor to send commands. A BOOL *Start* makes the motor turn. You can set the target speed and constrain the trajectory with the other parameters. If the position limits are at their maximum value (2147483647 and -2147483648 respectively) then they are disabled. If they are both at 0 the motor will not move.



PosLimitPos and *PosLimitNeg* have a default value of 0 in the AOI definition, which is exactly the combination that stops the motor. A freshly inserted instance is therefore configured not to move: write 2147483647 and -2147483648 into the two pins, or the limits you actually want, before expecting any motion. The same applies to *MM_Profile_Position_dfl*.

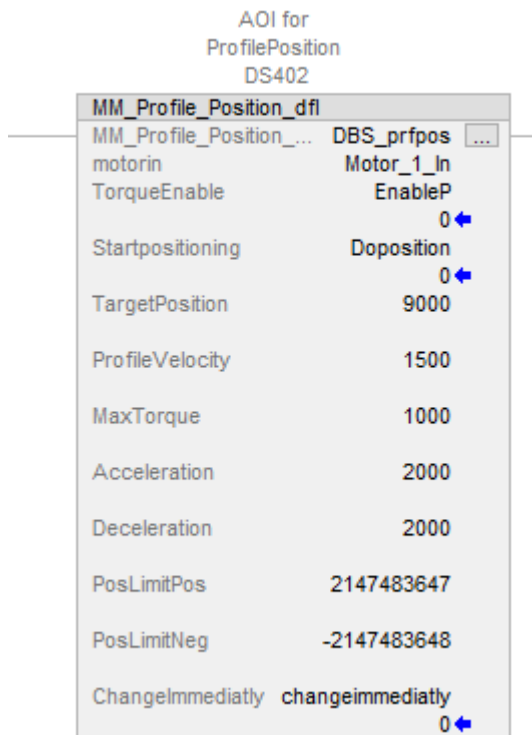
AOI to move DBS
motor in Profile
Velocity DS402

MM_Profile_Velocity_dfl	
MM_Profile_Velocity_...	DBS_profvel ...
motorin	Motor_1_In
Start	EnableV
	0
Target_Velocity	2000
Acceleration	1500
MaxTorque	1000
Deceleration	1500
PosLimitPos	2147483647
PoslimitNeg	-2147483648

3.5.6. MM_Profile_Position_dfl

The motor can be commanded to reach specific positions using its integrated absolute multiturn encoder.

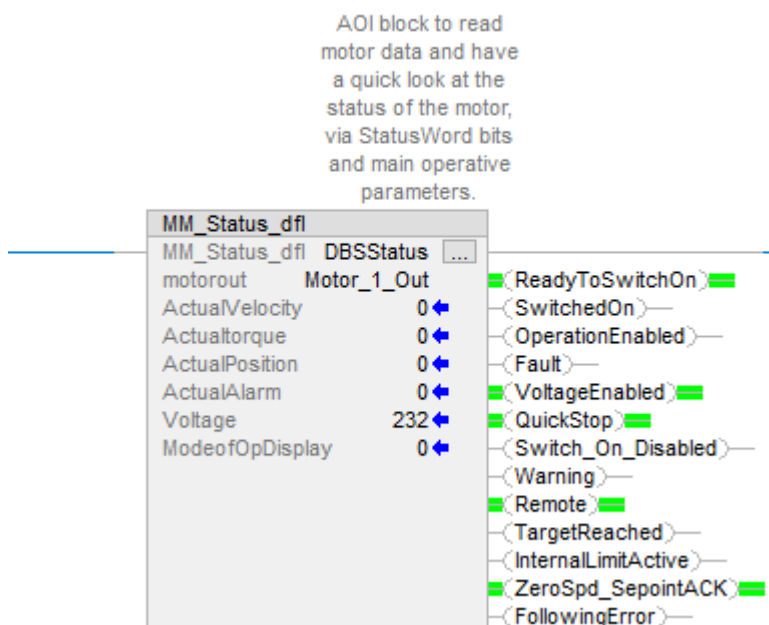
The AOI must be hooked to a *DBS55_In* data type of the motor to send commands. The AOI has two separate commands to start the motor. *TorqueEnable* makes the motor enabled in standstill, keeping the actual position. *Startpositioning* is a rising edge command that makes the motor start its positioning. The trajectory can be constrained via the parameters shown (velocity, deceleration, acceleration, position limits and max torque). The *Changelmmediatly* command is used to make the new position command execute immediately, discarding the current motion; when it is false the new target is buffered and executed at the end of the movement in progress.



3.5.7. MM_Status_dfl

This block can be used to have quick access to the motor status, hooking a **DBS55_Out** structure to it.

It shows the motor speed, actual torque, actual position, actual alarm, voltage and the mode of operation currently displayed by the drive. It also shows the bits of the StatusWord that can be used to power internal logics or diagnostics: the state machine bits (*ReadyToSwitchOn*, *SwitchedOn*, *OperationEnabled*, *Fault*, *VoltageEnabled*, *QuickStop*, *Switch_On_Disabled*), the communication and warning flags (*Warning*, *Remote*, *InternalLimitActive*, *FollowingError*) and the mode-dependent feedback (*TargetReached*, *ZeroSpd_SepointACK*). Remember that the meaning of the last ones changes with the mode of operation the motor is in.



This is the only block shared by the two families: it works just as well in a Rockwell-like project, since it only

needs a `DBS55_Out` structure to read from.



The output named `ZeroSpd_SepointACK` carries a typo in its name, which is visible on every instance. It is the CiA 402 StatusWord bit 12, and it has to be read as *Setpoint acknowledge* in Profile Position, *Speed is zero* in Profile Velocity and *Homing attained* in Homing.

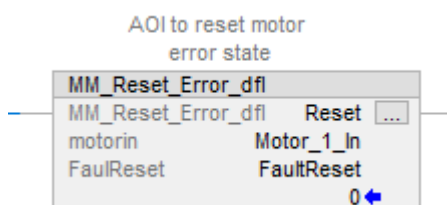
Knowing when a DS402 movement is over. The Profile blocks of the DS402 family are *modes*, not commands: they hold the motor in a mode of operation as long as *Start* is true, and none of them has a *Done* output – `MM_Profile_Homing_dfl` is the exception, because it is the only one that also receives `DBS55_Out`. If you work with the DS402 family only, the end of a movement is read from `MM_Status_dfl`, keeping in mind which mode you are in:

Mode	Movement finished when
Profile Position	<i>TargetReached</i> is high and <i>ZeroSpd_SepointACK</i> has gone back low after the set-point handshake
Profile Velocity	<i>TargetReached</i> is high, meaning the commanded speed has been reached
Homing	<i>ZeroSpd_SepointACK</i> is high, that is <i>Homing attained</i> , and <i>HomingDone</i> of <code>MM_Profile_Homing_dfl</code> follows it

In a Rockwell-like project you do not need any of this: the command blocks run the handshake internally and give you *Done*.

3.5.8. MM_Reset_Error_dfl

This AOI is used to reset the error status on the motor. It is hooked to a `DBS55_In` instance to send the command. The Fault status can be seen on the `MM_Status_dfl` block.



3.6. AOI Rockwell-Like Explanation

These AOI do exactly the same work as the DS402 ones, but they borrow the nomenclature, the pinout and the handshake of the Logix motion instructions. If your team already writes motion code for Rockwell drives, this set lets you keep those habits: you never touch the `ControlWord` and you do not need to know the CiA 402 state machine, because the whole command sequence is run inside the instruction.

3.6.1. How These Blocks Are Built

All the Rockwell-like AOI share the same shape.

The two axis pins. Every block is hooked to the two structures of one motor:

- **Axis_In** is the `DBS55_In` structure, what will be sent to the drive. This is where the instruction writes the `ControlWord`, the mode of operation and the set points.
- **Axis_Out** is the `DBS55_Out` structure, what has been read back from the drive. The instruction needs it to follow the `StatusWord` and to know how far its own sequence has got.

MM_MAU1 and MM_MAU0 are the exceptions, because they are the blocks that fill and empty those two structures. So is MM_MSL, which only writes two values into `DBS55_In` and needs nothing back from the drive.

The three status bits. Each instruction runs an internal state machine that walks the drive through the CiA 402 transitions needed by that command, and reports on it with three bits:

- **Busy** is high while the sequence is running, from the scan the command is taken to the scan it ends, whether it ends well or badly;
- **Done** goes high when the sequence has been completed, and stays high until the rung is dropped;
- **Error** goes high when the sequence has failed, and stays high until the rung is dropped. *Done* and *Error* are mutually exclusive.

Timeout and ErrorCode. Every command block has a *Timeout* input, in milliseconds, and an *ErrorCode* output. The timeout is armed when the command is taken and it covers the whole sequence: if the drive has not answered with the expected `StatusWord` by the time it expires, the block stops on a dedicated error state, raises *Error* and writes the reason in *ErrorCode*. This is what keeps a command that cannot complete from hanging the calling sequence for ever.

ErrorCode	Meaning
0	No error
1	Timeout. The drive did not reach the expected state within <i>Timeout</i> milliseconds
2	The drive is in fault, <code>StatusWord</code> bit 3. Reset it with MM_MAFR, then enable it again with MM_MSO
3	Command specific error. Today only MM_MAH uses it, for a homing error reported by the drive on <code>StatusWord</code> bit 13

The default *Timeout* is 5000 ms. The timer covers the parts of a sequence that have to answer quickly – the CiA 402 transitions, the set-point handshake – and deliberately **not** the parts whose duration depends on the machine. A movement in MM_MAM, a deceleration in MM_MAS and a jog that has already started are not timed: a fixed preset would turn a long but perfectly good move into an error just before it arrived. MM_MAH is the exception, because a homing that never finds its switch has to end somehow: there the timeout covers the whole procedure, so size it on the worst case.

The consequence is worth stating plainly: if a movement never completes, the block does not raise *Error*, it stays *Busy*. That is deliberate. A block visibly stuck on *Busy* tells you the truth; an *Error* that fires on a slow axis does not.

The trigger. All the command blocks work on the **rising edge** of the rung, and holding the rung true does not repeat the command: once the sequence has reached its final state the block stops acting on the axis. To issue the same command again, drop the rung and raise it back.



The edge behaviour comes from the internal state machine, not from an ONS hidden inside the instruction. What re-arms a block is the *EnableInFalse* routine, which runs when the rung goes false: it clears the internal state, drops *Done*, *Busy* and *Error*, and releases the `ControlWord` bits the block had taken. Dropping the rung is therefore not optional

housekeeping, it is the part of the cycle that makes the next command possible. This matters in one practical case: the inputs of a block are read on every scan until its sequence is over, so a parameter changed halfway through is picked up. MM_MAJ makes deliberate use of this, and lets you vary the jog speed while the motor is running.

One command at a time on one axis. The blocks write directly on the ControlWord and on the mode of operation of the axis, and there is no arbitration between them. Two command blocks enabled at the same time on the same axis overwrite each other, and which one wins depends on the order of the rungs.



Keep one command block active per axis at a time. The usual shape is a sequencer, or a set of mutually exclusive rungs, where the next command is enabled on the *Done* of the previous one. The same applies when you mix the two families on one axis, which is allowed and sometimes useful – MM_Profile_Torque_dfl inside a Rockwell-like project, for instance – but the DS402 Profile blocks hold their mode as long as *Start* is true, so their *Start* has to be dropped before a Rockwell-like command takes over the same axis.

The order in the scan. The layout of a routine is always the same: MM_MAJ first, all the command blocks and MM_MSL in the middle, MM_MAUO last. Anything written by a command block reaches the drive only when MM_MAUO is executed.

Connection loss. MM_MAJ copies the input assembly on every scan without checking that the EtherNet/IP connection is up. If the drive goes offline the DBS55_Out structure keeps the last values that arrived, so a disconnected axis can look like an axis sitting on its target. Take the connection status of the module with a GSV on the MODULE object and use it to hold the axis logic and to raise your own alarm; the AOI cannot do it for you, because they only see the data structures and not the module.

3.6.2. Equivalence with the DS402 AOI

If you already know one set, this table tells you where to look for the same function in the other one.

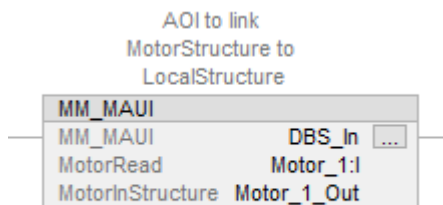
Rockwell-like	DS402 equivalent	What happens on the drive
MM_MAJ	MM_ReadData_dfl	Copies the input assembly Motor_x:I into the DBS55_Out structure
MM_MAUO	MM_SendData_dfl	Copies the DBS55_In structure into the output assembly Motor_x:O
MM_MSO	The <i>TorqueEnable</i> / <i>Start</i> pins of the profile AOI	Walks the state machine up to <i>Operation Enabled</i> : the axis is under control and holds its position
MM_MSF	Dropping <i>TorqueEnable</i> / <i>Start</i>	Leaves <i>Operation Enabled</i> and switches the power stage off
MM_MAFR	MM_Reset_Error_dfl	Fault reset
MM_MAM	MM_Profile_Position_dfl	Mode of operation 1, Profile Position
MM_MAJ	MM_Profile_Velocity_dfl	Mode of operation 3, Profile Velocity
MM_MAH	MM_Profile_Homing_dfl	Mode of operation 6, Homing
MM_MAS	<i>Halt</i> bit of the ControlWord	Halt followed by a position in place, so the axis stops and holds
MM_MSL	<i>PosLimitPos</i> and <i>PosLimitNeg</i> pins of the Profile Position and Profile Velocity AOI	Software position limits, object 607Dh
MM_Status_dfl	MM_Status_dfl	The same block serves both families

Two differences are worth pointing out before going through the blocks one by one.

- The DS402 blocks are **modes**: as long as their *Start* pin is true the motor stays in that mode of operation, and the block is the one keeping it alive. The Rockwell-like blocks are **commands**: they are fired once on a rising edge, they run their sequence and they tell you when it is over. This is why the DS402 set has a *Start* per profile, while the Rockwell-like set has MM_MSO and MM_MSF as separate enable/disable commands.
- There is no Rockwell-like block for Profile Torque. If you need torque mode, use MM_Profile_Torque_dfl from the DS402 set: it works on the same DBS55_In structure and can be dropped into a Rockwell-like project.

3.6.3. MM_MAUl

Motion Axis Update Input. This block links the motor structure of data exchange to the local structure of data exchange, and it must be the first instruction of the scan for that axis. *MotorRead* takes the module input tag *Motor_x:I* created by Studio 5000 when the drive was added to the I/O tree, and *MotorInStructure* takes the DBS55_Out structure that every other block will read. It is the Rockwell-like name of MM_ReadData_dfl.



3.6.4. MM_MSO

Motion Servo On. Enables the drive: the motor is torque enabled and in standstill, holding the position it is in. Internally the instruction runs the *Shutdown*, *Switch On* and *Enable Operation* transitions of the CiA 402 state machine, and raises *Done* once the drive has reached *Operation Enabled*.

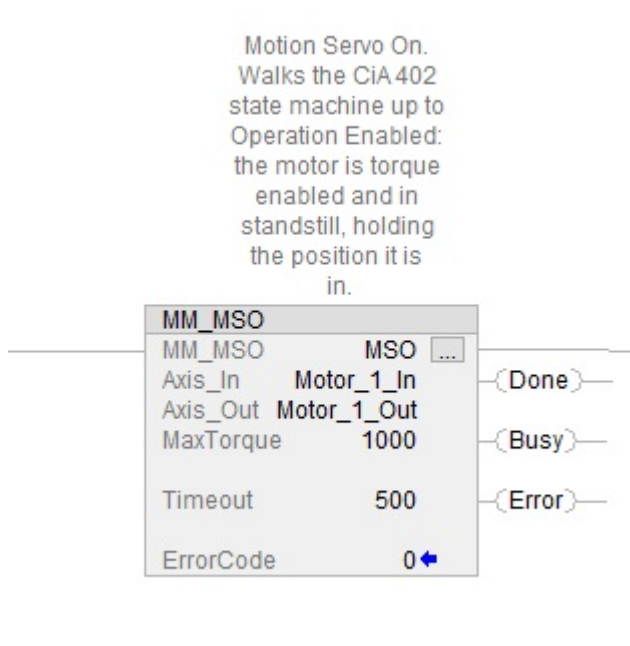
This is the first command to give to an axis: MM_MAM, MM_MAJ and MM_MAH act on a motor that is already enabled.

MaxTorque is the torque limit applied to the axis while it is enabled, in thousandths of the rated torque, 1000 being 100 %. It used to be forced to 1000 inside the block; from revision 1.1 it is a pin, so a limit set for mechanical protection is no longer silently raised back to the rated value at every servo on. It carries the same name as the equivalent pin of the DS402 Profile blocks.



While it enables the axis, the instruction also writes mode of operation 3 and a target speed of 0. This is deliberate, and it is what makes *Operation Enabled* a standstill instead of the restart of whatever profile happened to be configured before. It also means that a servo on given after setting up a torque or position profile cancels that mode: set the mode with the command block that follows, since MM_MAM, MM_MAJ and MM_MAH each write their own.

If the drive is in fault the block does not even try: it stops immediately with *Error* and *ErrorCode* = 2. Clear the alarm with MM_MAFR first.

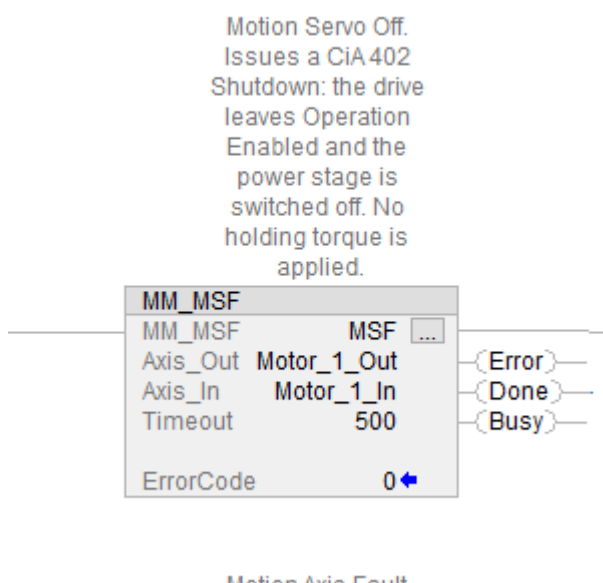


3.6.5. MM_MSF

Motion Servo Off. Disables the power stage: the axis is no longer under control and no holding torque is applied. Use it at the end of a cycle or as a controlled way to release the axis, not as an emergency stop – the safety functions of the machine must never rely on a fieldbus command.

The instruction issues a CiA 402 *Shutdown* and completes as soon as the drive drops out of *Operation Enabled*. A drive that is already in *Ready To Switch On*, in *Switch On Disabled* or in fault is a drive whose power stage is off, so all three are accepted and the block completes: calling MM_MSF on an axis that is already down is harmless and always ends with *Done*.

The three states are recognised with the standard CiA 402 masks, 6Fh and 4Fh, and not by reading single bits: the block tells *Ready To Switch On* from *Switch On Disabled* and from *Fault* instead of lumping them together.

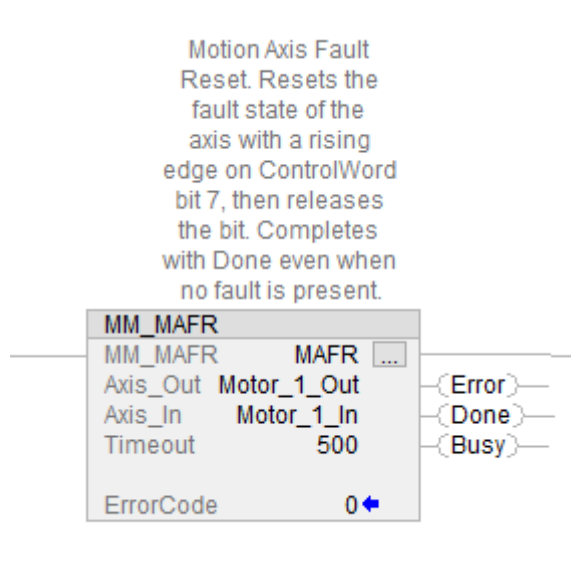


3.6.6. MM_MAFR

Motion Axis Fault Reset. This block is used for the reset of errors on the drive, and it is the Rockwell-like name of MM_Reset_Error_dfl. Check the main servo manual to see more information about alarms and conditions.

The instruction looks at the *Fault* bit of the StatusWord. If the axis is faulted it raises ControlWord bit 7, waits for the fault to clear and then releases the bit, which is what CiA 402 requires for the next fault to be resettable in turn. If the axis is not faulted there is nothing to do and the block completes immediately with *Done*: you can call it in a general reset, or at the start of a cycle, without having to check first whether an alarm is actually there.

After a successful reset the drive is back in a disabled state, so give a new MM_MSO to bring the axis under control again. If the cause of the alarm is still present the fault comes straight back: read the alarm code from MM_Status_dfl before resetting blindly.



3.6.7. MM_MAM

Motion Axis Move. This block moves the motor to a specific position. The trajectory of the motor is built via *Acceleration*, *Deceleration*, *Velocity* and *Position*, and it will always be a trapezoidal trajectory. Velocity is in rpm, acceleration and deceleration are in rpm/s, position is in encoder pulses or whatever scaled position you are using (see the main manual for how to scale). *MoveType* chooses between an absolute (0) and an incremental (1) move: with 0 the value in *Position* is the point to reach, with 1 it is the distance to travel from where the motor is now. The pin carries the name and the values of *Move Type* on a Logix MAM, and they agree with CiA 402, where ControlWord bit 6 is 0 for absolute and 1 for relative.

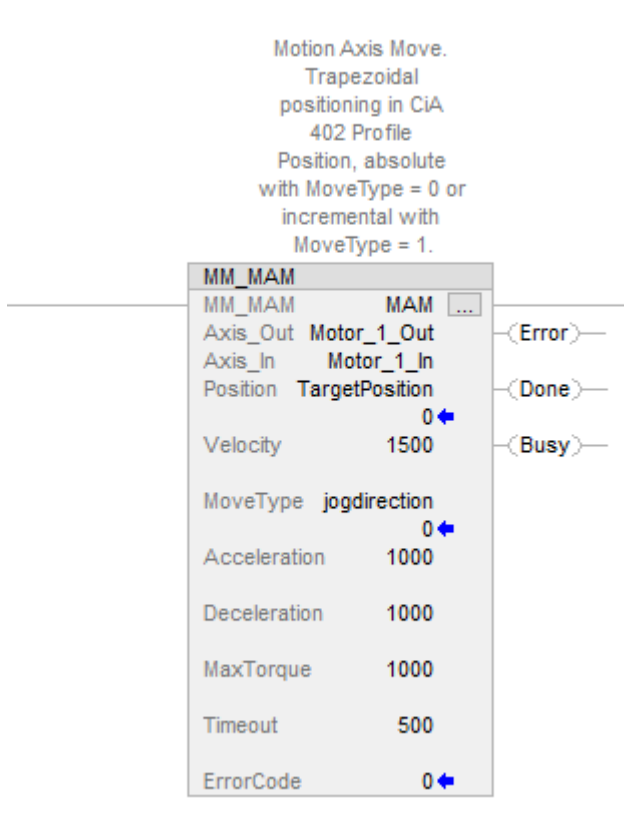
Internally the instruction selects mode of operation 1 and runs the standard Profile Position set-point handshake – raise ControlWord bit 4, wait for StatusWord bit 12, drop bit 4, wait for bit 12 to fall, then wait for bit 10, *Target Reached*. *Done* comes up when the movement has been completed, so it is the natural condition to chain the next step of a sequence.

MaxTorque is the torque limit applied to the move, in thousandths of the rated torque, and carries the same name as the equivalent pin of the DS402 Profile blocks. *Timeout* covers the set-point handshake only. The travel itself is not timed, so a long move cannot end in a spurious error.



In revision 1.0 this pin was called *Direction* and its polarity was inverted with respect to this description: 0 produced an **incremental** move and 1 an absolute one. Revision 1.1 renames

it *MoveType* and makes it behave as documented here. On an application written against 1.0 the two meanings swap over at the update, and a positioning that used to be absolute becomes a step that accumulates at every cycle. See [Updating from Revision 1.0](#).

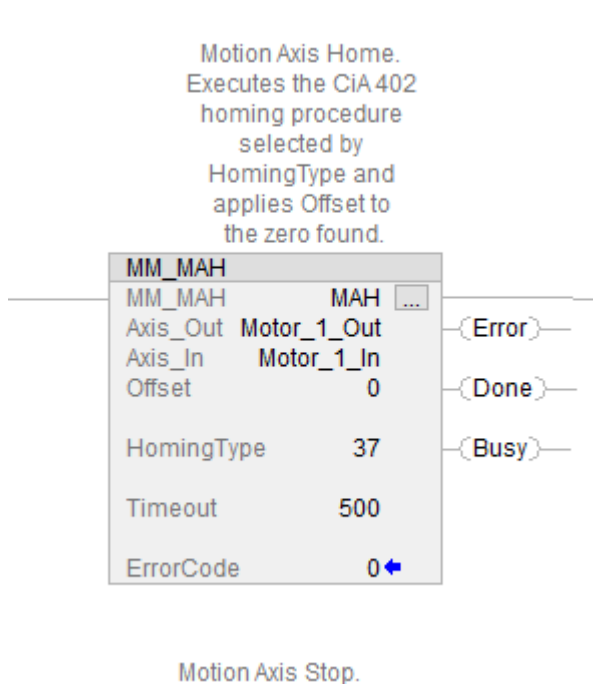


3.6.8. MM_MAH

Motion Axis Home. Block used for homing. *HomingType* selects the procedure and *Offset* is the value added to the zero once the procedure is done. Check the main manual for all homing modes; homing mode 37, the default one, is homing in place, which simply declares the current position as zero and therefore completes immediately without moving the motor.

The instruction selects mode of operation 6, loads the method and the offset, then starts the procedure with ControlWord bit 4. Completion is taken from StatusWord bit 12, *Homing attained*: the block first waits for the drive to clear it, which is how a new homing is acknowledged, and then for it to come back high, so a homing performed long ago cannot be mistaken for the one just commanded. StatusWord bit 13, *Homing error*, ends the sequence with *ErrorCode* = 3.

A homing that searches a switch can take a long time: size *Timeout* on the worst case, at the slowest search speed and from the far end of the stroke, otherwise a perfectly good homing ends in *ErrorCode* = 1.



3.6.9. MM_MAS

Motion Axis Stop. Stops the movement in progress. The instruction issues a Halt followed by a position in place, so the motor decelerates, stops and then holds the position it stopped at, staying enabled and ready for the next command. It is also the way to end a jog started with MM_MAJ.

In detail: the block switches the axis to mode of operation 3 with a target speed of zero and raises the Halt bit, which brings the motion down on the deceleration ramp whatever the axis was doing. It waits for the axis to fall inside the standstill window, then switches to mode of operation 1, takes the position where the axis stopped as target, releases Halt and engages the hold with the set-point handshake. At the end every bit it had taken is released, which is what makes the next command work – in revision 1.0 the Halt bit stayed up and the axis silently ignored everything that came after.

The stop does not rely on the Halt bit alone, and that is deliberate: on this drive Halt does not reliably interrupt a Profile Position move. Going through zero speed in Profile Velocity stops a jog and a positioning alike, and it is what makes MM_MAS behave like its Rockwell namesake, which interrupts and clears the active motion command rather than suspending it.



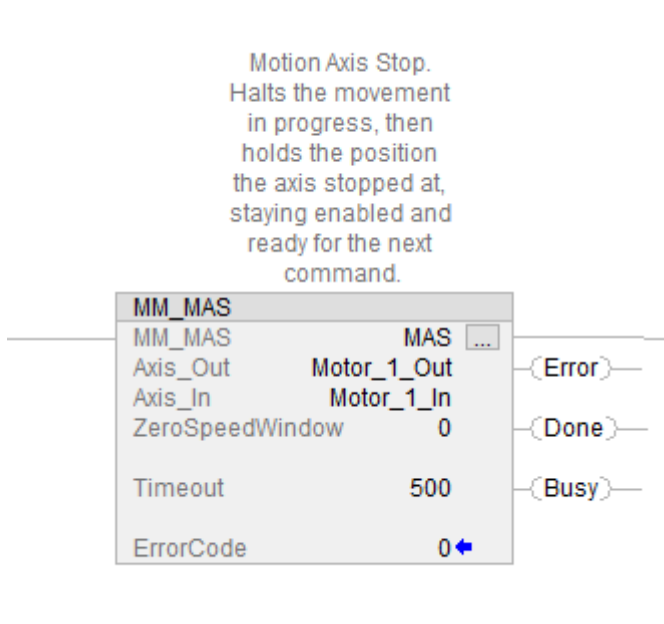
Since MM_MAS clears a jog by writing a target speed of zero, the rung of the MM_MAJ that started that jog must already be down. An MM_MAJ still held true rewrites the speed on every scan, and the jog resumes as soon as Halt is released. One command at a time on one axis, as always.

ZeroSpeedWindow is the half width of the standstill window, in rpm, and defaults to 20. On a high inertia axis, or with a gear ratio that makes 20 rpm a very small number at the motor shaft, widen it: too tight a window and the block waits for a standstill that never comes, until *Timeout* expires with *ErrorCode* = 1.



The block leaves the axis in mode of operation 1, Profile Position, and does not restore the mode that was active before the stop – that mode is what holds the position. Inside the Rockwell-like family this has no consequence, because MM_MAM, MM_MAJ and MM_MAH each write their own mode. Keep it in mind if the next thing to act on that axis is a DS402

Profile block.



3.6.10. MM_MAJ

Motion Axis Jog. Moves the axis in manual mode, in the direction chosen with *Direction*. The trajectory is trapezoidal, built with *Velocity*, *Acceleration* and *Deceleration*. Unlike MM_MAM there is no target: the motor keeps turning until you stop it.

Direction = 1 turns the motor in the positive sense of the drive, *Direction* = 0 in the other one. *Velocity* is taken in absolute value before the sense is applied, so passing a negative speed does not reverse the jog a second time.

The jog speed is written to the drive on every scan **while the rung is true**, so *Velocity* can be varied on the fly and the axis follows it on the *Deceleration* and *Acceleration* ramps. This is what makes a two-button manual panel with a speed potentiometer work with a single instance of the block. Let the rung go and the axis keeps the last speed it was given.

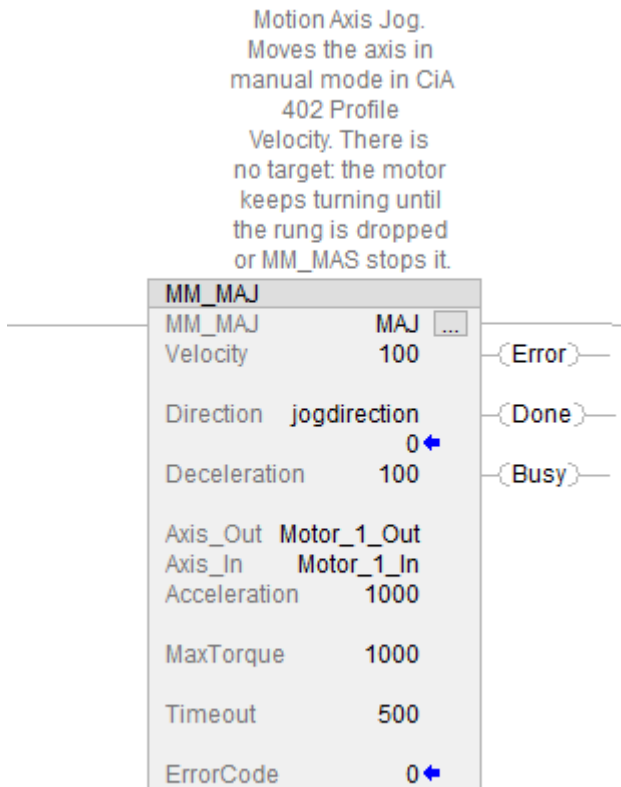
The jog keeps going when the rung goes down. MM_MAJ is fired once and forgotten, exactly like MM_MAM: the speed stays written to the drive and the motor keeps turning. Dropping the rung does not stop it, and on its way out the block does not touch the axis at all.

MM_MAS is what stops it. It clears the commanded speed and holds the position the axis stopped at. Short of MM_MSF or a fault, there is no other way to end a jog.

MaxTorque is the torque limit applied to the jog, in thousandths of the rated torque. *Timeout* only covers the start of the jog, that is the drive accepting mode 3: it does not limit how long the motor turns.



Done on this block means *command issued*, not *movement finished*, and *Busy* falls while the motor is still running. There is no target to reach, so there is nothing else it could mean.



3.6.11. MM_MSL

Motion Set Limits. Writes the software position limits of the axis, object 607Dh, into the `DBS55_In` structure. There is no Logix instruction with this name: on a Rockwell drive the soft travel limits are part of the axis configuration, while here they travel with the process data.

That is also why the block is needed. *PositionLimitNeg* and *PositionLimitPos* are two members of `DBS55_In`, and MM_MAUO sends them to the drive on every scan like everything else in the structure. What the drive applies is what the PLC sends, not the values stored in P164 and P166. A `DBS55_In` tag freshly created in Studio 5000 is all zero, and with both limits at 0 the drive holds the axis in place and refuses any movement. None of the command blocks writes these two members.

The two pins default to -2147483648 and 2147483647, the full range of the drive, which leaves the limits in effect disabled. An instance with nothing changed is therefore enough to get a working axis. The pins are optional: to set real limits, write them on the instance tag, for example `MSL_Axis1.PositionLimitPos`, from your code or from the HMI. They are expressed in the same unit as *Position* on MM_MAM, and a new value is applied on the next scan.

Put one instance per axis on a rung that is always true, anywhere before MM_MAUO. If the rung goes false the block stops writing and the structure keeps the last pair it received.

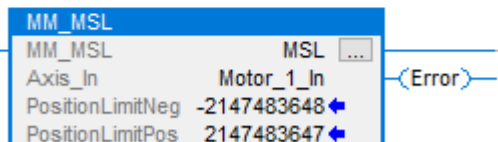
Error goes high when *PositionLimitPos* is not greater than *PositionLimitNeg*. In that case the block writes nothing, and the drive keeps the last valid pair: on a structure that never received one, that means 0 and 0 and an axis that does not move.



MM_Profile_Position_dfl and MM_Profile_Velocity_dfl write the limits themselves, from their

own *PosLimitPos* and *PosLimitNeg* pins. On an axis driven by those blocks you do not need *MM_MSL*; if you use both, whichever comes later in the scan wins.

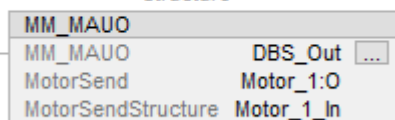
Motion Set Limits.
Writes the software
position limits of
the axis into the
DBS55_In structure
on every scan.
Without it the drive
receives both limits
at 0 and holds the
axis where it is.



3.6.12. MM_MAUO

Motion Axis Update Output. This block links the local data structure with the cyclic data exchange, and it must be the last instruction of the scan for that axis. *MotorSendStructure* takes the *DBS55_In* structure written by all the command blocks, and *MotorSend* takes the module output tag *Motor_x:0*. It is the Rockwell-like name of *MM_SendData_dfl*.

Function to hook
send data with PDO
structure



3.6.13. Updating from Revision 1.0

Revision 1.1 of the command blocks corrects defects that could stop a sequence or, in one case, move the axis the wrong way. Updating is worth doing, but it is not a matter of importing the files and downloading: two changes alter the interface and one alters the behaviour of a working machine.

What breaks the compilation. The input *Decerelation* of *MM_MAM* and *MM_MAJ* has been renamed *Deceleration*, and both ramps have grown from *INT* to *DINT* so that they match the fields they are written into and the corresponding pins of the *DS402 Profile* blocks. *Studio 5000* drops the argument of a renamed parameter: after the import, every instance of the two blocks shows an empty pin that has to be wired again. Nothing else in the interface has been removed.

What breaks the machine. The *Direction* input of *MM_MAM* is now called *MoveType*, the name it carries on a *Logix MAM*, and its polarity has been corrected: revision 1.0 disagreed with both the *Rockwell instruction* and *CiA 402*. The rename costs you a pin to wire again on every instance, which the compiler points out; the polarity costs you nothing visible, which is why it is the dangerous half. If your application was written by trying the block rather than by following the guide, it is tuned on the old meaning and every value has to be

inverted:

Value	Revision 1.0 did	Revision 1.1 does
0	Relative move	Absolute move
1	Absolute move	Relative move

What is new and optional. *Timeout* and *ErrorCode* appear on every command block, *MaxTorque* on MM_MSO, MM_MAM and MM_MAJ, where the torque limit used to be hard wired at 1000, and *ZeroSpeedWindow* on MM_MAS. They all carry a default, so an instance that ignores them keeps working; wiring *ErrorCode* to the HMI, though, is the whole point of the revision. MM_MSL is a new block altogether: if your application sets the position limits by writing fixed values into *DBS55_In*, it keeps working, and MM_MSL is the tidier way to do the same thing.

Suggested order of work.

1. Import the files into a copy of the project, MM_MSL included. They are single target exports, one Add-On Instruction each, so importing one no longer overwrites the whole family.
2. Rebuild the project and fix the empty *Deceleration* pins that the import has left behind.
3. Go through every instance of MM_MAM: wire the *MoveType* pin the rename has left empty, and invert the value, unless you can verify the instance was already following the documented meaning.
4. Set *Timeout* on the blocks that move the axis: the default 5000 ms is short for a long positioning or for a homing that searches a switch.
5. Bring the axis up with the mechanics free, and check the whole cycle: enable, homing, a positioning, a jog, a stop, a jog again. The jog, stop, jog cycle in particular did not work before and is the quickest way to confirm that the update is in.
6. Only then reconnect the mechanics.



With homing method 37, homing in place, the drive completes without moving and the handshake on *Homing attained* can be faster than one PLC scan. Check on the bench that MM_MAH raises *Done* and does not end on *ErrorCode* = 1.

3.7. Writing Parameters with Async Messaging

DBS/FC Parameters can be reached with Async messaging in EtherNet/IP, in lieu of the SDO system of CANopen and EtherCAT.

3.7.1. Async Messaging

The application layer for the DBS55 motor is very close to the CiA402 servodrive definition; the PLC can interact with the motor CiA402 state machine using *ControlWord* and *StatusWord*.

Nevertheless, an important part of the CiA402 servodrive model is managed via asynchronous read/write of the CANopen object dictionary. In the EtherNet/IP implementation, the asynchronous messaging is implemented as Class 3 GetAttribute/SetAttribute services, where portions of the CANopen object dictionary are exposed to the EtherNet/IP world via attributes.

There are 4 main EtherNet/IP manufacturer objects that expose the attributes which act as a gateway to the

CANopen world:

- **COM object – object 0x64:** this object exposes some basic information (e.g. software release) and allows saving parameters to non-volatile storage.
- **PAR object – object 0x65:** this object allows changing the motor parameters via PLC; this allows, for example, changing dynamically the Kp of the speed loop if the load of the motor changes during a machine phase.
- **MISC object – object 0x66:** this object allows reading and writing hardware related information (e.g. analog input value, heatsink temperature etc.).
- **CIA402 objects – object 0x67:** this object allows reading and writing CiA402 objects: for example, reading object 6081h.00 is translated to a GetAttribute on object 0x67, instance 0x60, attribute 0x81.

As an example we will read the BSI parameter P160, PosFactorNum, which is used to scale the motor position value. Using the MSG block, the configuration will be as follows.

As seen in the first chapter of this guide, P160 is part of the Class 0x65, it is in the first instance and its value in hex is A0. Remember to create a Destination element where to write down the value of interest.

Message Configuration - MSG_GetRev

Configuration Communication Tag

Message Type: CIP Generic

Service Type: Get Attribute Single

Service Code: e (Hex) Class: 65 (Hex) Instance: 1 Attribute: a0 (Hex)

Source Element:

Source Length: 0 (Bytes)

Destination Element: APL_Rev

New Tag...

☐ Enable
 ☐ Enable Waiting
 ☐ Start
 ☐ Done
 Done Length: 0

☐ Error Code:
 Extended Error Code:
 ☐ Timed Out

Error Path: Motor_1

Error Text:

OK Annulla Applica ?

Conversely, to write the same parameter, the MSG must be set in Set Attribute Single.

Message Configuration - MSG_GetRev
✕

Configuration*
Communication
Tag

Message Type: CIP Generic ▼

Service Type: Set Attribute Single ▼

Service Code: 10 (Hex) Class: 65 (Hex)

Instance: 1 Attribute: A0 (Hex)

Source Element: Posscaling ▼

Source Length: 2 (Bytes)

Destination Element: ▼

New Tag...

☐ Enable
☐ Enable Waiting
☐ Start
☐ Done
Done Length: 0

☐ Error Code:
Extended Error Code:
☐ Timed Out ↩

Error Path: Motor_1

Error Text:

OK
Annulla
Applica
?

4. Micro850 Application Example

Everything shown so far assumes a Logix controller: a scanner that reads EDS files, builds an I/O tree and runs Add-On Instructions. The Micro800 family works in a different way, and until recently it could not run the drive cyclically at all. It can now, and this chapter shows how.

4.1. Class 1 on Micro800: What Is Possible and On Which CPU

Implicit messaging – the Class 1 cyclic exchange used by the whole guide up to here – is available on Micro800 controllers only on the catalogue numbers containing the letter **E**, that is **2080-L50E** (Micro850) and **2080-L70E** (Micro870), with controller firmware **21.011 or later** and **Connected Components Workbench 21 or later**.



Older Micro850 controllers, catalogue number 2080-LC50-xxxxx, cannot act as an EtherNet/IP scanner: they only do explicit (Class 3) messaging. Read the label on the CPU before promising a delivery date – **2080-L50E-24QWB** is fine, **2080-LC50-48QBB** is not, and with the latter the only way to move the drive is the explicit messaging described in [Reading and Writing Parameters from CCW](#), or a change of CPU.



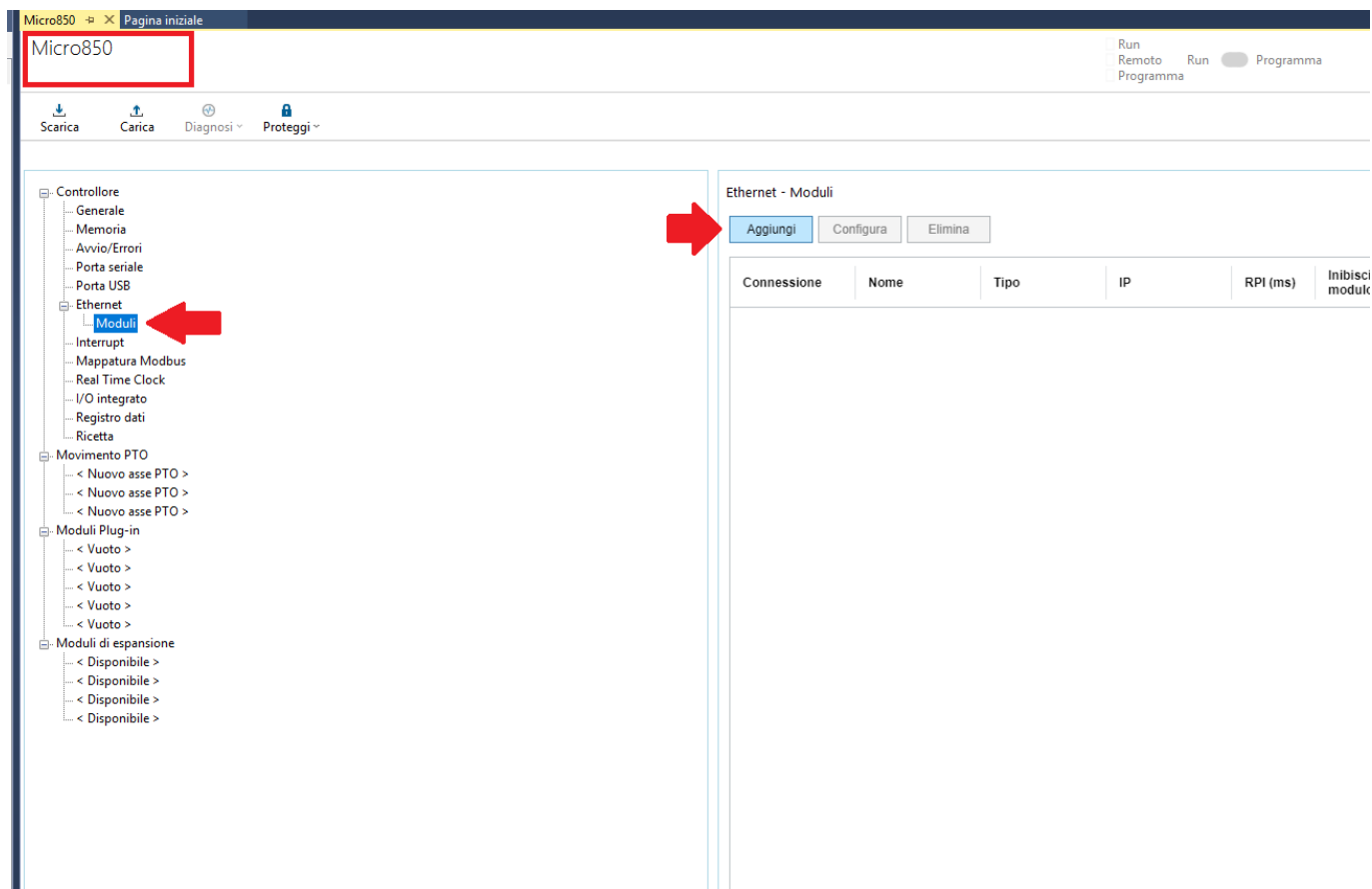
"MicroLogix 850" does not exist as a Rockwell designation, although it is how the controller is often asked for. MicroLogix (1100, 1400) is a different, older family, and it is not an EtherNet/IP scanner either.

Item	Requirement
Controller	Micro850 2080-L50E or Micro870 2080-L70E, firmware >= 21.011
Software	Connected Components Workbench >= 21, Standard or Developer Edition
Drive	DBS/FC with EtherNet/IP interface, firmware 5.004, P176 = 0 (Default layout)
Network	Direct connection or switch; drive and controller in the same subnet, static addresses

4.2. Add the DBS as a Generic Device

CCW has no EDS catalogue for third-party Class 1 devices: the drive is added as a **generic device** and the numbers that Studio 5000 would have taken from **DBS55_CIP_df1.eds** are typed in by hand. They are the same numbers documented in the *Process Data* section of the EtherNet/IP chapter, only expressed the way CCW asks for them.

- Set the address of the drive first, exactly as described in [Configure the Motor Address](#): static (P215 = 0) with P170, P172 and P174 written from BSI, or DHCP (P215 = 1) with a reserved lease. The generic device you are about to create points to one fixed address.
- Give the controller a static address in the same subnet, from the *Ethernet* node of the CCW project.
- In the project tree open *Controller* → *Ethernet* → *Modules* and click *Add*.



- Select *Generic Device* as the module type, give it a name — this example uses **DBS** — and enter the IP address of the drive.
- Set *Electronic Keying* to *Disable Keying*, as in the Studio 5000 flow.
- Fill in the communication configuration with the values below and confirm with OK.

Nuovo modulo

Generale

Nome:

DBS

Tipo:

Dispositivo generico

Indirizzo IP:

192.168.250.111

Codifica elettronica:

Disabilita codifica

Descrizione:

Mini Motor Integrated drive

Connessione

Intervallo pacchetti richiesti (Requested Packet Interval, RPI):

20.0

ms

☒

Connessione Unicast su EtherNet/IP

☐

Inibisci modulo

Errore di connessione:

Configura comunicazione

Formato comunicazione:

Dati - INT

Istanza di assemblaggio:

20

16-bit

Uscita:

100

50

16-bit

Configurazione:

102

4

8-bit

OK

Annulla

Field	Value	Where it comes from
Communication Format	Data - INT	The process data of the DFL layout is word aligned
Input – Assembly Instance	101	Assembly 0x65, T>>O, drive outputs
Input – Size	20	Length of assembly 0x65 in the DFL layout
Output – Assembly Instance	100	Assembly 0x64, O>>T, drive inputs
Output – Size	50	Length of assembly 0x64 in the DFL layout
Configuration – Assembly Instance	102	Assembly 0x66
Configuration – Size	4	Configuration assembly, always left at zero
RPI	20 ms	Starting point, see below



The three assembly lengths must match the layout selected on the drive with P176. The table above is the **Default (DFL)** layout, P176 = 0, the same one used by all the AOI and example programs of this guide. Any other layout has different lengths – 30 or 40 bytes on the input side – and the connection is simply refused.

CCW creates three global variables named after the module. With the module called **DBS** and the INT communication format they are:

Variable	Type	Content
DBS_0	INT[0..24]	Written to the drive, assembly 100
DBS_I	INT[0..9]	Read from the drive, assembly 101
DBS_C	SINT[0..3]	Configuration, leave every byte at zero

About the RPI: 20 ms is a sensible starting point. The Micro800 refreshes the **_I** and **_0** arrays asynchronously with respect to the program scan, and the position loop is closed inside the drive – the PLC only supervises and issues commands. Lowering the RPI does not improve the dynamics of the axis, it only loads the CPU. The maximum number of Class 1 nodes supported by the controller is in the Rockwell manual 2080-UM002, chapter 8.

Download the project and go online: on the module page the connection state must read *Running*. If it does not, see [Commissioning and Diagnostics](#).

4.3. Process Data Seen From CCW

With the INT communication format the two assemblies are seen as arrays of 16-bit words, and every item of the DFL layout falls on a word boundary. The tables below are the byte map of the *Process Data* section translated into word indexes; 32-bit values occupy two consecutive words, least significant word first.

DBS_0 – what the PLC writes, assembly 0x64:

Word	Name	Type	Units
0	ControlWord	UINT	CiA 402 control word
1...2	TargetPosition	DINT	position units
3	TorqueLimit	UINT	thousandths of rated torque
4...5	TargetVelocity	DINT	rpm
6	TargetTorque	INT	thousandths of rated torque
7	ModeOfOperation	SINT	low byte; high byte is padding, leave at 0
8...9	ProfileVelocity	UDINT	rpm
10...11	ProfileAccel	UDINT	rpm/s
12...13	ProfileDecel	UDINT	rpm/s
14...15	PositionLimit.Neg	DINT	position units
16...17	PositionLimit.Pos	DINT	position units
18...19	HomingOffset	DINT	position units
20	HomingMethod	SINT	low byte; high byte is padding, leave at 0
21...24	—	—	Not used by the DFL layout, leave at 0

DBS_I – what the drive returns, assembly 0x65:

Word	Name	Type	Units
0	StatusWord	UINT	CiA 402 status word
1...2	PositionActualValue	DINT	position units
3	TorqueActualValue	INT	thousandths of rated torque
4...5	VelocityActualValue	DINT	rpm
6	VDclink	UINT	0.1 V
7	ModeOfOperation display	SINT	low byte; high byte is padding
8	ActualAlarm	UINT	0 = no alarm, see the main DBS/FC manual
9	—	—	Padding



Before writing any logic, confirm the alignment online. Force `DBS_0[0]` to 16#1234 and check that the drive reacts as expected to that ControlWord, then write 16#00010000 into the target position and verify that `DBS_0[1]` reads 0 and `DBS_0[2]` reads 1. If the two words come out swapped, the COP calls of the next paragraph need *Swap* set to TRUE.

4.4. Structured Text Example

There are no Add-On Instructions for CCW: the CiA 402 state machine has to be written in the program, and this is what the two AOI families spare you on a Logix controller. The code below is the minimum that drives one axis — enable, homing, Profile Position move, fault reset — and is meant to be written as a UDFB so that it can be instanced once per motor.

Two peculiarities of the Micro800 ST dialect are worth remembering:

- **AND**, **OR** and **NOT** work on BOOL operands only; bitwise operations on integers use the `AND_MASK`, `OR_MASK`, `XOR_MASK` and `NOT_MASK` functions;
- **COP** is a function block, not a function, so every call needs its own instance declared among the local variables. It copies the bit pattern without numeric conversion, and its *Length* parameter counts **destination** elements.

4.4.1. Variables to Declare

Name	Type	Description
<code>iSw</code>	INT	StatusWord as read
<code>iActPos</code> , <code>iActVel</code>	DINT	Actual position and velocity
<code>iActTrq</code> , <code>iAlarm</code> , <code>iDcVlt</code> , <code>iModeDisp</code>	INT	Torque, alarm, DC link voltage, active mode
<code>oCw</code>	INT	ControlWord to be written
<code>oTargetPos</code> , <code>oTargetVel</code> , <code>oNegLimit</code> , <code>oPosLimit</code> , <code>oHomeOffs</code>	DINT	Set points and limits
<code>oProfVel</code> , <code>oProfAcc</code> , <code>oProfDec</code>	UDINT	Profile parameters

Name	Type	Description
<code>oTrqLimit</code> , <code>oTargetTrq</code> , <code>oMode</code> , <code>oHomeMethod</code>	INT	Torque limit and target, requested mode, homing method
<code>stReady</code> , <code>stSwitchedOn</code> , <code>stOpEnabled</code> , <code>stFault</code> , <code>stSwOnDisabled</code> , <code>stQuickStop</code>	BOOL	Decoded states
<code>stTargetReached</code> , <code>stSetpointAck</code> , <code>stHomingDone</code> , <code>stHomingError</code>	BOOL	StatusWord bits 10, 12 and 13
<code>cmdEnable</code> , <code>cmdReset</code> , <code>cmdHome</code> , <code>cmdMove</code> , <code>cmdStop</code>	BOOL	Commands from the application or the HMI
<code>seq</code>	INT	Sequence step
<code>cpRdPos ... cpWrHo</code>	COP	One COP instance per copy line



`oCw` and `iSw` are declared INT and not UINT because that is the element type of the assembly arrays. All the ControlWord values used here are below 16#8000, so the sign never gets in the way; if you need to set bit 15 you have to write the equivalent negative value.

4.4.2. Block 1—Reading and Decoding the Inputs

```
(* =====
ASSEMBLY 101 -> typed variables.
Word aligned items are assigned directly, 32 bit items are
copied with COP: Length counts DESTINATION elements.
===== *)

iSw      := DBS_I[0];          (* StatusWord      *)
iActTrq  := DBS_I[3];          (* 0.1 % of rated  *)
iDcVolt  := DBS_I[6];          (* 0.1 V           *)
iModeDisp := DBS_I[7];          (* padding byte is 0 *)
iAlarm   := DBS_I[8];          (* 0 = no alarm     *)

cpRdPos ( TRUE, DBS_I, 1, iActPos, 0, 1, FALSE );  (* words 1..2 *)
cpRdVel ( TRUE, DBS_I, 4, iActVel, 0, 1, FALSE );  (* words 4..5 *)

(* ---- CiA 402 state machine decoding ---- *)
stSwOnDisabled := ( AND_MASK( iSw, 16#004F ) = 16#0040 );
stReady        := ( AND_MASK( iSw, 16#006F ) = 16#0021 );
stSwitchedOn   := ( AND_MASK( iSw, 16#006F ) = 16#0023 );
stOpEnabled    := ( AND_MASK( iSw, 16#006F ) = 16#0027 );
stQuickStop    := ( AND_MASK( iSw, 16#006F ) = 16#0007 );
stFault        := ( AND_MASK( iSw, 16#004F ) = 16#0008 );

stTargetReached := ( AND_MASK( iSw, 16#0400 ) <> 0 ); (* bit 10 *)
stSetpointAck   := ( AND_MASK( iSw, 16#1000 ) <> 0 ); (* bit 12, mode 1 *)
stHomingDone    := ( AND_MASK( iSw, 16#1000 ) <> 0 ); (* bit 12, mode 6 *)
stHomingError   := ( AND_MASK( iSw, 16#2000 ) <> 0 ); (* bit 13, mode 6 *)
```

The masks are the ones of the CiA 402 state machine described in the DS402 chapter of this guide. Bits 12 and 13 change meaning with the mode of operation, which is why they are decoded twice under two different names.

4.4.3. Block 2—Enable and Motion Sequence

```
(* =====
APPLICATION STATE MACHINE
seq:  0 = idle          10..30 = power up
      40 = ready        50..52 = homing
      60..63 = position 90..92 = fault handling
===== *)

CASE seq OF

0:  (* --- Idle: no torque --- *)
    oCw := 16#0000;
    IF stFault THEN      seq := 90;
    ELSIF cmdEnable THEN seq := 10;
    END_IF;

10: (* --- Shutdown: towards Ready To Switch On --- *)
    oCw := 16#0006;
    IF stFault THEN      seq := 90;
    ELSIF stReady THEN   seq := 20;
    END_IF;

20: (* --- Switch On: power applied --- *)
    oCw := 16#0007;
    IF stFault THEN      seq := 90;
    ELSIF stSwitchedOn THEN seq := 30;
    END_IF;

30: (* --- Enable Operation: axis under torque --- *)
    oCw := 16#000F;
    IF stFault THEN      seq := 90;
    ELSIF stOpEnabled THEN seq := 40;
    END_IF;

40: (* --- Ready: waiting for a command --- *)
    oCw := 16#000F;
    IF stFault THEN      seq := 90;
    ELSIF NOT cmdEnable THEN seq := 0;
    ELSIF cmdHome THEN
        oMode := 6;      seq := 50;  (* Homing          *)
    ELSIF cmdMove THEN
        oMode := 1;      seq := 60;  (* Profile Position *)
    END_IF;

(* ----- HOMING, mode 6 ----- *)
50: oCw := 16#000F;
    IF iModeDisp = 6 THEN seq := 51;  (* mode accepted *)
    END_IF;

51: oCw := 16#001F;      (* bit 4 = start homing *)
    IF stHomingDone THEN seq := 52;
    ELSIF stHomingError OR stFault THEN
        seq := 90;
    END_IF;

52: oCw := 16#000F;      (* bit 4 low: handshake done *)
    cmdHome := FALSE;
    seq := 40;
```

```

(* ----- PROFILE POSITION, mode 1 ----- *)
60: oCw := 16#000F;
    IF iModeDisp = 1 THEN      seq := 61;
    END_IF;

61: (* oTargetPos, oProfVel, oProfAcc and oProfDec have already
    been written by the application.
    bit 4 = new set-point, bit 5 = change set immediately,
    add 16#0040 for a RELATIVE move. *)
    oCw := 16#003F;
    IF stSetpointAck THEN      seq := 62;
    ELSIF stFault THEN        seq := 90;
    END_IF;

62: oCw := 16#000F;           (* release bit 4 *)
    IF NOT stSetpointAck THEN  seq := 63;
    END_IF;

63: oCw := 16#000F;
    IF stTargetReached THEN
        cmdMove := FALSE;      seq := 40;
    ELSIF stFault THEN        seq := 90;
    ELSIF cmdStop THEN
        oCw := 16#010F;        (* bit 8 = Halt *)
    END_IF;

(* ----- FAULT HANDLING ----- *)
90: oCw := 16#0000;
    IF cmdReset THEN          seq := 91;
    END_IF;

91: oCw := 16#0080;           (* rising edge on bit 7 *)
    seq := 92;

92: oCw := 16#0000;           (* release bit 7 *)
    cmdReset := FALSE;
    IF stSwOnDisabled THEN    seq := 0;
    END_IF;

END_CASE;

```

The steps from 10 to 30 are what **MM_MS0** does inside itself on a Logix controller, steps 50 to 52 are **MM_MAH** or **MM_Profile_Homing_dfl**, and steps 60 to 63 are **MM_MAM** or **MM_Profile_Position_dfl**. The set-point handshake of Profile Position — raise bit 4, wait for bit 12, drop bit 4, wait for bit 10 — is the single most common reason for a motor that "does not start", and it cannot be skipped.



A fault reset leaves the drive disabled. After step 92 the sequence goes back to idle and a new **cmdEnable** is needed to bring the axis under control. If the cause of the alarm is still present the fault comes straight back: read **iAlarm** before resetting blindly.

4.4.4. Block 3—Writing the Outputs

```

(* =====
Typed variables -> ASSEMBLY 100
===== *)

```

```

DBS_0[0] := oCw; (* ControlWord *)
cpWrTp ( TRUE, oTargetPos, 0, DBS_0, 1, 2, FALSE ); (* TargetPos *)
DBS_0[3] := oTrqLimit; (* TorqueLimit *)
cpWrTv ( TRUE, oTargetVel, 0, DBS_0, 4, 2, FALSE ); (* TargetVel *)
DBS_0[6] := oTargetTrq; (* TargetTorque *)
DBS_0[7] := oMode; (* ModeOfOp *)
cpWrPv ( TRUE, oProfVel, 0, DBS_0, 8, 2, FALSE );
cpWrPa ( TRUE, oProfAcc, 0, DBS_0, 10, 2, FALSE );
cpWrPd ( TRUE, oProfDec, 0, DBS_0, 12, 2, FALSE );
cpWrNl ( TRUE, oNegLimit, 0, DBS_0, 14, 2, FALSE );
cpWrPl ( TRUE, oPosLimit, 0, DBS_0, 16, 2, FALSE );
cpWrHo ( TRUE, oHomeOffs, 0, DBS_0, 18, 2, FALSE );
DBS_0[20] := oHomeMethod; (* HomingMethod *)

```



ModeOfOperation and **HomingMethod** are single bytes followed by a padding byte, so writing the whole word is correct as long as the value is positive – which is the case for all the standard modes and homing methods. A negative homing method would sign-extend into the padding byte.

4.4.5. Initial Values

```

(* Executed once, on the first scan *)
IF NOT bInitDone THEN
  oProfVel := 3000; (* rpm *)
  oProfAcc := 3000; (* rpm/s *)
  oProfDec := 3000; (* rpm/s *)
  oTrqLimit := 1000; (* 1000 = 100 % of rated torque *)
  oHomeMethod := 37; (* homing in place *)
  oHomeOffs := 0;
  oNegLimit := -2147483648; (* position limits disabled *)
  oPosLimit := 2147483647;
  bInitDone := TRUE;
END_IF;

```



The position limits at their extreme values are disabled; if they are both left at 0 the motor will not move. The same applies to the profile references: with **ProfileVelocity**, **ProfileAccel** or **ProfileDecel** at zero the axis is enabled but no motion is possible.

4.5. Reading and Writing Parameters from CCW

The acyclic access described in *Writing Parameters with Async Messaging* works from a Micro850 as well, through the **MSG_CIPGENERIC** function block, and it is the same four manufacturer objects: COM (0x64), PAR (0x65), MISC (0x66) and CiA 402 (0x67). Taking again P160, *PosFactorNum* – class 0x65, instance 1, attribute 0xA0 – the configuration structures are:

```

CIPAPPCFG.Service := 16#0E; (* Get_Attribute_Single *)
(* 16#10 to write *)
CIPAPPCFG.Class := 16#65; (* PAR object *)
CIPAPPCFG.Instance := 1; (* P000...P255 *)
CIPAPPCFG.Attribute := 16#A0; (* P160 *)

```

```
CIPTARGETCFG.Path      := '2, 192.168.250.111'; (* port 2, drive IP *)
CIPTARGETCFG.CipConnMode := 0;                (* unconnected *)
CIPTARGETCFG.UcmmTimeout := 1000;             (* ms *)
```

Issue one message at a time and always wait for *Done* or *Error* before starting the next one. Remember that a parameter written this way lives in RAM: use the *Save parameters* attribute of the COM object, class 0x64 instance 1 attribute 0x10, to make it permanent.

On a 2080-LC50, which has no Class 1 scanner, the same block is the only way to reach the drive: the two assemblies can be read and written as attribute 3 of the Assembly object, service 16#0E on class 16#04 instance 101 for the status and service 16#10 on class 16#04 instance 100 for the commands, alternating the two messages.



Cyclic explicit messaging is a fallback, not an equivalent. A realistic update time is 50 to 200 ms and it is not guaranteed, so it is only suitable for commanded point-to-point moves with generous timeouts on the set-point handshake. For real axis control the 2080-L50E with a Class 1 connection is the right tool.

4.6. Commissioning and Diagnostics

Bring the axis up in this order, with the mechanics disconnected or at least free to move:

1. Download and check that the generic device reaches the *Running* state.
2. Go online and watch *iSw*, *iModeDisp*, *iAlarm* and *iDcVolt*. The StatusWord must be non-zero, the alarm must read 0 and the DC link voltage must be the expected one.
3. Confirm that the StatusWord decodes as *Switch On Disabled*, which is the normal state of a powered but disabled drive.
4. Set *cmdEnable* and follow the sequence 10 → 20 → 30 → 40. If it stalls, the StatusWord tells you exactly where.
5. Run a homing, then a first Profile Position move with a short distance and a much reduced *ProfileVelocity*.
6. Only then reconnect the mechanics and raise speeds and accelerations progressively.

Symptom	What to check
The connection never reaches <i>Running</i>	Assembly instances (100 / 101 / 102) and sizes (50 / 20 / 4). One byte of mismatch is enough for the drive to refuse the connection.
Connection error, extended status 0x0126, 0x0127 or 0x0128	Size errors on the configuration, O>>T and T>>O assembly respectively: they point straight at which of the three values is wrong.
Connection refused as already owned	The drive accepts one Exclusive Owner connection at a time. Close any session left open by another controller or by a configuration tool.
StatusWord always reads 0	The connection is formally up but no process data is arriving: check P176 on the drive, it must be 0 for the layout used here.
Half-written or nonsensical values	A COP offset is off by one word, or the word order is reversed: repeat the alignment check of the previous section.
The sequence stalls at <i>Switched On</i>	A hardware permit is missing on the drive, or a latched alarm is present: read <i>iAlarm</i> .

Symptom	What to check
Motion does not start in Profile Position	Incomplete bit 4 handshake, mode display not equal to 1, or a target outside the position limits.



Enabling the drive over EtherNet/IP is not a safety function. Emergency stop, STO and hardware limit switches must stay hard wired and independent of both the PLC and the network. Define as well the behaviour of the drive on loss of the Class 1 connection: that is configured on the drive, because in that condition the PLC is no longer writing anything.