

POWERLINK SET UP GUIDE

Riccardo Piccinini
Mini Motor S.P.A.
FBS-EPL-EN
Firmware Revision: 5.004 , Elker Cavina
Mini Motor S.P.A.
FBS-EPL-EN
Firmware Revision: 5.004

Version 4, 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

2. POWERLINK . . . **10**
- 2.1. Overview . . . 10
- 2.2. DS301 Communication Layer . . . 10
- 2.3. Object Dictionary . . . 12

3. Application Example . . . **19**
- 3.1. Tools . . . 19
- 3.2. Overview . . . 19
- 3.3. DBS Import . . . 20
- 3.4. Profile . . . 20
- 3.5. CSP . . . 27

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

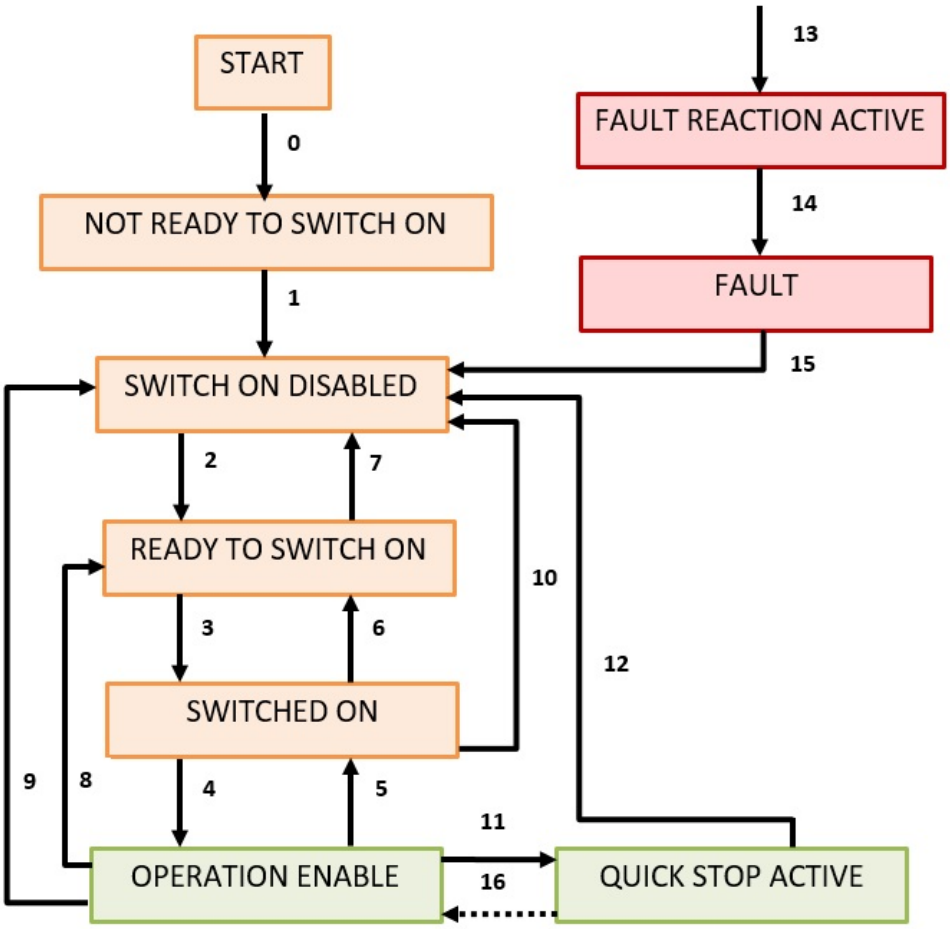
Whatever the mode, the interaction between the master (the controller) and the slave (the servo drive) always follows the same pattern:

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 *Ready To Switch On*
Switch On	`16#0007`	to *Switched On*
Enable Operation	`16#000F`	to *Operation Enabled* (motor runs)
Disable Voltage	`16#0000`	to *Switch On Disabled*
Quick Stop	`16#0002`	to *Quick Stop Active*
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_ModeOfOp` / `M1_ModeOfOpDisplay` 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.

2. POWERLINK

2.1. Overview



Note: available only for drivers equipped with CANopen/Modbus RTU, CANopen over EtherCAT or Ethernet POWERLINK. (---/---/C; ---/---/ETH; ---/---/EPL).

The present document describes the adherence of the Minimotor stack to the following specification:

CANopen

- CiA DS301 v4.02 (CANopen DLL)
- CiA DS402 v3.0.1.15 (servodrive profile)

CANopen over EtherCAT:

- ETG1000-2 v1.0.1 (physical layer)
- ETG1000-3 v1.0.1 (DLL services)
- ETG1000-4 v1.0.1 (DLL protocols)
- ETG1000-5 v1.0.1 (AL services)
- ETG1000-6 v1.0.1 (AL protocols)
- CiA DS402 v3.0.1.15 (servodrive)
- ETG6010 v1.1.0 (CiA 402 impl. directives)

Ethernet POWERLINK (EPL)

- EPSG DS 301 V1.2.0 (POWERLINK DLL)

The purpose of the present document is to describe what optional features are implemented by the servo-motor CANopen, CoE or EPL.

Applicable ---/---/--- software release: ≥v4.036

2.2. DS301 Communication Layer

This paragraph describes the details of the DS301 layer of the CANopen interface. It applies only to the CANopen over can-bus implementation.

2.2.1. Physical Layer

Supported baudrate:

- 1Mbps tq=71ns SamplingPoint=11tq
- 500Kbps tq=142ns SamplingPoint=12tq

- 250Kbps tq=284ns SamplingPoint=12tq
- 125Kbps tq=571ns SamplingPoint=12tq
- 50Kbps tq=1.43us SamplingPoint=12tq
- 20Kbps tq=3.57us SamplingPoint=12tq

2.2.2. SDO protocol

- Maximum supported object size: 32bit
- Expedited transfer supported SUPPORTED
- Segmented transfer: NOT SUPPORTED
- Block transfer: NOT SUPPORTED

2.2.3. TIME protocol

The time protocol is NOT supported.

2.2.4. PDO protocol

The implementation supports up to 4 Pdo TX and 4 Pdo RX; each object can map up to 8 objects; mapping can be done only using the whole size of the object (i.e. is not possible to map 8 bit of a 32 bit object).

The following transmit type are implemented for PDO TX:

Transmission Type	Description
0	PDO is emitted when changes occur after SYNC is received
1..240	PDO emitted after SYNC, depending on SYNC message occurrence (e.g. 1 = every SYNC, 2 = every 2 SYNC, etc.)
241..251	Not supported
252	PDO emitted after RTR reception * Data sampled on SYNC * Emitted after RTR
253	PDO emitted after RTR reception * Data sampled and emitted after RTR
254	Not supported
255	PDO emitted asynchronously Two features supported for emission control: * Inhibit time (emission only when PDO content changes) * Event timer (periodical asynchronous emission)

Defaults:

- inhibit time: 100ms
- event time: 0 (periodic emission on TT=255 disabled by default)

See application profile for default PDO configuration. ==== EMCY protocol

Emergency protocol is implemented for reporting alarms from the servodrive; to control the emission of

EMCY messages, the inhibit time (object 1015h) can be modified. See the application profile for detailed list of alarms. (See paragraph 8.3)

2.2.5. NMT protocol

The NMT implementation includes standard commands for NMT state machine management. For node alive checking, the NODE GUARDING protocol is implemented; the behaviour of node guarding is set by objects 100Ch (guard time) and 100Dh (life factor).

2.2.6. HEARTBEATING

Heartbeating Protocol is supported. Register 1016h defines the time in multiples of 1ms for the Heartbeat protocol consumer. Register 1017h determines the time in multiples of 1ms for the producer of the protocol. Object 1029h determines the behaviour when the error is triggered.

2.3. Object Dictionary

Object dictionary is divided in the following sections:

Range Objects	Functions
1000h .. 1FFFh	Communication profile area
2000h .. 21FFh	Manufacturer parameter area
3000h .. 3FFFh	Manufacturer specific object area
6000h .. 67FFh	Profiled objects

Other areas are unused; the object list is common between CANopen, CoE and EPL implementation.

2.3.1. Communication area

The following table shows the list of communication area objects; since some objects are relevant only for CANopen or CoE implementation, the table shows in which fieldbus the object is relevant.

Index	Description	CANopen	CoE	EPL
1000	Device type	X	X	X
1001	Error register	X	X	X
1003	Predefined error	X	X	
1006	Cycle length			X
100C	Guard time	X		
100D	Life factor	X		
1010	Store parameters, see Saving parameters to non-volatile memory	X	X	X
1014	EMCY cob-id	X		

Index	Description	CANopen	CoE	EPL
1015	EMCY inhibit time	X	X	
1016	Consumer Heartbeat Time	X		
1017	Producer Heartbeat Time	X		
1018	Identity object	X	X	X
1020	CFM Verify Configuration			X
1029	Error Behaviour	X		
1030	NMT Interface Group 0h			X
1300	SDO Sequ Layer Timeout			X
1400	PDO RX 1 – config [1]	X	X	X
1401	PDO RX 2 – config	X		
1402	PDO RX 3 – config	X		
1403	PDO RX 4 – config	X		
1600	PDO RX 1 – mapping [1]	X	X	X
1601	PDO RX 2 – mapping	X		
1602	PDO RX 3 – mapping	X		
1603	PDO RX 4 – mapping	X		
1800	PDO TX 1 – config [1]	X	X	X
1801	PDO TX 2 – config	X		
1802	PDO TX 3 – config	X		
1803	PDO TX 4 – config	X		
1A00	PDO TX 1 – mapping [1]	X	X	X
1A01	PDO TX 2 – mapping	X		
1A02	PDO TX 3 – mapping	X		
1A03	PDO TX 4 – mapping	X		
1C00	Sync manager types		X	
1C0A	DLL_CnCollision REC			X
1C0B	DLL_CnLoss REC			X
1C0C	DLL_CnLossSoC_REC			X
1C0D	DLL_CnLossSoA REC			X
1C0E	DLL_CnSpCJitter REC			X
1C0F	DLL_CnCRCError REC			X
1C10	Sync manager 0 config		X	
1C11	Sync manager 1 config		X	

Index	Description	CANopen	CoE	EPL
1C12	Sync manager 2 config		X	
1C13	Sync manager 3 config [1]		X	
	DLL_CnSocJitterRange [1]			X
1C14	DLL_CnLossOfSocTolerance			X
1C32	Output sync parameters		X	
1C33	Input sync parameters		X	
1E40	NWL_IpAddrTable_0h_REC			X
1E4A	NWL_IpGroup_REC			X
1F82	NMT_FeatureFlags_U32			X
1F83	NMT_EPLVersion_U8			X
1F8C	NMT_CurrNMTState_U8			X
1F93	NMT_EPLNodeID_REC			X
1F98	NMT_CycleTiming_REC			X
1F99	NMT_CNBasicEthernetTimeout_U32			X
1F9A	NMT_HostName_VSTR			X
1F9E	NMT_ResetCmd_U8			X



Saving parameters to non-volatile memory

Writing an object in the 2000h … 21FFh range 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 *Store parameters* object `1010h.1`.

2.3.2. Manufacturer parameter area

The object range 2000h … 21FFh allows to access servodrive parameters; the following table summarize the details of a parameter object:

Object details

Field	Value
Subidx	0
Object type	16bit, signed integer
Access	Read/Write
PDO mappable	NO

Object dictionary for manufacturer parameters

Index	Description
2000h	Parameter 0
2001h	Parameter 1
…	…
21FFh	Parameter 511

For details of the parameter meaning and encoding refer to the servodrive manual. IMPORTANT: Writing objects in the 2000h … 21FFh range will alter only the RAM image of the parameter; to actually save a parameter to non-volatile storage a 32 bit write of the value `65766173h` ("SAVE") to object `1010h.1` needs to be performed, as described in Saving parameters to non-volatile memory.

2.3.3. Manufacturer Specific Area

This area contains several object that allows to read/write manufacturer specific data.

Index	Access	Data type	PDO mappable	Description
3000h	RO	INT16	YES	Ain0 value (-32768 to 32768)
3001h	RO	UINT16	YES	Digital input bits 0–4
3002h	RW	UINT16	YES	Digital output bit0
3003h	RO	INT16	YES	Heatsink temperature (x0.1 °C)
3005h	RO	INT16	YES	Motor temperature (x0.1 °C)
3008h	RO	UINT16	YES	Actual alarm code (not EMCY)
3009h	RO	UINT	NO	Events (0–16, see section 6.1)
3010h	RO	UINT16	YES	Power stage tension (x0.1 V)
3011h	RO	INT16	YES	Motor IQ current (mA)
3012h	RW	UINT16	YES	Motor IQ limit (mA)
3014h	RO	INT16	YES	Output power (1W/lsb)
3020h	RW	INT32	YES	Touchprobe positioning offset
3021h	RW	UINT32	YES	Touchprobe positioning modulo
3030h	RO	UINT16	YES	Acceleration RMS modulus (milli-G)
3031h	RO	UINT16	YES	Acceleration RMS x (milli-G)
3032h	RO	UINT16	YES	Acceleration RMS y (milli-G)
3033h	RO	UINT16	YES	Acceleration RMS z (milli-G)
3034h	RO	UINT16	YES	Acceleration DC x (milli-G)
3035h	RO	UINT16	YES	Acceleration DC y (milli-G)
3036h	RO	UINT16	YES	Acceleration DC z (milli-G)

Index	Access	Data type	PDO mappable	Description
3037h	RO	INT16	YES	Instantaneous Acceleration x (milli-G)
3038h	RO	INT16	YES	Instantaneous Acceleration y (milli-G)
3039h	RO	INT16	YES	Instantaneous Acceleration z (milli-G)
3070h	RW	INT64	YES	Set point time, ns of the CIP Sync (PTP) clock. Used only by EtherNet/IP with CIP Sync
3071h	RO	INT64	YES	Position time, ns of the CIP Sync (PTP) clock. Used only by EtherNet/IP with CIP Sync
3072h	RO	UINT16	YES	Global time (CIP Sync) status: bit0 PTP synchronised, bit1 time valid, bit2 sync0 ok, bit3 locked to the time base, bit4 set point time received. Used only by EtherNet/IP with CIP Sync

2.3.4. Profiled Objects

The range 6000h … 67FFh of objects are defined as specified by CiA402; refer to the application area for further information.

Index	Access	Type	PDO mappable	Description
6040h	RW	U16	YES	Control word
6041h	RO	U16	YES	Status word
605Ah	RW	U16		Quickstop option code
605Bh	RW	U16		Shutdown option code
605Ch	RW	U16		Disable operation option code
605Dh	RW	U16		Halt option code
605Eh	RW	U16		Fault reaction option code
6060h	RW	S8	YES	Mode of operation
6061h	RO	S8	YES	Mode of operation display
6062h	RO	S32	YES	Position demand value
6064h	RO	S32	YES	Position actual value
6065h	RW	U32		Following error window
6066h	RW	U16		Following error time
6067h	RW	U32		Position window
6068h	RW	U16		Position window time
606Bh	RO	S32	YES	Velocity demand
606Ch	RO	S32	YES	Velocity actual value
606Dh	RW	U16		Velocity window

Index	Access	Type	PDO mappable	Description
606Eh	RW	U16		Velocity window time
606Fh	RW	U16		Velocity threshold
6070h	RW	U16		Velocity threshold time
6071h	RW	S16	YES	Target torque
6072h	RW	U16	YES	Max torque
6073h	RW	U16	YES	Max current
6074h	RO	S16	YES	Torque demand value
6075h	RO	U32	YES	Motor rated current
6076h	RO	U32	YES	Motor rated torque
6077h	RO	S16	YES	Torque actual value
6078h	RO	S16	YES	Current actual value
6079h	RO	U32	YES	DC link voltage
607Ah	RW	S32	YES	Target position
607Bh.0	RO	U8	YES	Position Range Limit – number of subindex
607Bh.1	RW	S32	YES	Negative limit
607Bh.2	RW	S32	YES	Positive limit
607Ch	RW	S32	YES	Home offset
607Dh.0	RO	U8	YES	Software position limit – number of subindex
607Dh.1	RW	S32	YES	Negative limit
607Dh.2	RW	S32	YES	Positive limit
607Eh	RW	U8		Polarity
607Fh	RO	U32	YES	Max speed
6080h	RO	U32	YES	Max speed (duplicate index)
6081h	RW	U32	YES	Profile velocity
6083h	RW	U32	YES	Profile acceleration
6084h	RW	U32	YES	Profile deceleration
6085h	RW	U32	YES	Quick stop deceleration
6087h	RW	U32	YES	Torque slope
6098h	RW	S8	YES	Homing method
6099h.0	RO	U8	YES	Homing speed – number of subindex
6099h.1	RW	U32	YES	Switch speed
6099h.2	RW	U32	YES	Index speed

Index	Access	Type	PDO mappable	Description
609Ah	RW	U32	YES	Homing acceleration
60B0h	RW	S32	YES	Position offset
60B1h	RW	S32	YES	Velocity offset
60B2h	RW	S16	YES	Torque offset
60B8h	RW	U16	YES	Touch-probe function
60B9h	RW	U16	YES	Touch-probe status
60BAh	RW	S32	YES	Touch-probe positive edge latched position
60BBh	RW	S32	YES	Touch-probe negative edge latched position
60C0h	RO	S16	NO	IpData submode selection
60C1h.0	RO	U8	YES	IpDataRecord – number of subindex
60C1h.1	RW	S32	YES	IpData record
60C2h.0	RO	U8		Interpolation time period – number of subindex
60C2h.1	RW	S8		Ip time units
60C2h.2	RW	S8		Ip time index
60F2h	RW	U16	YES	Position Option Code
60F4h	RO	U32	YES	Following Error
60FDh	RO	U32	YES	Digital inputs
60FEh.0	RO	U8	YES	Digital outputs – number of subindex
60FEh.1	RW	U32	YES	Physical outputs
60FEh.2	RW	U32	YES	Output mask
60FFh	RW	S32	YES	Target velocity
6502h	RO	U32	NO	Supported mode of operation

[1] PDO config and mapping have different encoding for CoE/CANopen and EPL. Please check DLL manual for details.

3. Application Example

3.1. Tools

POWERLINK Profile Example

POWERLINK CSP Example

DBS/FC BSI Interface Software

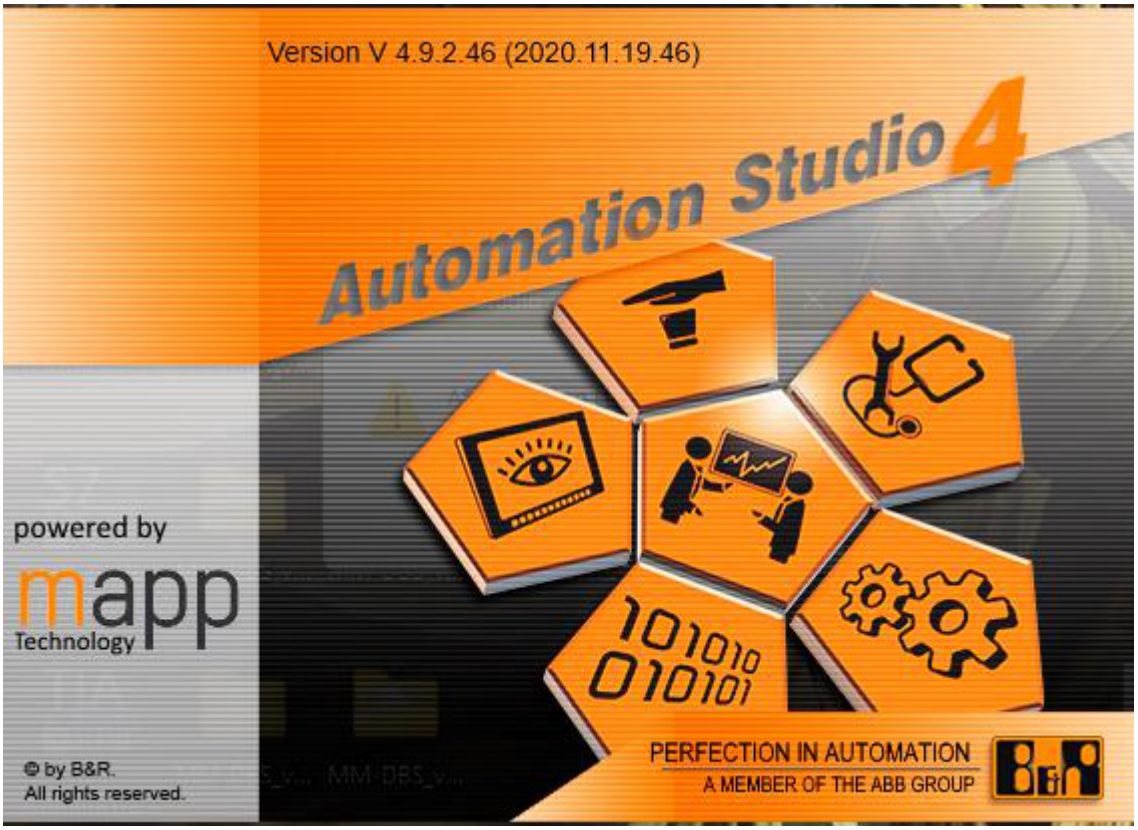
3.2. Overview

This document shows the creation of a Mini Motor Integrated Drive application using the POWERLINK communication interface; a B&R controller is configured and an application is written into the Automation Studio development environment.

References:

- B&R PLC 4PPC50 for the Profile example and 4PPC70 for the CSP example
- DBS/FC firmware version 5.004
- Automation Studio V 4.9.2.46

DBS and FC systems have the same electronics and firmware, so they follow the same procedure.



3.3. DBS Import

The first time using the DBS product, import the XDD descriptor into Automation Studio:

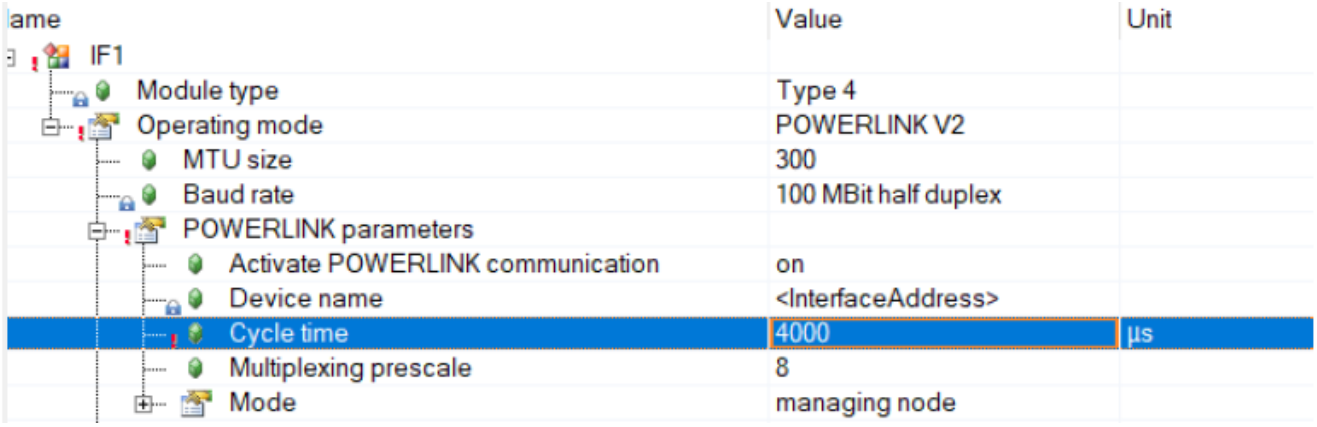
- Tools → Import Fieldbus device
- Search for MiniMot_DBS55_EPL.xdd in the DBS software distribution and press OK
- In the catalog, under POWERLINK devices, the Minimotor DBS device should now be present

The next steps depend on how you want to control the motor: as a simple node using the CANopen DS402 Profile Velocity, Torque, Position and Homing, or in interpolated mode, using — as in the example — the motor with the ACP10 subsystem.

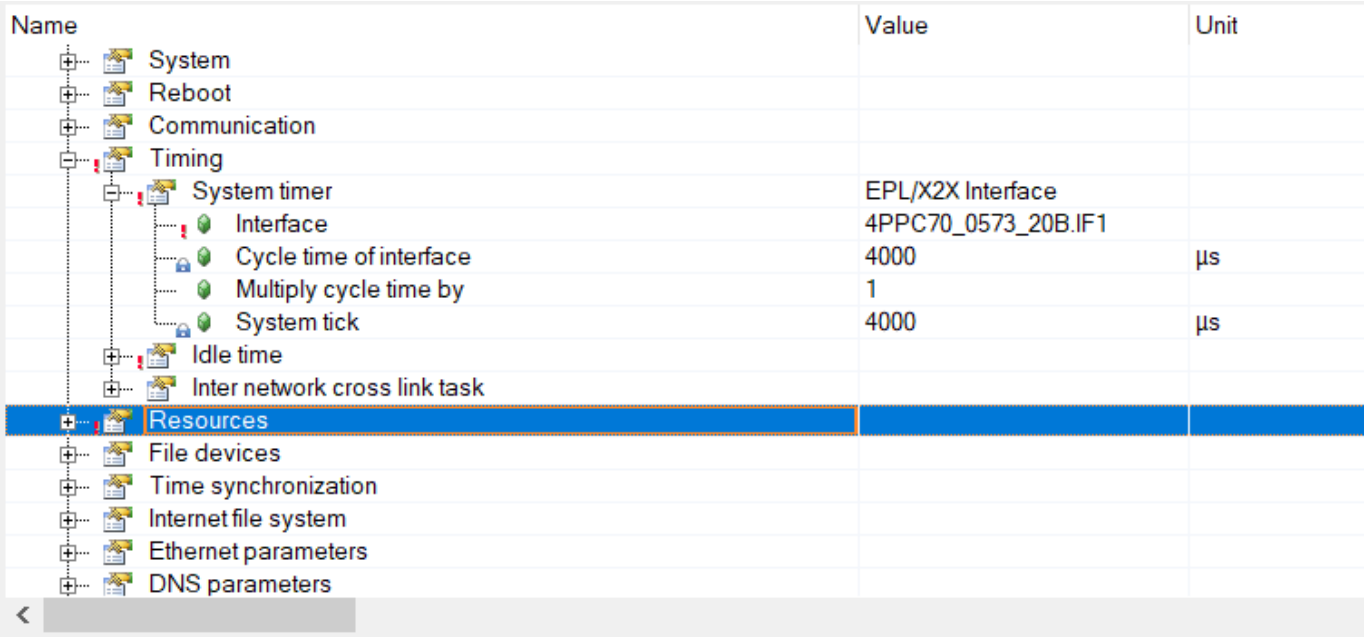
3.4. Profile

3.4.1. Initial Steps

- Create a new project.
- Configure the POWERLINK cycle time.



- Sync the CPU system timer with the POWERLINK time.



- Add the DBS using *Add Hardware Module* and configure the channels. Channel mappings depend on the specific requirement of the application. For this example, we map the following.

Channel Name	Process Variable	Data Type
ModuleOk	::M1_ok	BOOL
ActualAlarm_I3008		UINT
DC_LinkVoltage_0_1V_lsb_I3010	::M1_VDC	UINT
ControlWord_I6040Out	::M1_Cw	UINT
StatusWord_I6041	::M1_SW	UINT
ModesOfOperation_I6060Out	::M1_ModeofOp	SINT
ModesOfOperationDisplay_I6061	::M1_ModeofOpDisplay	SINT
PositionActualValue_I6064	::M1_ActualPos	DINT
VelocityActualValue_I606C	::M1_Actualvel	DINT
TargetTorque_I6071Out	::M1_TargetTorque	INT
MaxCurrent_I6073Out	::M1_TorqueLimit	UINT
TorqueActualValue_I6077	::M1_ActualTorque	INT
DC_linkCircuitVoltage_I6079		UDINT
TargetPosition_I607AOut	::M1_TargetPos	DINT
HomeOffset_I607COut	::M1_HomingOffset	DINT
NegativeLimit_I607D_S01Out	::M1_NegLim	DINT
PositiveLimit_I607D_S02Out	::M1_PosLim	DINT
ProfileVelocity_I6081Out	::M1_ProfileVel	UDINT
ProfileAcceleration_I6083Out	::M1_ProfileAcc	UDINT
ProfileDeceleration_I6084Out	::M1_ProfileDec	UDINT
HomingMethod_I6098Out	::M1_HomMethod	USINT
TargetVelocity_I60FFOut	::M1_TargetVelocity	DINT

3.4.2. Sample Application

In the sample application we create a program that can run the motor in speed mode, in position mode towards a single position, and in position mode between two different positions.

To do so, in the Global.typ we build a Machine_State type.

```
TYPE
    Machine_State :
        (
        Idle,
        Velocity,
        Position,
        Auto_Enter,
        Auto_A,
        Auto_B,
        Auto_Moving,
        Auto_Wait,
        Home,
        Stop
        );
END_TYPE
```

In the Global variables we have two sets of variables. The GUI_ set is used in the user interface, while the M1_ set is linked to the motor channels.

```
VAR
```

```
    Case_OP : Machine_State;
    GUI_auto : BOOL;
    GUI_Dir : BOOL;
    GUI_en_hom : BOOL;
    GUI_en_pos : BOOL;
    GUI_en_vel : BOOL;
    GUI_Home : BOOL;
    GUI_hom_done : INT;
    GUI_m1_ok : INT;
    GUI_new_set : BOOL;
    GUI_pos : DINT;
    GUI_pos_reach : INT;
    GUI_Run : BOOL;
    GUI_Stop : BOOL;
    GUI_tpos : DINT;
    GUI_tposA : DINT;
    GUI_tposB : DINT;
    GUI_trq : REAL;
    GUI_tvel : DINT;
    GUI_vdc : REAL;
    GUI_vel : DINT;
    M1_ActualAlarm : UINT;
    M1_ActualPos : DINT;
    M1_ActualTorque : INT;
    M1_Actualvel : DINT;
    M1_Cw : UINT;
    M1_HomingOffset : DINT;
    M1_HomMethod : USINT;
    M1_ModeofOp : SINT;
    M1_ModeofOpDisplay : SINT;
    M1_NegLim : DINT;
    M1_ok : BOOL;
    M1_PosLim : DINT;
    M1_ProfileAcc : UDINT;
    M1_ProfileDec : UDINT;
    M1_ProfileVel : UDINT;
    M1_SW : UINT;
    M1_TargetPos : DINT;
    M1_TargetTorque : INT;
    M1_TargetVelocity : DINT;
    M1_TorqueLimit : UINT;
    M1_VDC : UINT;
END_VAR
```

Link the M1 variables to the channels as shown in the picture before. In the program part, under Variables.Var, write the following variables:

```
VAR
    Counter : USINT;
    Arrived_A : BOOL;
    SetAck : BOOL;
    Arrived_B : BOOL;
END_VAR
```

3.4.3. Code

In the _INIT code part, set the defaults:

```
PROGRAM _INIT
    (* Insert code here *)
    Case_OP := Idle;
    M1_PosLim := 2147483647;
    M1_NegLim := -2147483648;
    M1_TorqueLimit := 1000;
    Counter := 0;
END_PROGRAM
```

The _CYCLIC code implements the state machine that drives the motor through the control word / status word dialogue:

```
CASE Case_OP OF

    Idle:
        M1_Cw := 16#06;
        GUI_new_set := FALSE;
        GUI_hom_done := 0;
        Counter := 0;
        Arrived_A := 0;
        Arrived_B := 0;
        SetAck := 0;

        IF (NOT GUI_Run) THEN
            GUI_en_vel := FALSE;
            GUI_en_pos := FALSE;
            GUI_auto := FALSE;
        END_IF;

        (* Exit *)
        IF (GUI_en_hom) THEN
            Case_OP := Home;
        END_IF;
        IF (GUI_Run) THEN
            IF (GUI_en_vel) THEN
                Case_OP := Velocity;
            END_IF;
            IF (GUI_en_pos) THEN
                Case_OP := Position;
            END_IF;
            IF (GUI_auto) THEN
                Case_OP := Auto_Enter;
            END_IF;
        END_IF;

    Velocity:
        M1_ModeofOp := 3;
        IF (M1_ModeofOpDisplay = 3) THEN
            M1_Cw := 16#0F;
        ELSE
            M1_Cw := 16#06;
        END_IF;
        (* setpoint *)
        IF (GUI_Dir) THEN
            M1_TargetVelocity := -GUI_tvel;
        ELSE
            M1_TargetVelocity := GUI_tvel;
        END_IF;
        (* Exit *)
```

```
        IF (NOT GUI_en_vel) THEN
            Case_OP := Idle;
        END_IF;

    Position:
        M1_ModeofOp := 1;
        IF (M1_ModeofOpDisplay = 1) THEN
            M1_Cw := 16#0F;
        ELSE
            M1_Cw := 16#06;
        END_IF;
        (* new set *)
        IF (GUI_pos < GUI_tpos-2 OR GUI_pos > GUI_tpos+2) THEN
            Counter := Counter + 1;
            IF (Counter > 3) THEN
                M1_Cw := 16#1F;
            ELSIF ((M1_SW AND 16#1000) = 16#1000) THEN
                M1_Cw := 16#0F;
                Counter := 0;
            END_IF;
        END_IF;
        M1_TargetPos := GUI_tpos;
        M1_ProfileVel := GUI_tvel;
        (* Exit *)
        IF (NOT GUI_en_pos) THEN
            Case_OP := Idle;
        END_IF;

    Auto_Enter:
        M1_ModeofOp := 1;
        IF (M1_ModeofOpDisplay = 1) THEN
            M1_Cw := 16#0F;
        ELSE
            M1_Cw := 16#06;
        END_IF;
        Counter := Counter + 1;
        (* Exit *)
        IF (Counter > 3) THEN
            IF (Arrived_A) THEN
                Case_OP := Auto_B;
                Counter := 0;
            ELSIF (Arrived_B) THEN
                Case_OP := Auto_A;
                Counter := 0;
            ELSE
                Case_OP := Auto_A;
                Counter := 0;
            END_IF;
        END_IF;
        IF (NOT GUI_auto) THEN
            Case_OP := Idle;
        END_IF;

    Auto_A:
        M1_TargetPos := GUI_tposA;
        M1_ProfileVel := GUI_tvel;
        M1_Cw := 16#1F;
        IF (M1_SW.12 AND M1_Cw = 16#1F) THEN
            SetAck := TRUE;
        END_IF;
        (* Exit *)
```

```
        IF (SetAck) THEN
            Case_OP := Auto_Moving;
            Arrived_B := FALSE;
        END_IF;
        IF (NOT GUI_auto) THEN
            Case_OP := Idle;
        END_IF;

    Auto_B:
        M1_TargetPos := GUI_tposB;
        M1_ProfileVel := GUI_tvel;
        M1_Cw := 16#1F;
        IF (M1_SW.12 AND M1_Cw = 16#1F) THEN
            SetAck := TRUE;
        END_IF;
        (* Exit *)
        IF (SetAck) THEN
            Case_OP := Auto_Moving;
            Arrived_A := FALSE;
        END_IF;
        IF (NOT GUI_auto) THEN
            Case_OP := Idle;
        END_IF;

    Auto_Moving:
        M1_Cw := 16#0F;
        Counter := Counter + 1;
        SetAck := FALSE;
        IF (Counter > 2) THEN
            IF (M1_SW.10 AND M1_Actualvel = 0) THEN
                IF (GUI_tposA-2 < M1_ActualPos AND M1_ActualPos < GUI_tposA+2) THEN
                    Arrived_A := TRUE;
                ELSIF (GUI_tposB-2 < M1_ActualPos AND M1_ActualPos < GUI_tposB+2) THEN
                    Arrived_B := TRUE;
                END_IF;
            END_IF;
        END_IF;
        (* Exit *)
        IF (Arrived_A OR Arrived_B) THEN
            Counter := 0;
            Case_OP := Auto_Wait;
        END_IF;
        IF (NOT GUI_auto) THEN
            Case_OP := Idle;
        END_IF;

    Auto_Wait:
        Counter := Counter + 1;
        IF (Counter > 6) THEN
            Case_OP := Auto_Enter;
        END_IF;
        IF (NOT GUI_auto) THEN
            Case_OP := Idle;
        END_IF;

    Home:
        M1_ModeofOp := 6;
        M1_HomMethod := 35;
        IF (M1_ModeofOpDisplay = 6) THEN
            M1_Cw := 16#1F;
        ELSE
```

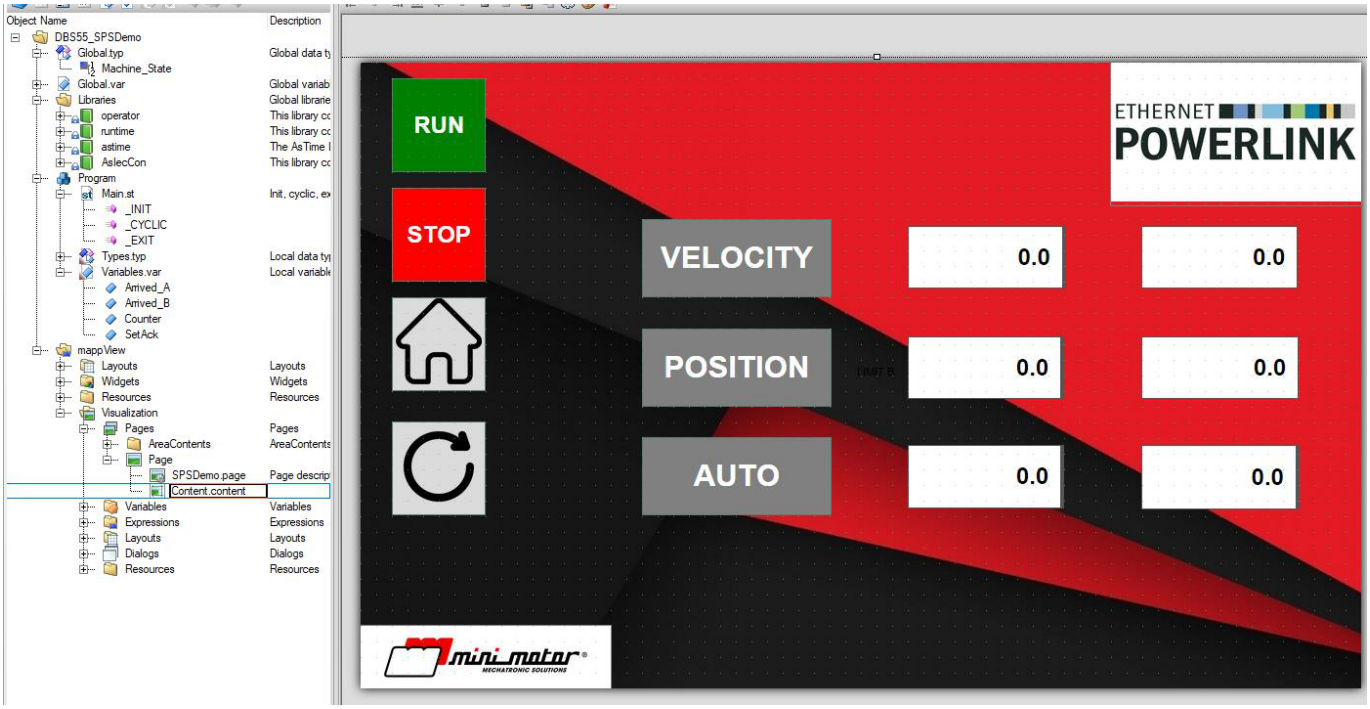
```
            M1_Cw := 16#06;
        END_IF;
        (* Status *)
        IF ((M1_SW AND 16#400) = 16#400) THEN
            GUI_hom_done := 10; (* green - DONE *)
        ELSIF ((M1_SW AND 16#2000) = 16#2000) THEN
            GUI_hom_done := 45; (* red - ERROR *)
        ELSE
            GUI_hom_done := 41; (* orange - IN PROGRESS *)
        END_IF;
        (* Exit *)
        IF (GUI_hom_done = 10) THEN
            GUI_en_hom := FALSE;
            Case_OP := Idle;
        END_IF;

    Stop:
        M1_Cw := 16#06;
        GUI_new_set := FALSE;
        GUI_hom_done := 0;
        GUI_en_pos := FALSE;
        GUI_en_vel := FALSE;
        GUI_en_hom := FALSE;
        GUI_auto := FALSE;
        GUI_Stop := FALSE;
        Case_OP := Idle;

END_CASE;

IF (GUI_Stop) THEN
    GUI_Run := FALSE;
    Case_OP := Stop;
END_IF;
```

The program comes with a GUI suitable for a 4PPC50.

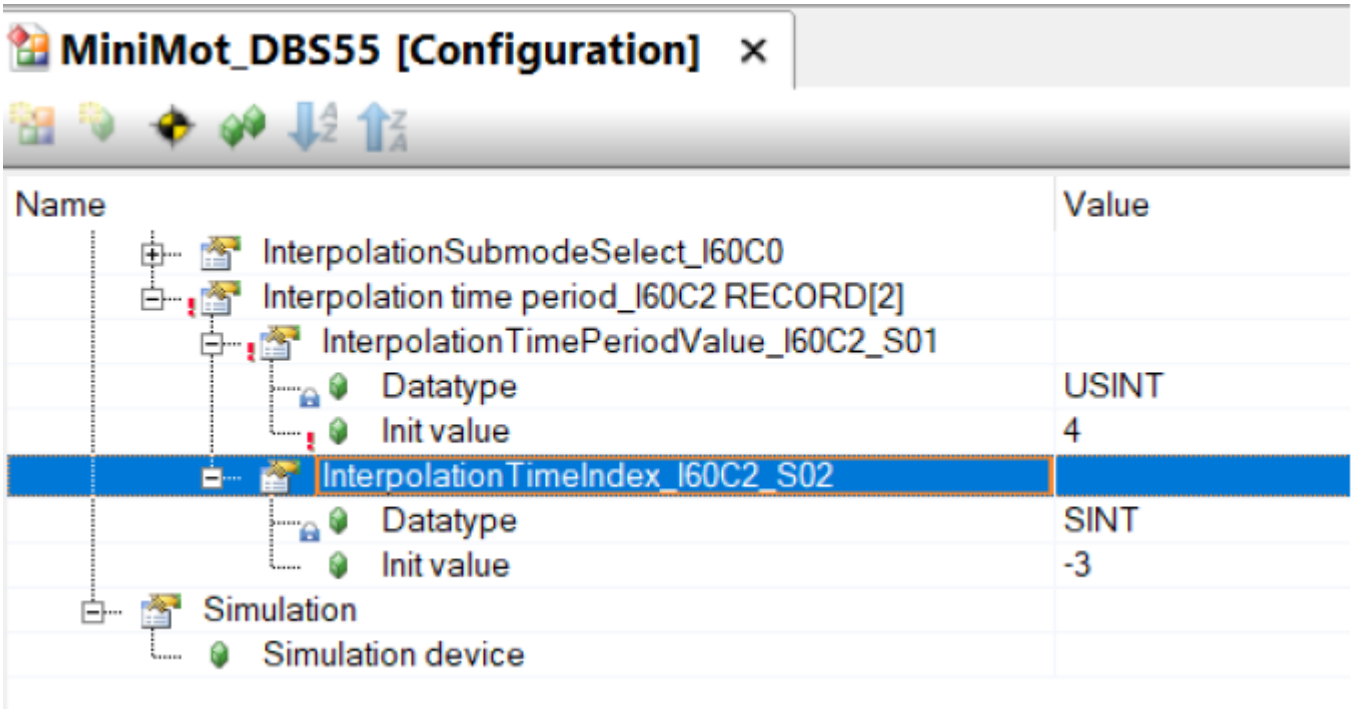


3.5. CSP

The example shows how to interface a DBS integrated motor, CSP enabled, with the B&R platform using the ACP10 library.

3.5.1. Initial Steps

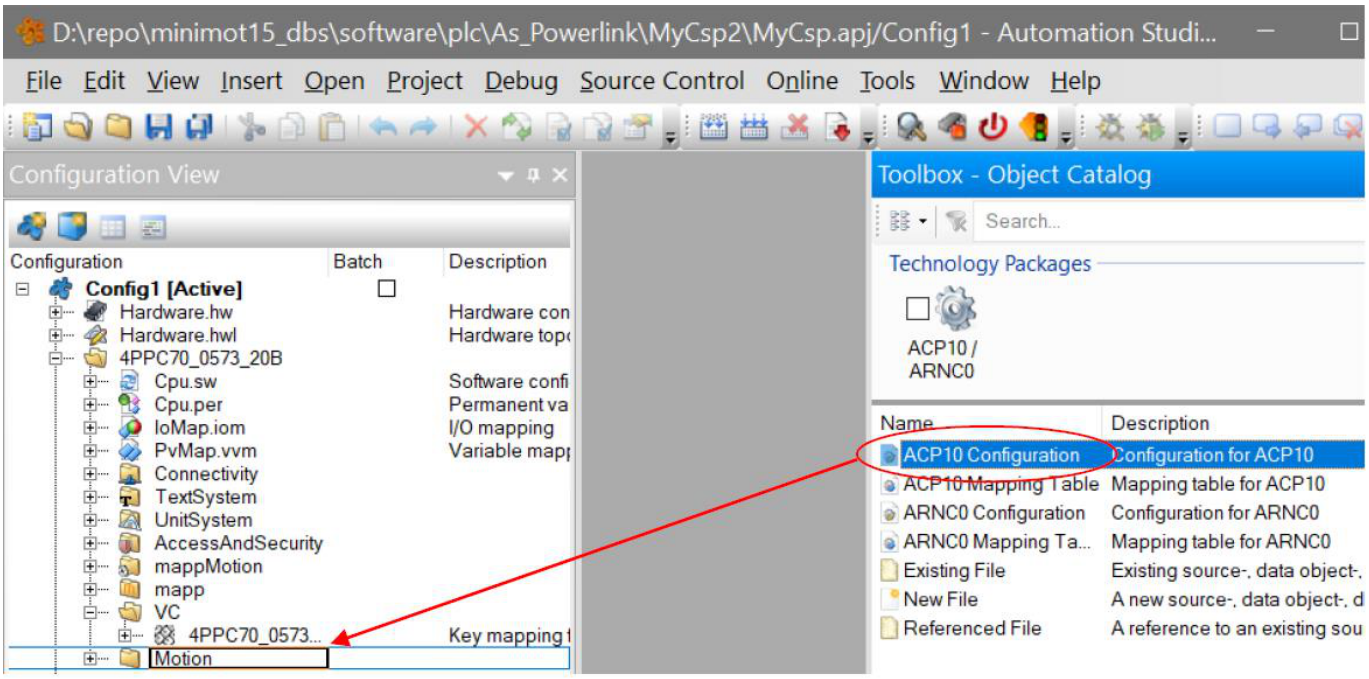
- Create a new project.
- Configure the POWERLINK cycle time.
- Sync the CPU system timer with the POWERLINK time.
- Add the DBS using *Add Hardware Module* and configure the channels. The minimum required objects for the axis function are:
  - Write: ControlWord, Mode of Operation, Target Position
  - Read: Status Word, Mode of Operation Display, Position Actual Value
- Set up the interpolation time to match the POWERLINK cycle rate. The setup in the picture is 4 ms.



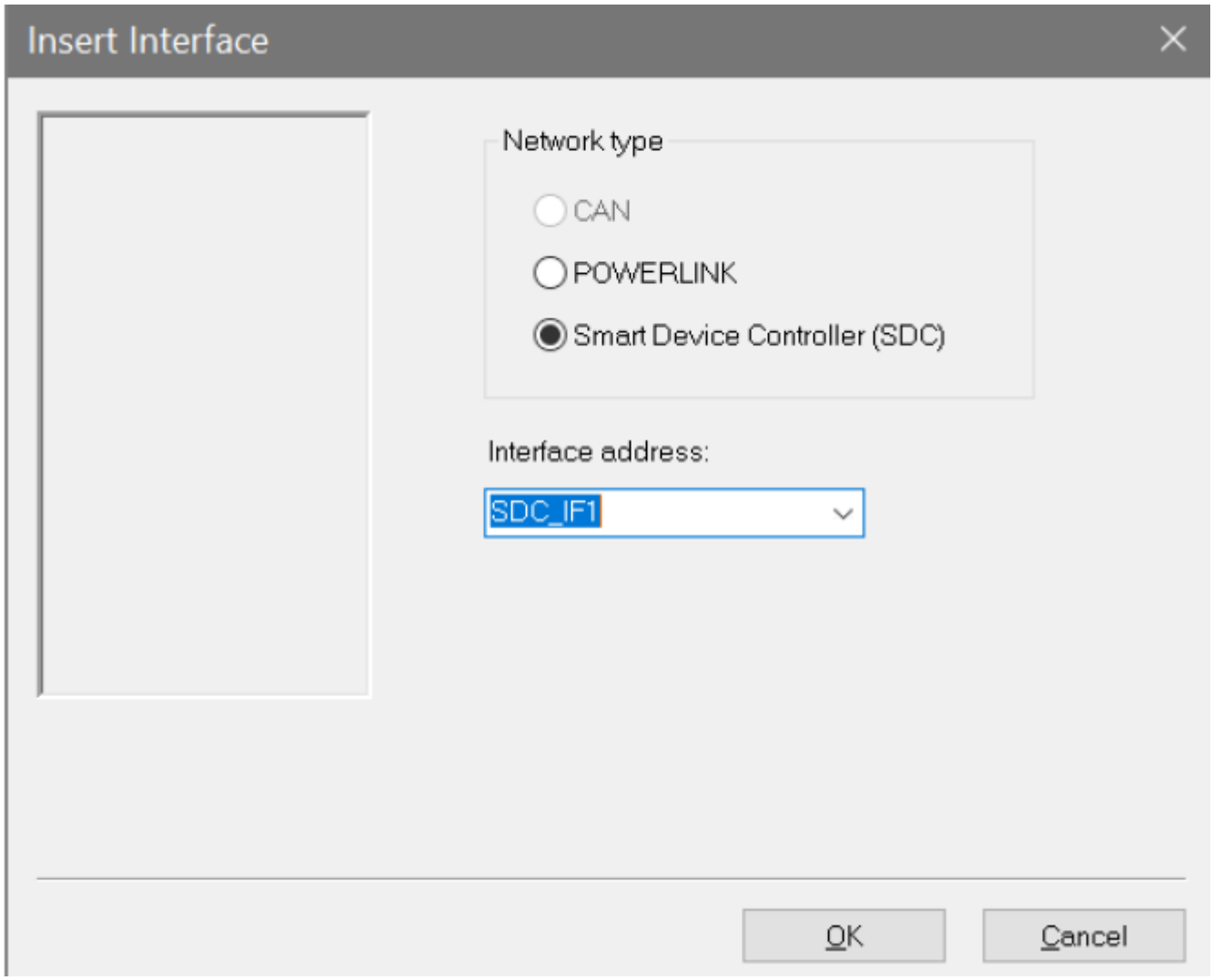
- Add the relevant variables in the Global.Var and link them to the DBS I/O mapping. In the provided example there are more variables linked, to better monitor the DBS system.

3.5.2. Add Axis to the NDC

- Add the motion folder in the configuration of the CPU, adding the ACP10 Configuration.

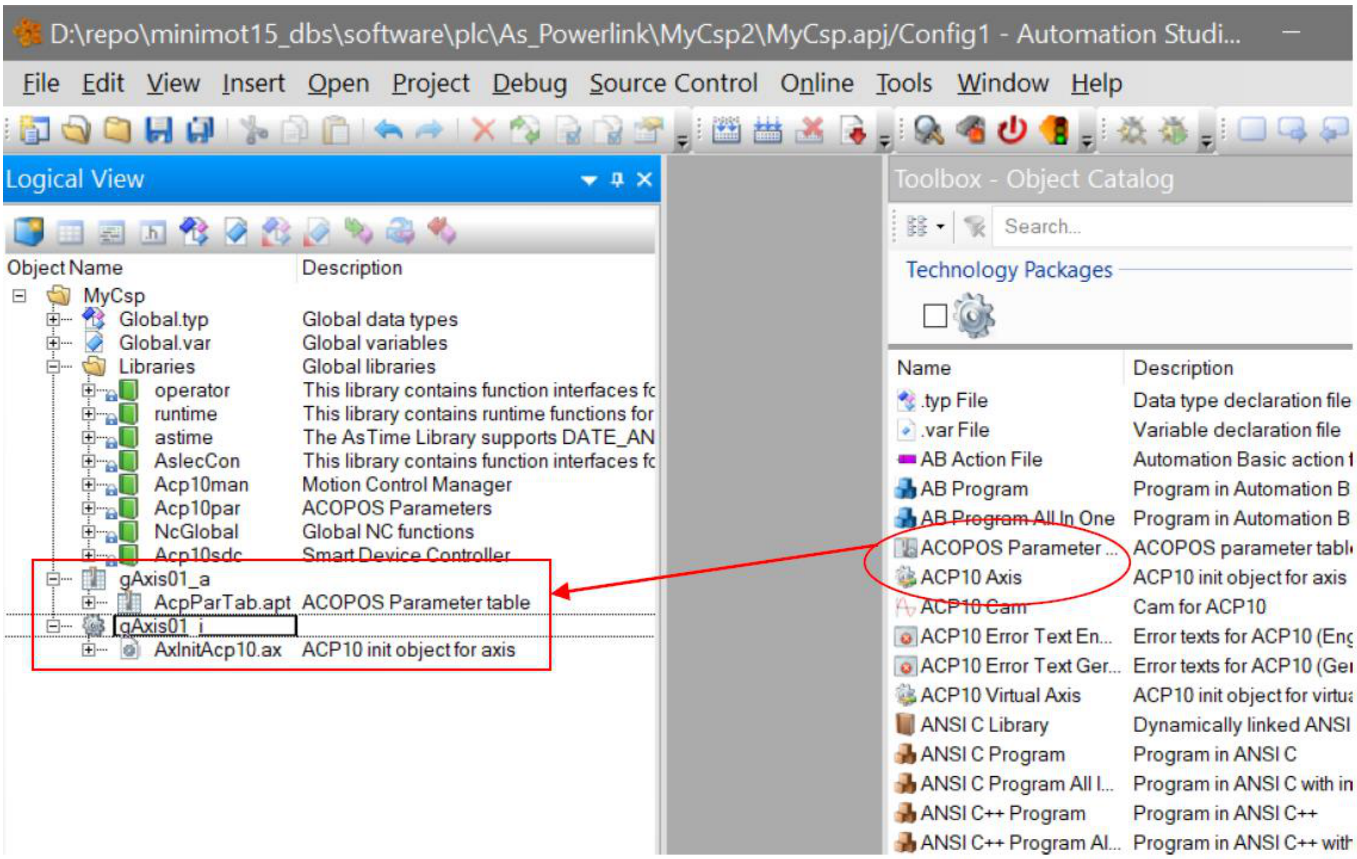


- Open Acp10cfg.ncc, right click in the blank space and select *Insert Interface…*, then select Smart Device Controller and confirm with OK.

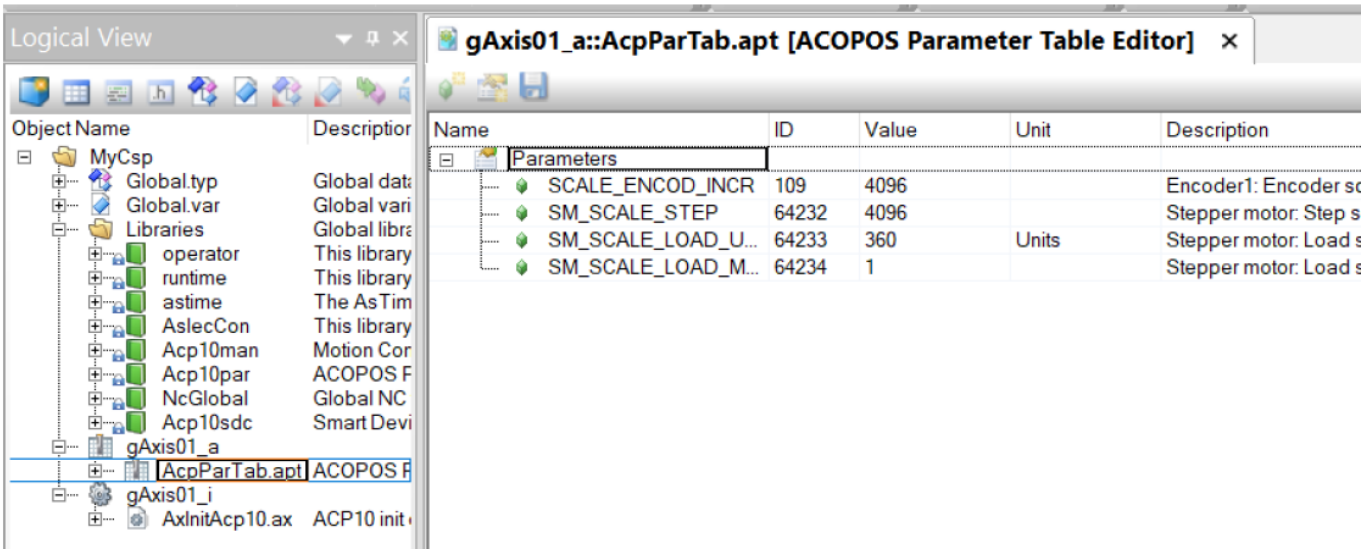


- In the Logical tab, select the root node and add an *ACOPOS parameter table* and an *ACP10 Axis*; then

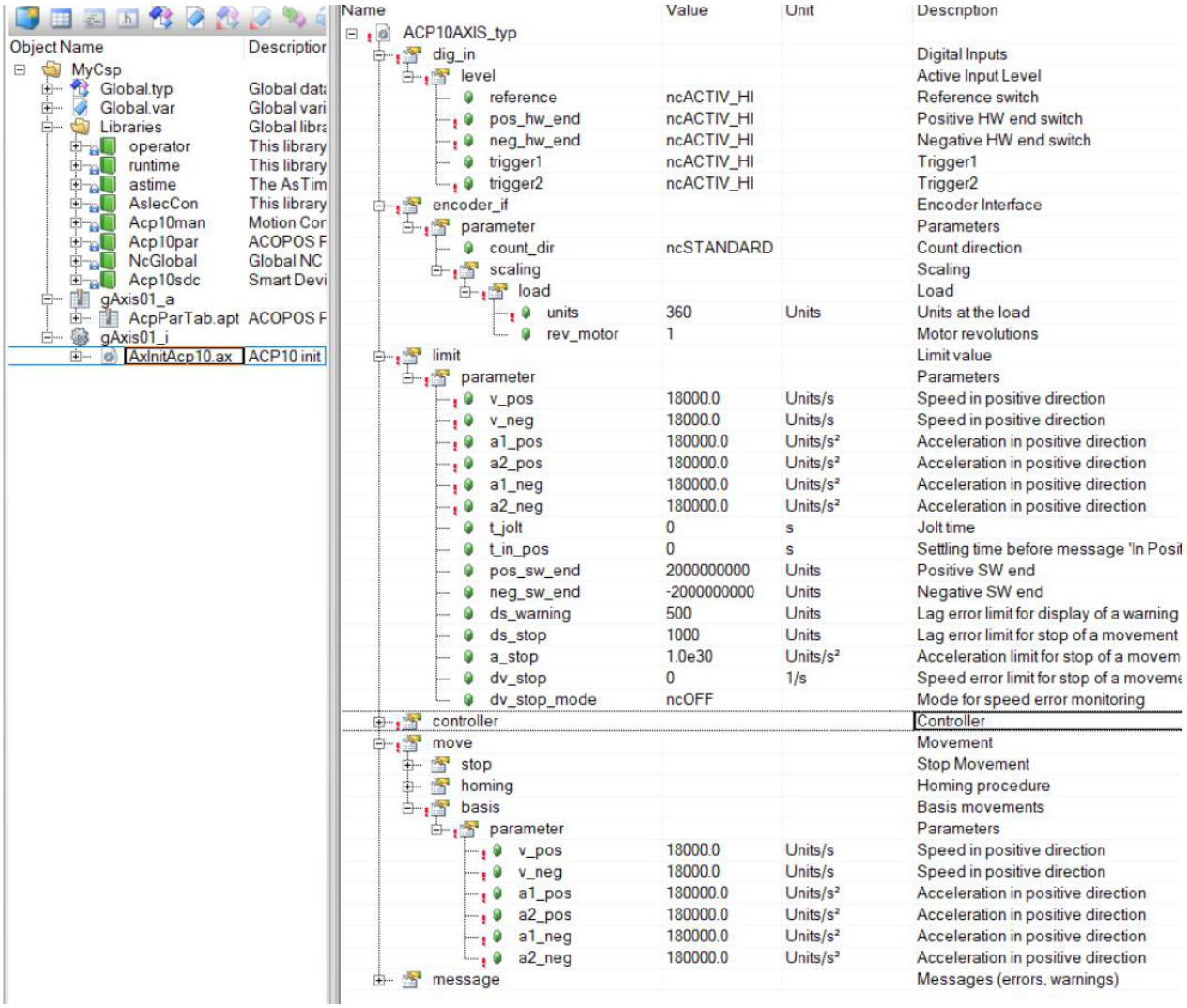
rename them as gAxis01_a and gAxis01_i.



- In the Configuration tab, open Acp10map1.ncm, right click and select *Add NC Object*, then compile it as:
  - Object name: gAxis01
  - PLC Address: SDC_IF1.ST1
  - Nc object: ncAxis
  - Channel: 1
  - Init parameter gAxis01_i
  - ACOPOS Parameter gAxis01_a
- Open gAxis01_a parameters and set up the following:
  - SCALE_ENCOD_INCR (ID=109) 4096 (pulses per revolution of DBS)
  - SM_SCALE_STEP (ID=64232) 4096 (pulses per revolution of DBS)
  - SM_SCALE_LOAD_UNITS (ID=64233) 360 (units per revolution, degrees)
  - SM_SCALE_LOAD_MOTREV (ID=64234) 1 (units referred to one revolution)

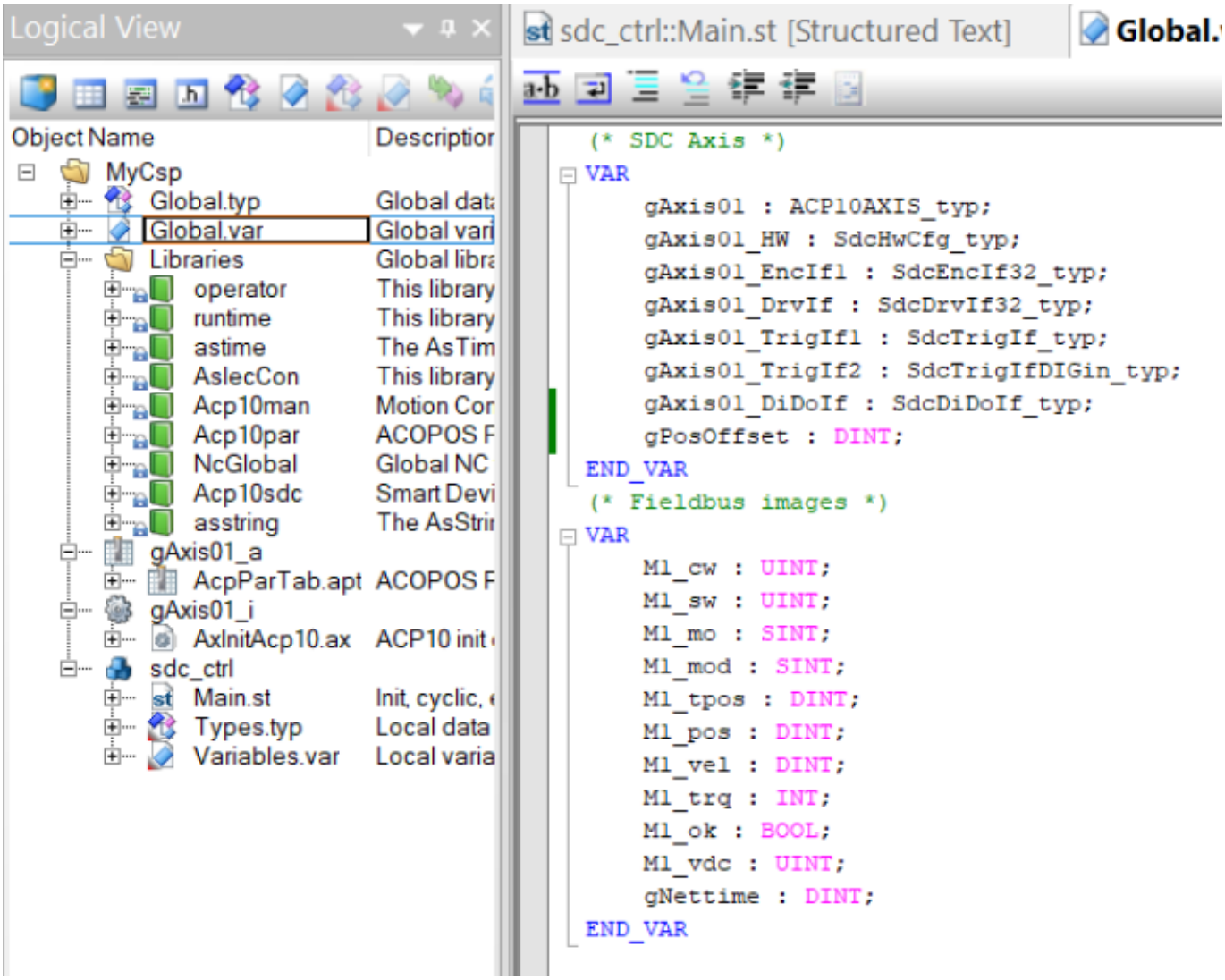


- Open the gAxis01_i parameter tree and set the following parameters:
  - dig_in level: ncACTIVE_HI
  - encoder_if\parameter\scaling\load\units: 360 (degrees per rev)
  - limit\parameter\v_*: 18000 (degrees/sec, 3000 rpm)
  - limit\parameter\a*: 180000 (degrees/sec^2, 30000 rpm/s)
  - move\basis\parameter\v_*: 18000 (degrees/sec, 3000 rpm)
  - move\basis\parameter\a*: 180000 (degrees/sec^2, 30000 rpm/s)



3.5.3. SDC – DS402 Connection

- In the Global.var add the SDC axis variables.
- Add a new *ST Program all in one* and rename it sdc_ctrl.
- Add the B&R library asstring.
- Open Cpu.sw and ensure that sdc_ctrl runs in the *Cyclic 1* at 4 ms (it needs to run synchronously to the POWERLINK data exchange).



3.5.4. Code

In the _INIT code part, add the configuration of the SDC axis; the code links all the sub-structures.

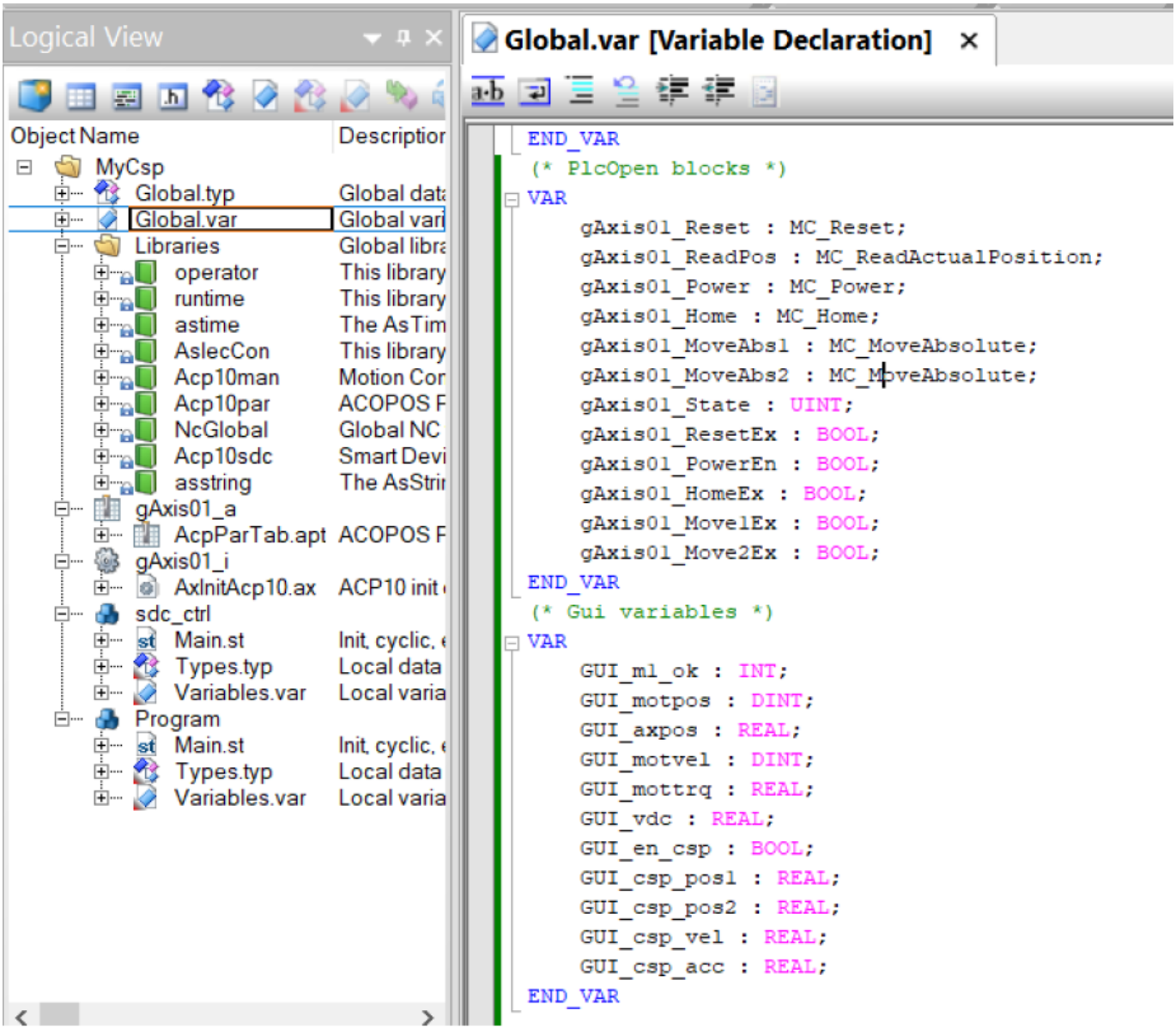
```
(* initialize EncIf1 *)
gAxis01_HW.EncIf1_Typ := ncSDC_ENC32;
strcpy( ADR(gAxis01_HW.EncIf1_Name[0]), ADR('gAxis01_EncIf1') );
(* initialize DrvIf *)
gAxis01_HW.DrvIf_Typ := ncSDC_DRVSM32;
strcpy( ADR(gAxis01_HW.DrvIf_Name[0]), ADR('gAxis01_DrvIf') );
(* initialize TrigIf1 *)
gAxis01_HW.TrigIf1_Typ := ncSDC_TRIG;
strcpy( ADR(gAxis01_HW.TrigIf1_Name[0]), ADR('gAxis01_TrigIf1') );
(* initialize TrigIf2 *)
gAxis01_HW.TrigIf2_Typ := ncSDC_TRIGDIGin;
strcpy( ADR(gAxis01_HW.TrigIf2_Name[0]), ADR('gAxis01_TrigIf2') );
(* initialize DiDoIf *)
gAxis01_HW.DiDoIf_Typ := ncSDC_DIDO;
strcpy( ADR(gAxis01_HW.DiDoIf_Name[0]), ADR('gAxis01_DiDoIf') );
```

The _CYCLIC code is composed of two blocks. The first simulates the life counters in order to comply with the liveness management from the numerical control; the OK flags are linked to the ModuleOK flag coming from POWERLINK.

```
(* life counter simulation *)
gAxis01_DrvIf.iLifeCnt := gAxis01_DrvIf.iLifeCnt + 1;
gAxis01_EncIf1.iLifeCnt := gAxis01_EncIf1.iLifeCnt + 1;
gAxis01_EncIf1.iActTime := DINT_TO_INT(gNettime AND 16#FFFF);
gAxis01_DiDoIf.iLifeCntDriveEnable := gAxis01_DiDoIf.iLifeCntDriveEnable + 1;
gAxis01_DiDoIf.iLifeCntDriveReady := gAxis01_DiDoIf.iLifeCntDriveReady + 1;
gAxis01_DiDoIf.iLifeCntNegHwEnd := gAxis01_DiDoIf.iLifeCntNegHwEnd + 1;
gAxis01_DiDoIf.iLifeCntPosHwEnd := gAxis01_DiDoIf.iLifeCntPosHwEnd + 1;
gAxis01_DiDoIf.iLifeCntReference := gAxis01_DiDoIf.iLifeCntReference + 1;
gAxis01_TrigIf1.iLifeCnt := gAxis01_TrigIf1.iLifeCnt + 1;
gAxis01_TrigIf2.iLifeCnt := gAxis01_TrigIf2.iLifeCnt + 1;
(* drive ready emulation *)
gAxis01_EncIf1.iEncOK := M1_ok;
gAxis01_DiDoIf.iDriveReady := M1_ok;
gAxis01_DrvIf.iDrvOK := M1_ok;
```

The second part of the _CYCLIC code is the actual DS402 glue, where the control word is driven from the gAxis enable and the target is linked to the gAxis output reference.

```
(* ds402 linking *)
gAxis01_EncIf1.iActPos := M1_pos; (* position actual value to axis *)
gAxis01_DrvIf.iStatusEnable := gAxis01_DrvIf.iDrvOK;
M1_mo := 8; (* CSP *)
IF (M1_mod = 8) THEN
    IF ((M1_sw AND 16#4F) = 16#08) THEN
        M1_cw := 16#80; (* FAULT ACK *)
    ELSE
        IF gAxis01_DiDoIf.oDriveEnable THEN
            M1_cw := 16#0F; (* ENABLE *)
        ELSE
            M1_cw := 16#06; (* DISABLE *)
            gPosOffset := M1_pos - gAxis01_DrvIf.oSetPos;
        END_IF
    END_IF
ELSE
    M1_cw := 16#06; (* DISABLE *)
END_IF
(* motor target position *)
M1_tpos := gAxis01_DrvIf.oSetPos + gPosOffset; (* position target to drive *)
```



3.5.5. Sample Application

The provided example contains a sample application that enables the drive, does homing and continuously moves the motor between two position points.

- Add the B&R library Acp10_MC.
- In the Global.var add the PLCopen block variables (MC_Reset, MC_Power, MC_Home, MC_MoveAbsolute, etc.).

In the _INIT code insert the parameter defaults:

```
GUI_csp_acc := 36000.0;
GUI_csp_vel := 3600.0;
GUI_csp_pos1 := 0.0;
GUI_csp_pos2 := 3600.0;
```

In the _CYCLIC the main part is the PLCopen blocks for driving the motor and the state machine to drive them.

```
gAxis01_Reset(Axis := ADR(gAxis01), Execute := gAxis01_ResetEx);
```

```
gAxis01_Power(Axis := ADR(gAxis01), Enable := gAxis01_PowerEn);
gAxis01_ReadPos(Axis := ADR(gAxis01), Enable := TRUE);
gAxis01_Home(Axis := ADR(gAxis01), Execute := gAxis01_HomeEx);
gAxis01_MoveAbs1(Axis := ADR(gAxis01), Execute := gAxis01_Move1Ex, Position := GUI_csp_pos1,
    Velocity := GUI_csp_vel, Acceleration := GUI_csp_acc, Deceleration := GUI_csp_acc);
gAxis01_MoveAbs2(Axis := ADR(gAxis01), Execute := gAxis01_Move2Ex, Position := GUI_csp_pos2,
    Velocity := GUI_csp_vel, Acceleration := GUI_csp_acc, Deceleration := GUI_csp_acc);

(* Axis switch on *)
IF (GUI_en_csp) THEN
    CASE (gAxis01_State) OF
        1: (* Init reset *)
            gAxis01_ResetEx := FALSE;
            gAxis01_PowerEn := FALSE;
            gAxis01_HomeEx := FALSE;
            gAxis01_Move1Ex := FALSE;
            gAxis01_Move2Ex := FALSE;
            gAxis01_State := 2;
        2: (* Reset *)
            gAxis01_ResetEx := TRUE;
            IF (gAxis01_Reset.Done) THEN
                gAxis01_State := 3;
            END_IF
        3: (* Power *)
            gAxis01_ResetEx := FALSE;
            gAxis01_PowerEn := TRUE;
            IF (gAxis01_Power.Status) THEN
                gAxis01_State := 11;
            END_IF
        11: (* Home *)
            gAxis01_HomeEx := TRUE;
            IF (gAxis01_Home.Done) THEN
                gAxis01_State := 21;
            END_IF
        21: (* Move POS1 *)
            gAxis01_Move1Ex := TRUE;
            gAxis01_Move2Ex := FALSE;
            IF (gAxis01_MoveAbs1.Done) THEN
                gAxis01_State := 22;
            END_IF
        22: (* Move POS2 *)
            gAxis01_Move1Ex := FALSE;
            gAxis01_Move2Ex := TRUE;
            IF (gAxis01_MoveAbs2.Done) THEN
                gAxis01_State := 21;
            END_IF
    END_CASE
ELSE
    gAxis01_State := 1;
    gAxis01_ResetEx := FALSE;
    gAxis01_PowerEn := FALSE;
END_IF
```

Finally, the provided example adds a GUI suitable for a 4PPC70 that allows switching the drive on and off and setting up the targets and the parameters for the position trajectories.