TECHNOSOFT

MOTION TECHNOLOGY

How to restore the operation after a fault

Application Note

Easy Motion Studio II

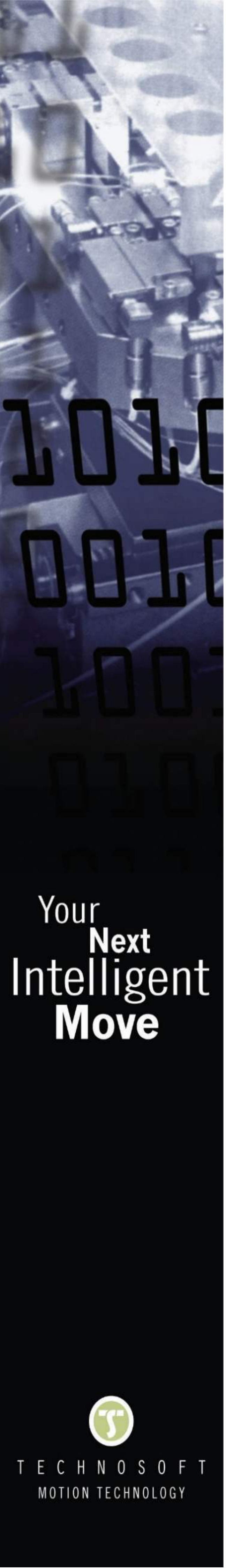


Table of contents

1. Application description

This application presents an example on how to restore the application operation after the drive enters a fault state. The idea behind the default operation of the drive is that in case of an error the user intervention is required (to determine the reason for the error and to take the corrective actions needed to prevent it from happening again).

Default behavior in case of an error:

- Switch off green LED
- Switch on red LED
- Disable the power stage by cutting the PWM outputs and suspend the controllers / trajectory generation
- Aborts any ongoing special calls / functions / homing and terminate the execution of the TML program.

Proposed recovery sequence:

- Switch off the green LED
- Switch on the red LED
- Cutoff drive PWM outputs
- Wait until the error condition disappears
- Program the motor to hold its current position
- Re-activate the drive PWM outputs
- Switch off the red LED and switch on the green LED
- Resume the main program and process the commands received from the master.

The application uses the software protection interrupt, which monitors the following software protections:

- Over current;
- Over temperature – drive (if available);
- Over temperature – motor (if available);
- I2t drive;
- I2t motor;
- Over voltage;
- Under voltage;
- Encoder broken wire (when incremental encoder is used and digital hall sensors are also present);

The easiest way to evaluate this application is to create an over-voltage condition thus having the drive enter the fault state. This can be done by decreasing the over-voltage protection threshold value under the supply voltage value. The parameter that stores the over voltage protection threshold value is called "UMAXPROT". We can then notice the drive behaviour and by returning UMAXPROT to its initial value, we can see the actual error recovery taking place.

2. Application flow chart

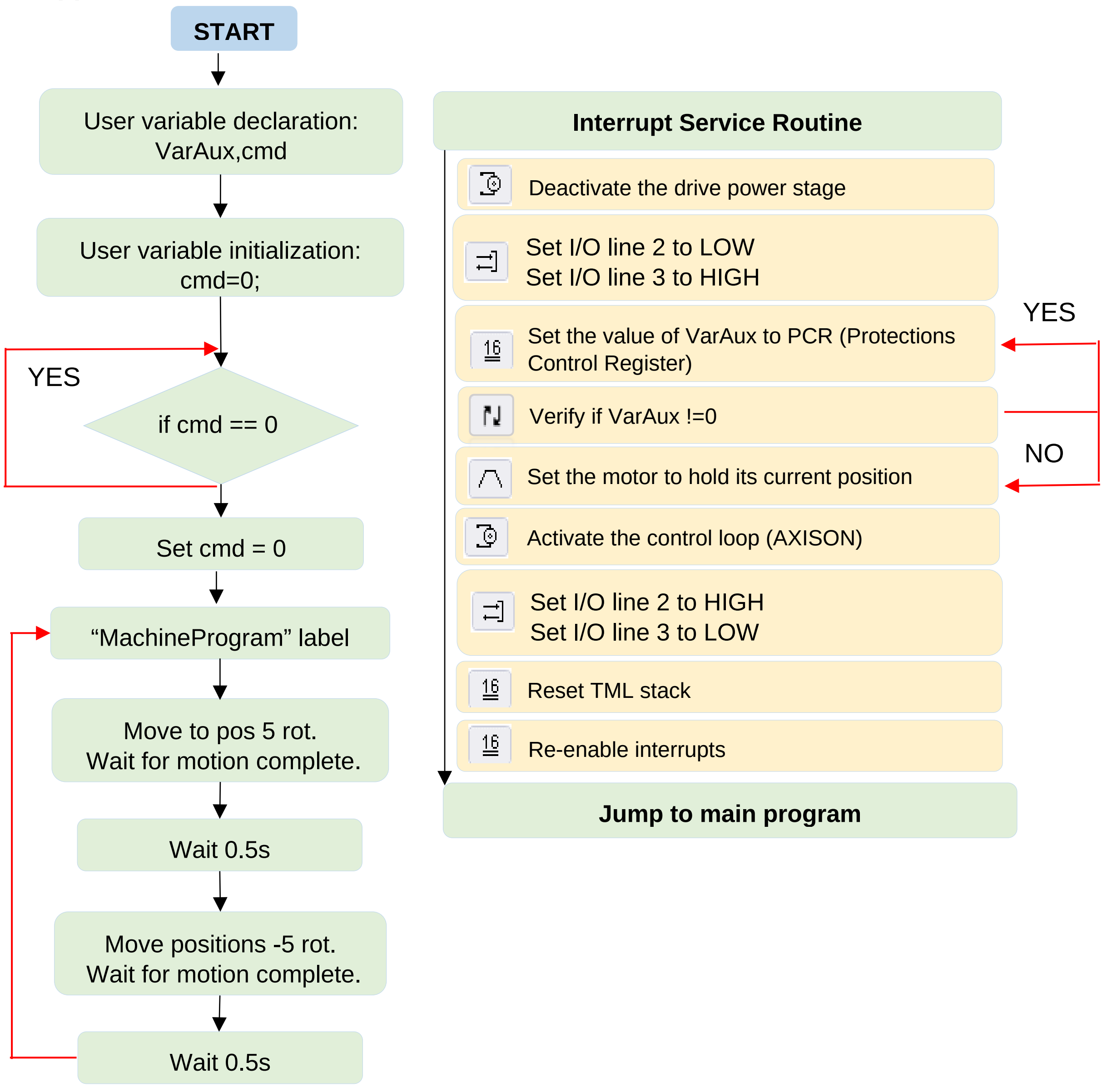

START
User variable declaration: VarAux,cmd
User variable initialization: cmd=0;
YES
if cmd == 0
Set cmd = 0
"MachineProgram" label
Move to pos 5 rot. Wait for motion complete.
Wait 0.5s
Move positions -5 rot. Wait for motion complete.
Wait 0.5s
Interrupt Service Routine
Deactivate the drive power stage
Set I/O line 2 to LOW
Set I/O line 3 to HIGH
Set the value of VarAux to PCR (Protections Control Register)
Verify if VarAux !=0
Set the motor to hold its current position
Activate the control loop (AXISON)
Set I/O line 2 to HIGH
Set I/O line 3 to LOW
Reset TML stack
Re-enable interrupts
YES
NO
Jump to main program

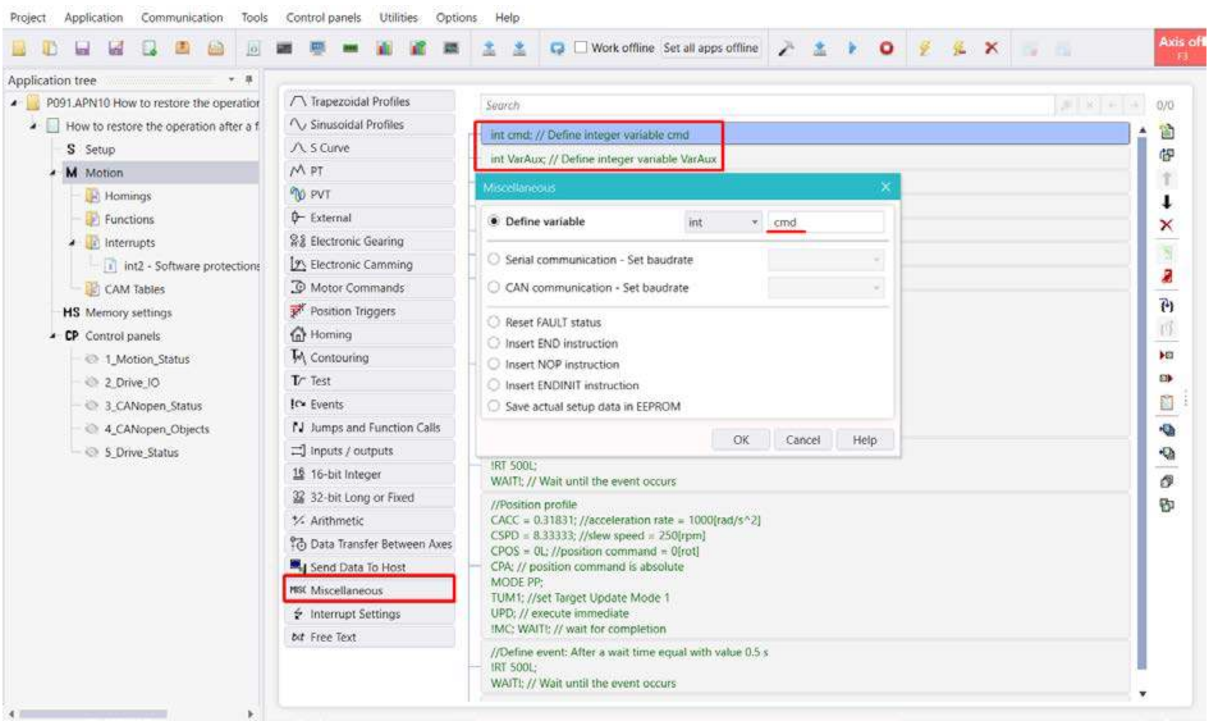
Figure 1 – Application structure

3. EasyMotion Studio II implementation

The application note outlines the TML steps need to automatically restore drive operation after a software protection / non-critical fault is triggered, creating the conditions for avoiding the user intervention.

3.1 Motion section

The “Miscellaneous” dialogue allows defining user variables (integer - 16bit, fixed or long - 32bit).



```
int cmd; // Define integer variable cmd
int VarAux; // Define integer variable VarAux
```

Miscellaneous:

- [✓ Define variable] int cmd
- Serial communication - Set baudrate
- CAN communication - Set baudrate
- Reset FAULT status
- Insert END instruction
- Insert NOP instruction
- Insert ENDINIT instruction
- Save actual setup data in EEPROM

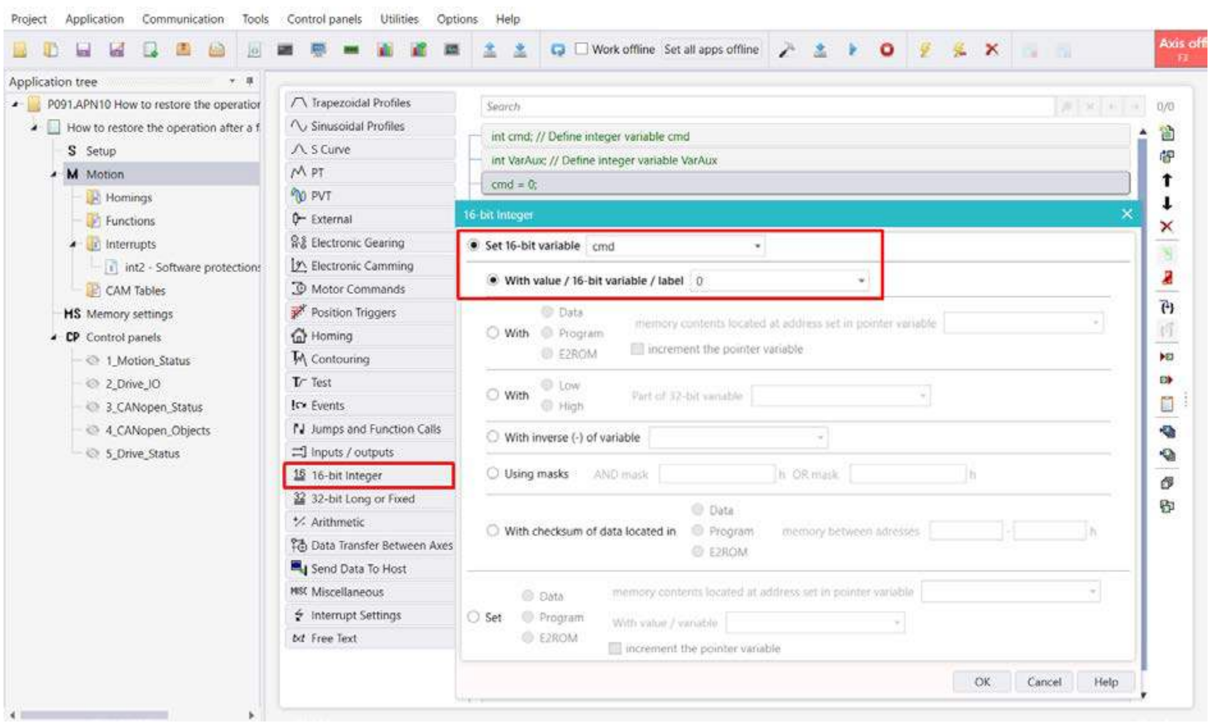
```
!RT 500L;
WAIT!; // Wait until the event occurs

//Position profile
CACC = 0.31831; //acceleration rate = 1000[rad/s^2]
CSPD = 8.33333; //slew speed = 250[rpm]
CPOS = 0L; //position command = 0[rot]
CPA; // position command is absolute
MODE PP;
TUM1; //set Target Update Mode 1
UPD; // execute immediate
!MC; WAIT!; // wait for completion

//Define event: After a wait time equal with value 0.5 s
!RT 500L;
WAIT!; // Wait until the event occurs
```

Figure 2 – Defining user variables

The “16-bit Integer” dialogue helps to assign a value to a 16-bit integer variable. This way a variable can be initialized or its value can be modified.



```
int cmd; // Define integer variable cmd
int VarAux; // Define integer variable VarAux
cmd = 0;
```

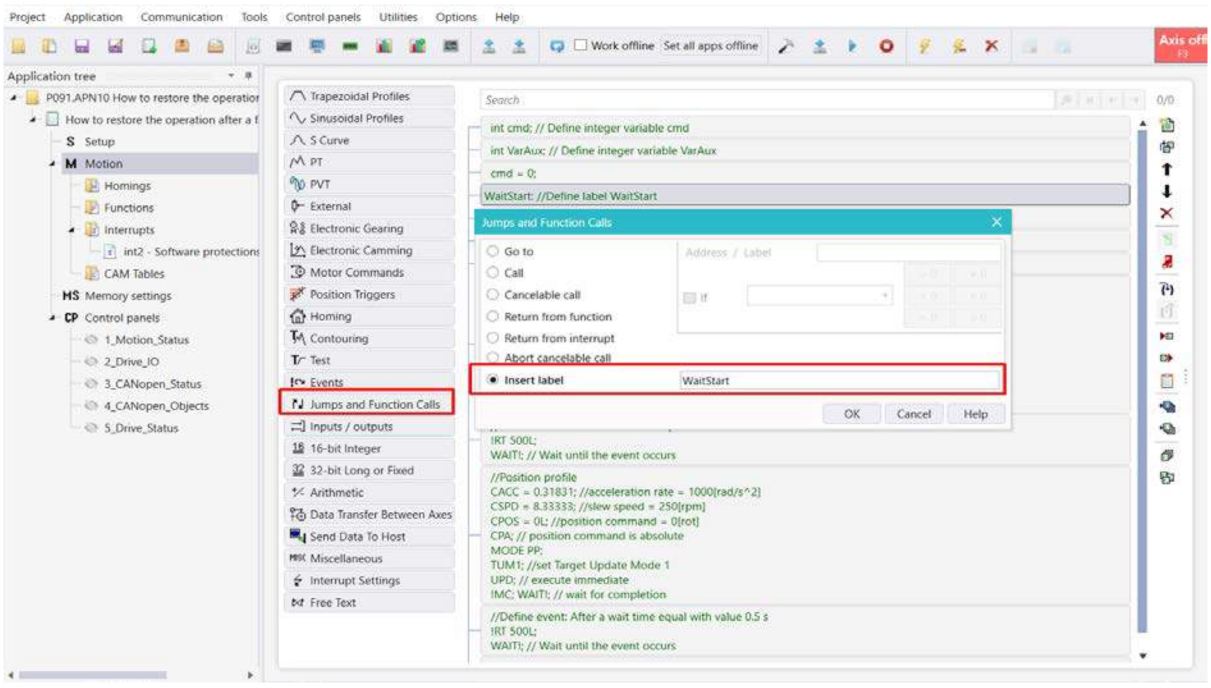
16-bit Integer

- (●) Set 16-bit variable: cmd
- (●) With value / 16-bit variable / label: 0

Figure 3 – Initializing the variables

The “Jumps and Function Calls” dialogue allows controlling the TML program flow through unconditional or conditional jumps and unconditional, conditional or cancelable calls of TML functions.

In this case, this wizard was used to create the “WaitStart” label where the program will be instructed to jump depending on the command set by the host controlling the drive to start the motion sequence, by setting the “cmd” user variable to a value different than 0. The labels, just like the names of the functions or variables are case insensitive. We prefer writing the label as “WaitStart” to make it more visible to the user, but “waitstart” or “WAITSTART” (or anything in between) as equally accepted.



```
int cmd; // Define integer variable cmd
int VarAux; // Define integer variable VarAux
cmd = 0;
WaitStart: //Define label WaitStart
```

Jumps and Function Calls

- (●) Insert label: WaitStart

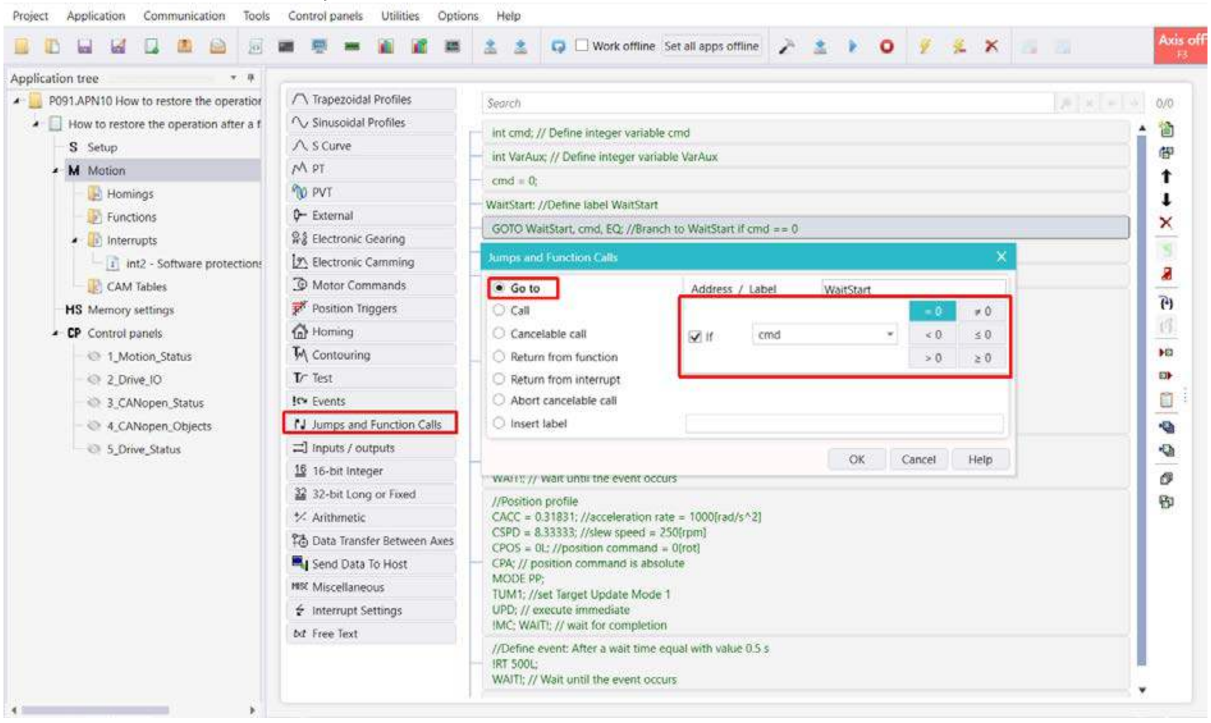
```
!RT 500L;
WAIT!; // Wait until the event occurs

//Position profile
CACC = 0.31831; //acceleration rate = 1000[rad/s^2]
CSPD = 8.33333; //slew speed = 250[rpm]
CPOS = 0L; //position command = 0[rot]
CPA; // position command is absolute
MODE PP;
TUM1; //set Target Update Mode 1
UPD; // execute immediate
!MC; WAIT!; // wait for completion

//Define event: After a wait time equal with value 0.5 s
!RT 500L;
WAIT!; // Wait until the event occurs
```

Figure 4 – Defining WaitStart label

This application notes depicts the situation of a host controlling the system from a higher level (i.e. giving a generic start command for example), hence the drive must wait until instructed to proceed. This waiting is implemented as a conditional jump that will keep the program in the "WaitStart" loop until the "cmd" variable is updated from outside.

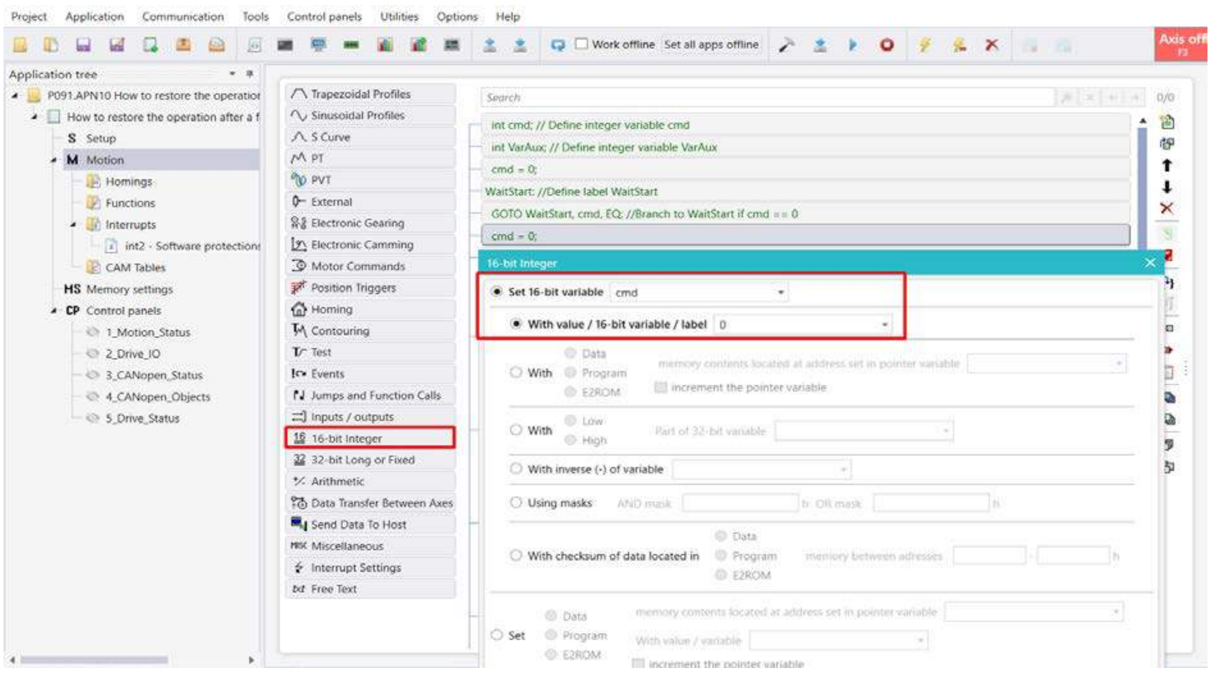


```
int cmd; // Define integer variable cmd
int VarAux; // Define integer variable VarAux
cmd = 0;
WaitStart: //Define label WaitStart
GOTO WaitStart, cmd, EQ; //Branch to WaitStart if cmd == 0
WAIT!; // Wait until the event occurs
//Position profile
CACC = 0.31831; //acceleration rate = 1000[rad/s^2]
CSPD = 8.33333; //slew speed = 250[rpm]
CPOS = 0L; //position command = 0[rot]
CPA; // position command is absolute
MODE PP;
TUM1; //set Target Update Mode 1
UPD; // execute immediate
!MC; WAIT!; // wait for completion
//Define event: After a wait time equal with value 0.5 s
!RT 500L;
WAIT!; // Wait until the event occurs
```

Jumps and Function Calls: [✓ Go to] Call / Cancelable call / Return from function / Return from interrupt / Abort cancelable call / Insert label; Address / Label: WaitStart; [✓ If] cmd [✓ = 0] ≠ 0 < 0 ≤ 0 > 0 ≥ 0

Figure 5 – Conditional jump to "WaitStart" label

The variable "cmd" is initialized to 0 using the "16-bit Integer" dialog to ensure it starts from a known state. Since an user variable is just an assignation of a special RAM memory zone to the name used, the initial value present in the RAM area is not controlled, hence we need to assign a known value to it. This practice helps prevent undefined behavior and ensures the program logic operates as intended from a predictable baseline.

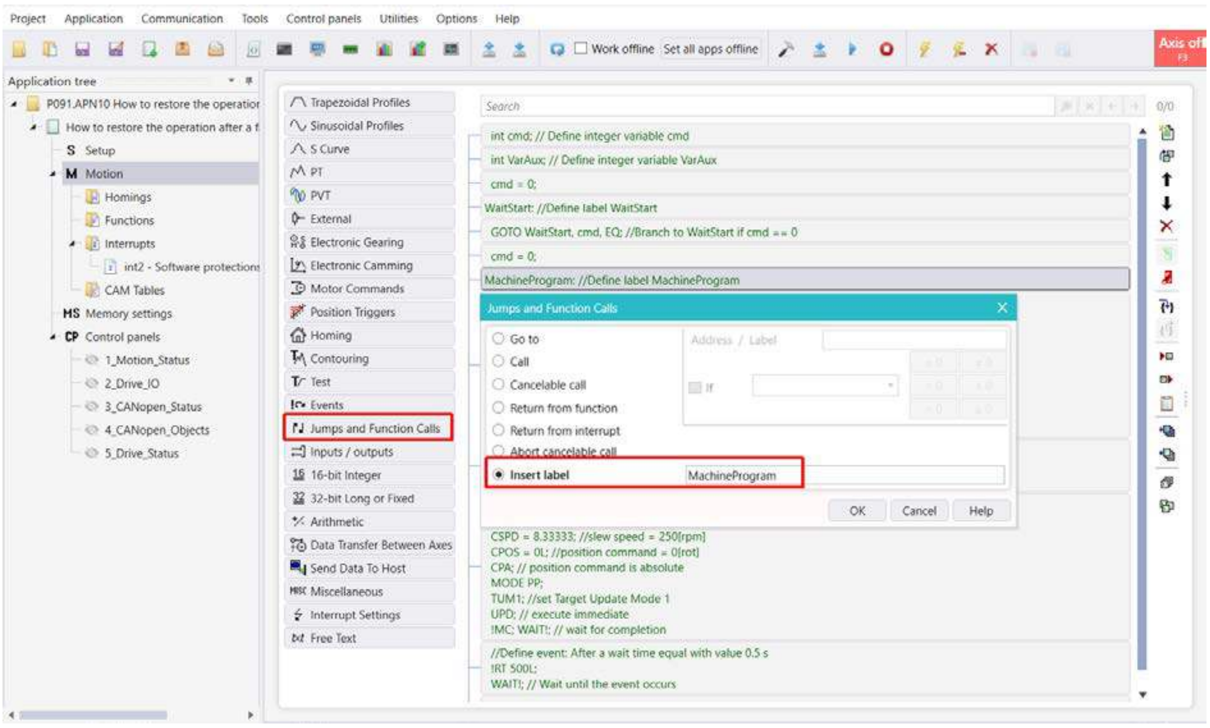


```
int cmd; // Define integer variable cmd
int VarAux; // Define integer variable VarAux
cmd = 0;
WaitStart: //Define label WaitStart
GOTO WaitStart, cmd, EQ; //Branch to WaitStart if cmd == 0
cmd = 0;
```

16-bit Integer: [✓ Set 16-bit variable] cmd; [✓ With value / 16-bit variable / label] 0

Figure 6 – Initializing user variable "cmd"

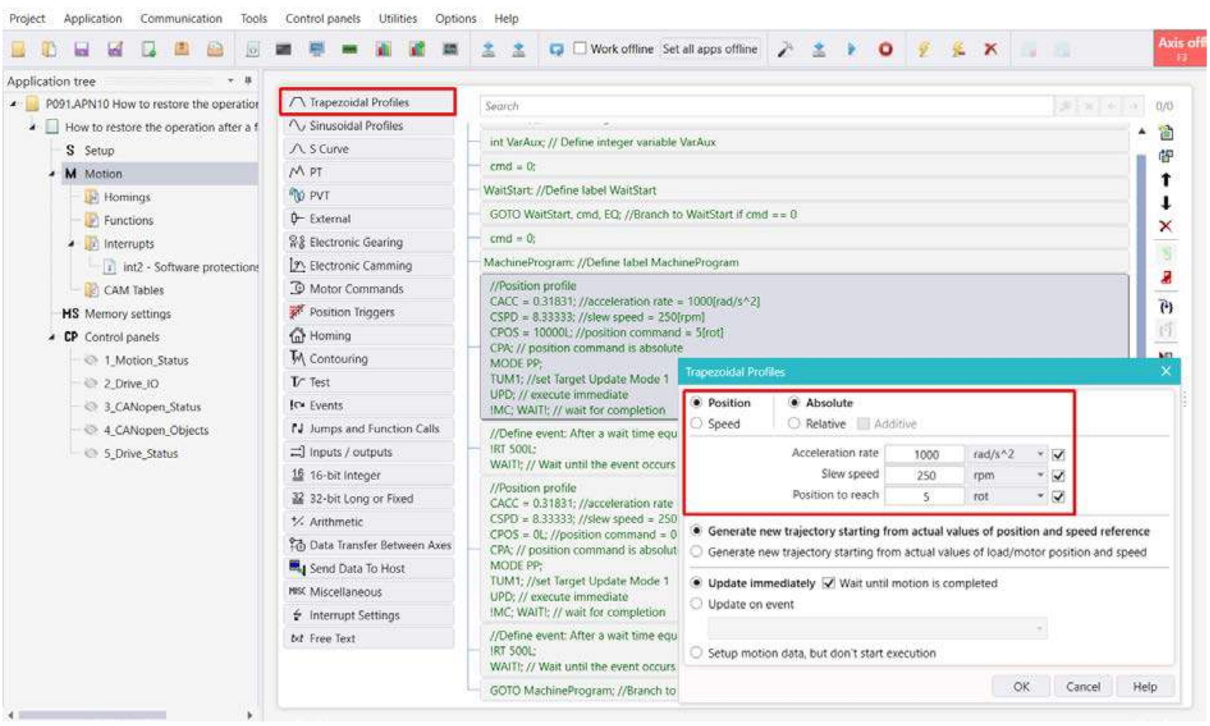
The label “MachineProgram” is defined in the code to designate a specific section where machine operations are executed. This label acts as a reference point within the program, allowing the control flow to jump to this section when required and enabling the possibility to repeat the same action as many times as needed.


```
int cmd; // Define integer variable cmd
int VarAux; // Define integer variable VarAux
cmd = 0;
WaitStart: //Define label WaitStart
GOTO WaitStart, cmd, EQ; //Branch to WaitStart if cmd == 0
cmd = 0;
MachineProgram: //Define label MachineProgram
CSPD = 8.33333; //slew speed = 250[rpm]
CPOS = 0L; //position command = 0[rot]
CPA; // position command is absolute
MODE PP;
TUM1; //set Target Update Mode 1
UPD; // execute immediate
!MC; WAIT!; // wait for completion
//Define event: After a wait time equal with value 0.5 s
!RT 500L;
WAIT!; // Wait until the event occurs
```

Jumps and Function Calls: Go to / Call / Cancelable call / Return from function / Return from interrupt / Abort cancelable call / [✓ Insert label] MachineProgram

Figure 7 – Defining “MachineProgram” label

The “Motion – Trapezoidal Profiles” dialogue allows you to program a position or speed profile with a trapezoidal shape of the speed, due to a limited acceleration.


```
int VarAux; // Define integer variable VarAux
cmd = 0;
WaitStart: //Define label WaitStart
GOTO WaitStart, cmd, EQ; //Branch to WaitStart if cmd == 0
cmd = 0;
MachineProgram: //Define label MachineProgram
//Position profile
CACC = 0.31831; //acceleration rate = 1000[rad/s^2]
CSPD = 8.33333; //slew speed = 250[rpm]
CPOS = 10000L; //position command = 5[rot]
CPA; // position command is absolute
MODE PP;
TUM1; //set Target Update Mode 1
UPD; // execute immediate
!MC; WAIT!; // wait for completion
//Define event: After a wait time equ
!RT 500L;
WAIT!; // Wait until the event occurs
//Position profile
CACC = 0.31831; //acceleration rate
CSPD = 8.33333; //slew speed = 250
CPOS = 0L; //position command = 0
CPA; // position command is absolut
MODE PP;
TUM1; //set Target Update Mode 1
UPD; // execute immediate
!MC; WAIT!; // wait for completion
//Define event: After a wait time equ
!RT 500L;
WAIT!; // Wait until the event occurs
GOTO MachineProgram; //Branch to
```

Trapezoidal Profiles: [✓ Position] Speed; [✓ Absolute] Relative, Additive

Acceleration rate	1000	rad/s^2	✓
Slew speed	250	rpm	✓
Position to reach	5	rot	✓

[✓ Generate new trajectory starting from actual values of position and speed reference] / Generate new trajectory starting from actual values of load/motor position and speed

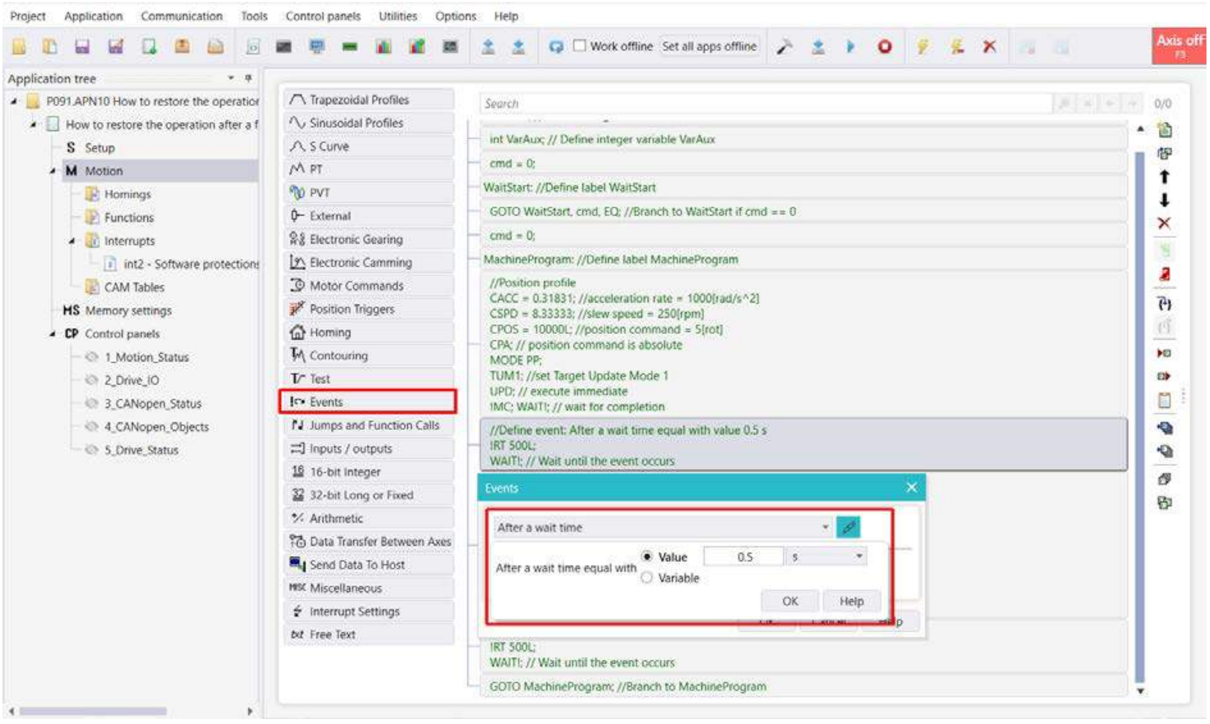
[✓ Update immediately] ✓ Wait until motion is completed / Update on event / Setup motion data, but don't start execution

Figure 8 – Inserting a trapezoidal position profile

The “Events” dialogue allows you to define events. An event is a certain condition that the drive can be programmed to monitor for its occurrence. Although there are more types of events, the drive can have only one event ready at any given moment. If you program 2 or more events (without specifically waiting for each of them) only the last one will be used when the drive will be instructed to wait for the event to happen.

The following actions can be connected to an event:

- stop the motion when the event occurs.
- wait for the programmed event to occur.
- exit the waiting for event loop after a given time.



```
int VarAux; // Define integer variable VarAux
cmd = 0;
WaitStart: //Define label WaitStart
GOTO WaitStart, cmd, EQ; //Branch to WaitStart if cmd == 0
cmd = 0;
MachineProgram: //Define label MachineProgram
//Position profile
CACC = 0.31831; //acceleration rate = 1000[rad/s^2]
CSPD = 8.33333; //slew speed = 250[rpm]
CPOS = 10000L; //position command = 5[rot]
CPA; // position command is absolute
MODE PP;
TUM1; //set Target Update Mode 1
UPD; // execute immediate
!MC; WAIT!; // wait for completion
//Define event: After a wait time equal with value 0.5 s
!RT 500L;
WAIT!; // Wait until the event occurs
!RT 500L;
WAIT!; // Wait until the event occurs
GOTO MachineProgram; //Branch to MachineProgram
```

Events

After a wait time

After a wait time equal with [✓ Value] 0.5 s / Variable

OK Help

Figure 9 – Defining a wait time event

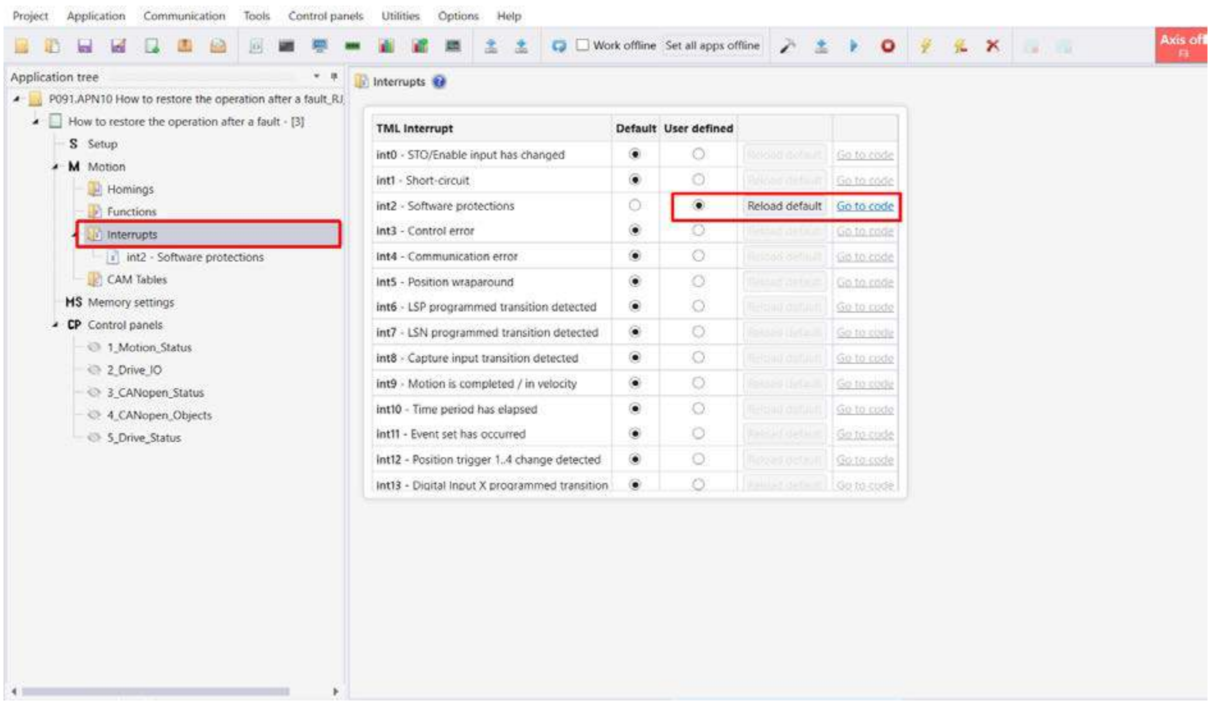
After the 0.5 seconds wait, another positioning with a waiting time is added to create a back and forth movement to have a clear distinction between the moments when the application is running and when it is in error state.

The final GOTO MachineProgram instruction is closing the loop, thus the motor is continuously moving 5 rotations with a pause of 0.5sec.

3.2 Software Protections Interrupt – INT2

The TML interrupts are special conditions that are continuously monitored by the drive firmware. When a TML interrupt occurs, the main TML program execution is suspended and the TML code associated with the interrupt, called Interrupt Service Routine (in short ISR), is executed.

Remark: While any TML interrupt is active, the other interrupts are deactivated. It is recommended to keep the ISR as short as possible. If this is not the case, then the user should consider if other interrupts should be re-enabled using the “Interrupts Settings” dialogue.

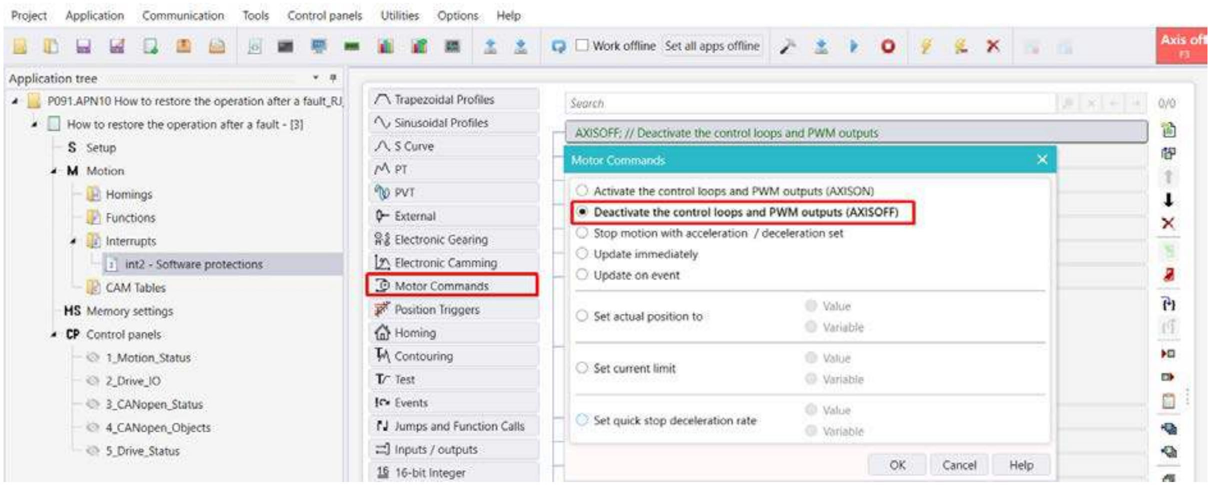


TML Interrupt	Default	User defined		
int0 - STO/Enable input has changed	●	○	Reload default	Go to code
int1 - Short-circuit	●	○	Reload default	Go to code
int2 - Software protections	○	●	Reload default	Go to code
int3 - Control error	●	○	Reload default	Go to code
int4 - Communication error	●	○	Reload default	Go to code
int5 - Position wraparound	●	○	Reload default	Go to code
int6 - LSP programmed transition detected	●	○	Reload default	Go to code
int7 - LSN programmed transition detected	●	○	Reload default	Go to code
int8 - Capture input transition detected	●	○	Reload default	Go to code
int9 - Motion is completed / in velocity	●	○	Reload default	Go to code
int10 - Time period has elapsed	●	○	Reload default	Go to code
int11 - Event set has occurred	●	○	Reload default	Go to code
int12 - Position trigger 1..4 change detected	●	○	Reload default	Go to code
int13 - Digital Input X programmed transition	●	○	Reload default	Go to code

Figure 10 – Customizing the TML interrupt service routines

This application was implemented using the “Int2 - Software protections” interrupt, that was customized to contain the code that corresponds to the recovery sequence proposed in the first chapter of this document.

The “Motor Commands” dialogue was used to deactivate the control loops and the power stage PWM output commands (AXISOFF) when the error has occurred.



```
AXISOFF; // Deactivate the control loops and PWM outputs
```

Motor Commands

- ○ Activate the control loops and PWM outputs (AXISON)
- ● Deactivate the control loops and PWM outputs (AXISOFF)
- ○ Stop motion with acceleration / deceleration set
- ○ Update immediately
- ○ Update on event
- ○ Set actual position to — Value / Variable
- ○ Set current limit — Value / Variable
- ○ Set quick stop deceleration rate — Value / Variable

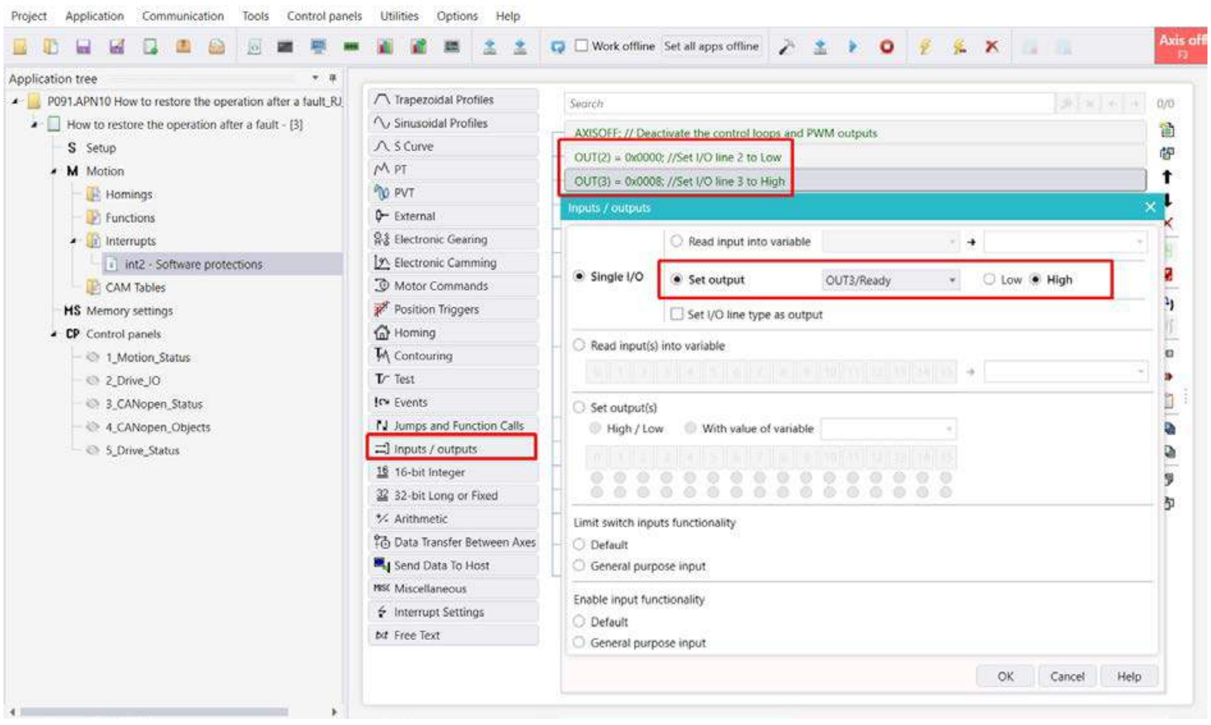
OK Cancel Help

Figure 11 – Disabling the power stage

The “I/O (Inputs/Outputs)” dialogue allows programming the following operations with the digital inputs and outputs:

- read and save the status of a digital input into a variable
- set low or high a digital output
- read and save the status of multiple digital inputs into a variable
- set multiple digital outputs according with an immediate value or the value of 16-bit variable

In this application, the “I/O” dialogue is used to set LOW / HIGH the “Ready” and “Error” digital output, that are also associated to the green and respective red LED on the drive. Upon entering the interrupt, the red LED will be switched on and the green one off, to signal the acknowledge of the error condition.


```
AXISOFF; // Deactivate the control loops and PWM outputs
OUT(2) = 0x0000; //Set I/O line 2 to Low
OUT(3) = 0x0008; //Set I/O line 3 to High
```

Inputs / outputs: Single I/O [✓] — Set output [✓] OUT3/Ready — Low [✓ High]

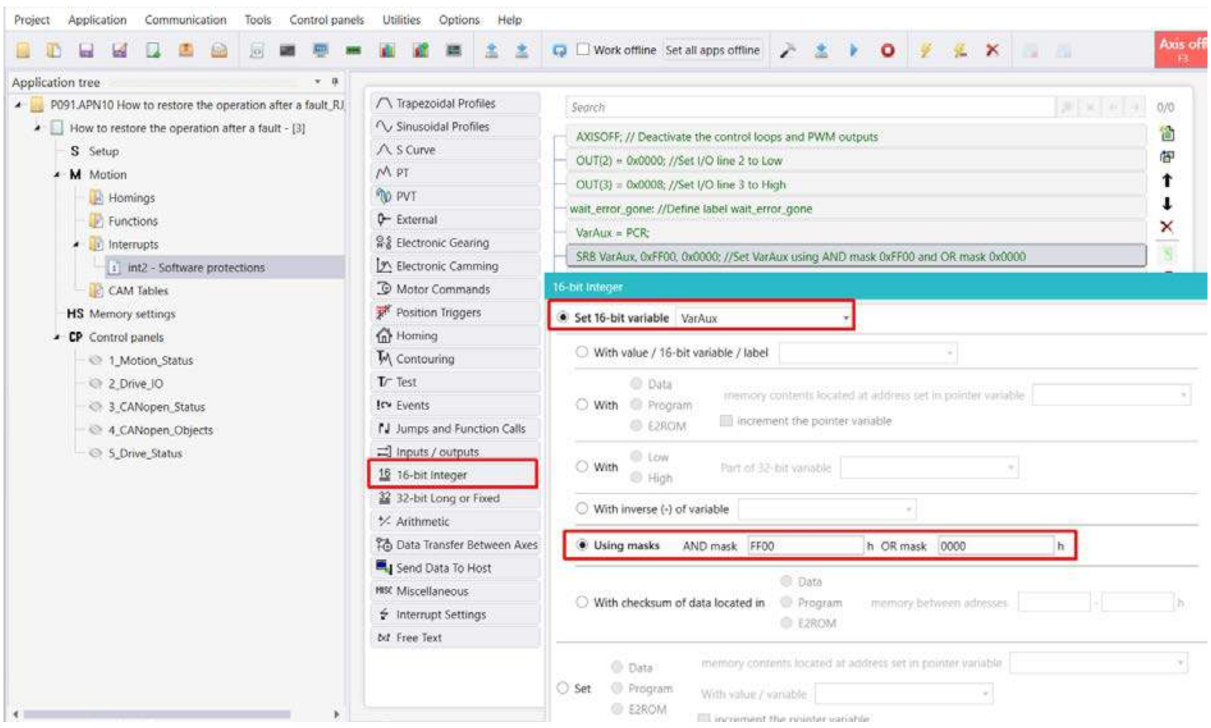
Figure 12 – Setting output lines status

The next part of the routine is a small conditional loop which is used to program the drive to wait until the error condition is gone. For this purpose, the PCR (is used.

The “16-bit Integer” dialogue is used to copy the value of PCR (Protection Control Register) register in the 16-bit integer VarAux.

Inside the “wait_error_gone” loop we apply an “AND” and “OR” mask to the “VarAux” user variable. The mask isolates bits 15 to 8 in the “VarAux” variable, to check if the error state has disappeared.

Remark: PCR (Protections Control Register) is a 16-bit command and status register, containing the status information of the TML protections. A detailed description of the PCR register is presented in the EasyMotion Studio II help topics.



```
AXISOFF; // Deactivate the control loops and PWM outputs
OUT(2) = 0x0000; //Set I/O line 2 to Low
OUT(3) = 0x0008; //Set I/O line 3 to High
wait_error_gone: //Define label wait_error_gone
VarAux = PCR;
SRB VarAux, 0xFF00, 0x0000; //Set VarAux using AND mask 0xFF00 and OR mask 0x0000
```

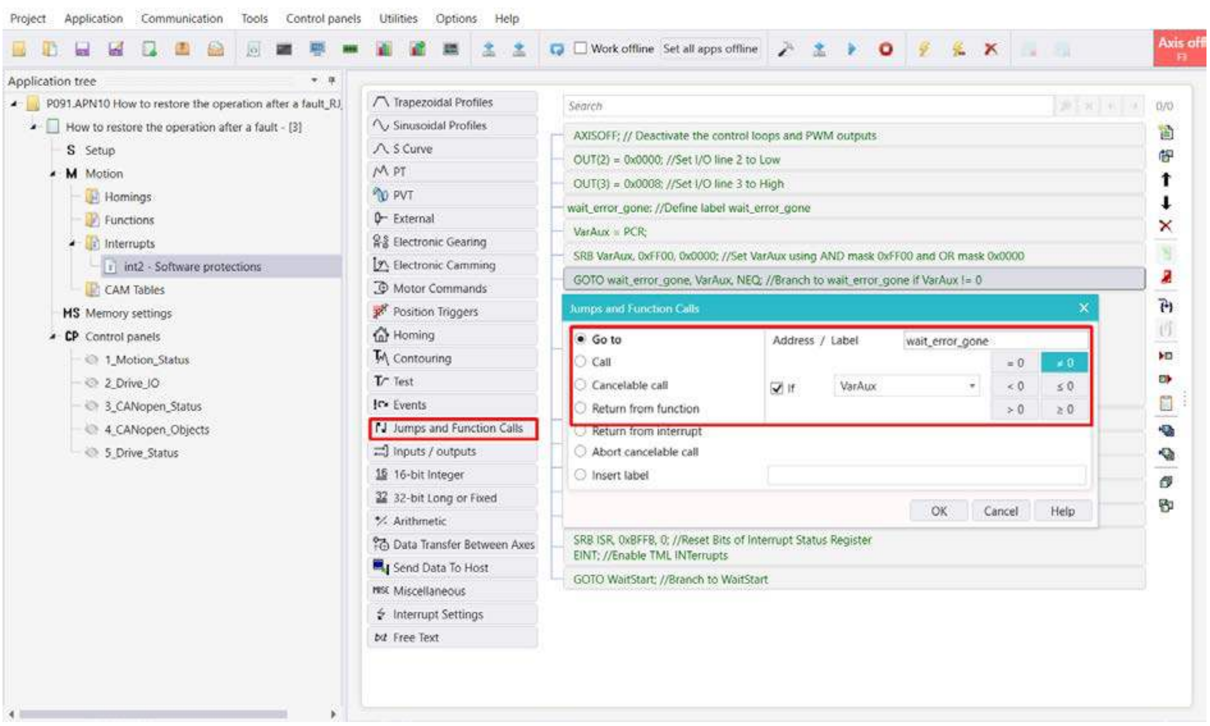
16-bit Integer

Set 16-bit variable: VarAux

Using masks: AND mask FF00 h OR mask 0000 h

Figure 13 – Applying an AND and OR mask to a 16-bits variable

The jump to “wait_error_gone” is performed to create a loop that continuously checks the state of the “VarAux” variable. This is done to ensure that the program waits until the error condition is gone before proceeding further.



```
AXISOFF; // Deactivate the control loops and PWM outputs
OUT(2) = 0x0000; //Set I/O line 2 to Low
OUT(3) = 0x0008; //Set I/O line 3 to High
wait_error_gone: //Define label wait_error_gone
VarAux = PCR;
SRB VarAux, 0xFF00, 0x0000; //Set VarAux using AND mask 0xFF00 and OR mask 0x0000
GOTO wait_error_gone, VarAux, NEQ; //Branch to wait_error_gone if VarAux != 0
SRB ISR, 0xBFF8, 0; //Reset Bits of Interrupt Status Register
EINT; //Enable TML INTerrupts
GOTO WaitStart; //Branch to WaitStart
```

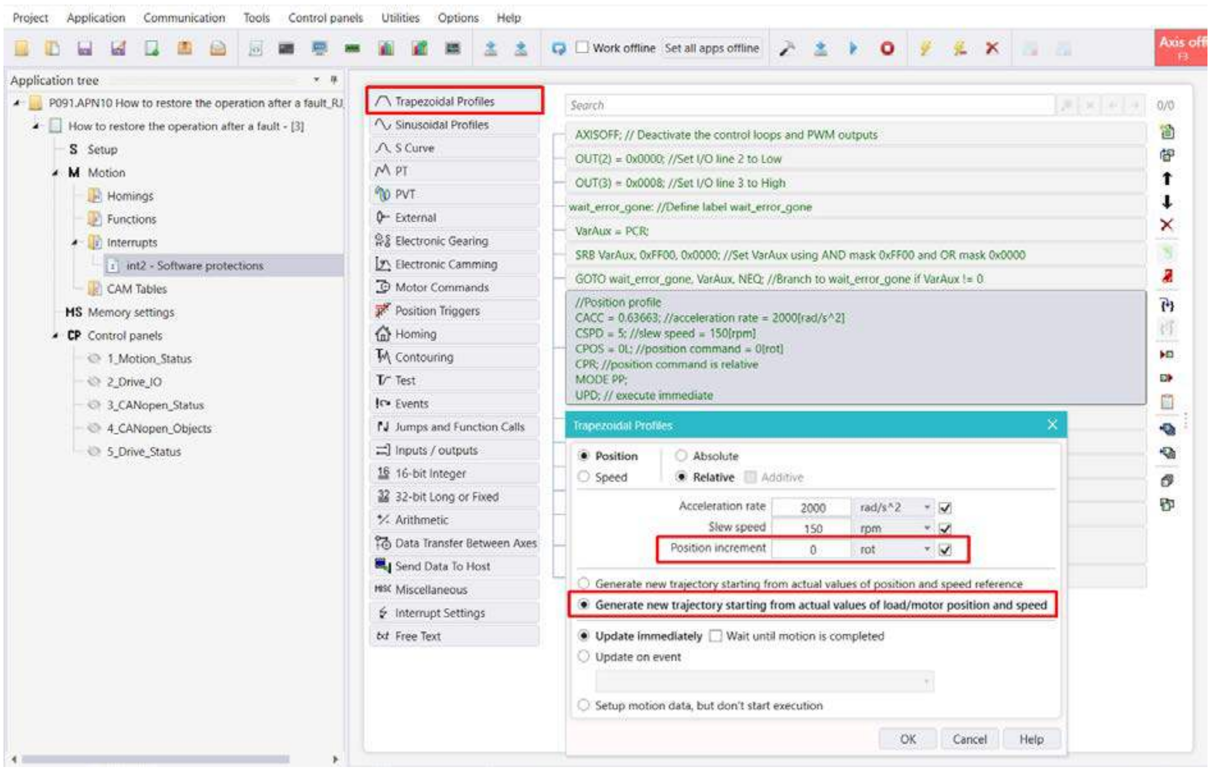
Jumps and Function Calls

Go to — Address / Label: wait_error_gone

If VarAux ≠ 0

_Figure 14 – Jump to “wait_error_gone” label_

The first part of the actual recovery from error is to use the “Trapezoidal Profiles” dialogue to program the motor to hold the current position, so that the axis will be re-enabled the motor will not move.



```
AXISOFF; // Deactivate the control loops and PWM outputs
OUT(2) = 0x0000; //Set I/O line 2 to Low
OUT(3) = 0x0008; //Set I/O line 3 to High
wait_error_gone: //Define label wait_error_gone
VarAux = PCR;
SRB VarAux, 0xFF00, 0x0000; //Set VarAux using AND mask 0xFF00 and OR mask 0x0000
GOTO wait_error_gone, VarAux, NEQ; //Branch to wait_error_gone if VarAux != 0
//Position profile
CACC = 0.63663; //acceleration rate = 2000[rad/s^2]
CSPD = 5; //slew speed = 150[rpm]
CPOS = 0L; //position command = 0[rot]
CPR; //position command is relative
MODE PP;
UPD; // execute immediate
```

Trapezoidal Profiles

[✓ Position] Speed — Absolute [✓ Relative] Additive

Acceleration rate 2000 rad/s^2
Slew speed 150 rpm
Position increment 0 rot

Generate new trajectory starting from actual values of position and speed reference
[✓ Generate new trajectory starting from actual values of load/motor position and speed]

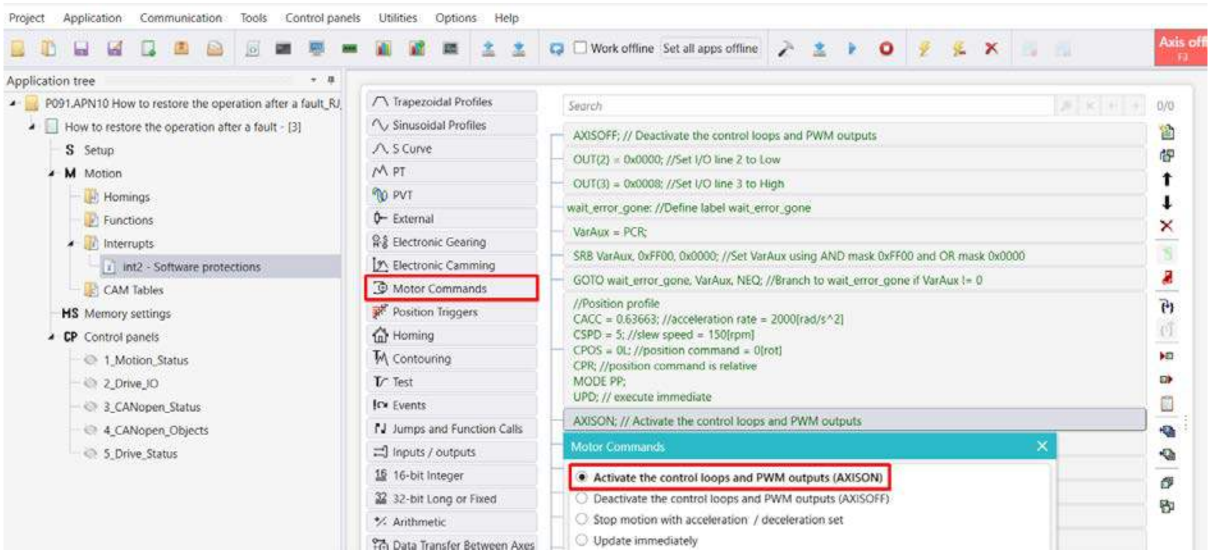
[✓ Update immediately] Wait until motion is completed
Update on event
Setup motion data, but don't start execution

OK Cancel Help

Figure 15 – Instructing the motor to hold position

Remark: When resuming motor control after an AXISOFF situation, during which time the motor might have been moved by external means, it is mandatory to use the “Generate new trajectory starting from actual values of load/motor position and speed” not the other option (default – start from reference). This is needed if the motor moves while the drive is disable – if the trajectory is resumed without this change, the PID will see the difference between the last known reference point and the current motor position as a position error and the motor will be forced rather aggressively back to the position where it was when the error appeared.

Next, the “Motor Commands” dialogue is used again to reactivate the drive PWM outputs and re-enable the motor.



```
AXISOFF; // Deactivate the control loops and PWM outputs
OUT(2) = 0x0000; //Set I/O line 2 to Low
OUT(3) = 0x0008; //Set I/O line 3 to High
wait_error_gone: //Define label wait_error_gone
VarAux = PCR;
SRB VarAux, 0xFF00, 0x0000; //Set VarAux using AND mask 0xFF00 and OR mask 0x0000
GOTO wait_error_gone, VarAux, NEQ; //Branch to wait_error_gone if VarAux != 0
//Position profile
CACC = 0.63663; //acceleration rate = 2000[rad/s^2]
CSPD = 5; //slew speed = 150[rpm]
CPOS = 0L; //position command = 0[rot]
CPR; //position command is relative
MODE PP;
UPD; // execute immediate
AXISON; // Activate the control loops and PWM outputs
```

Motor Commands

[✓ Activate the control loops and PWM outputs (AXISON)]
Deactivate the control loops and PWM outputs (AXISOFF)
Stop motion with acceleration / deceleration set
Update immediately

Figure 16 – Activating the drive power stage

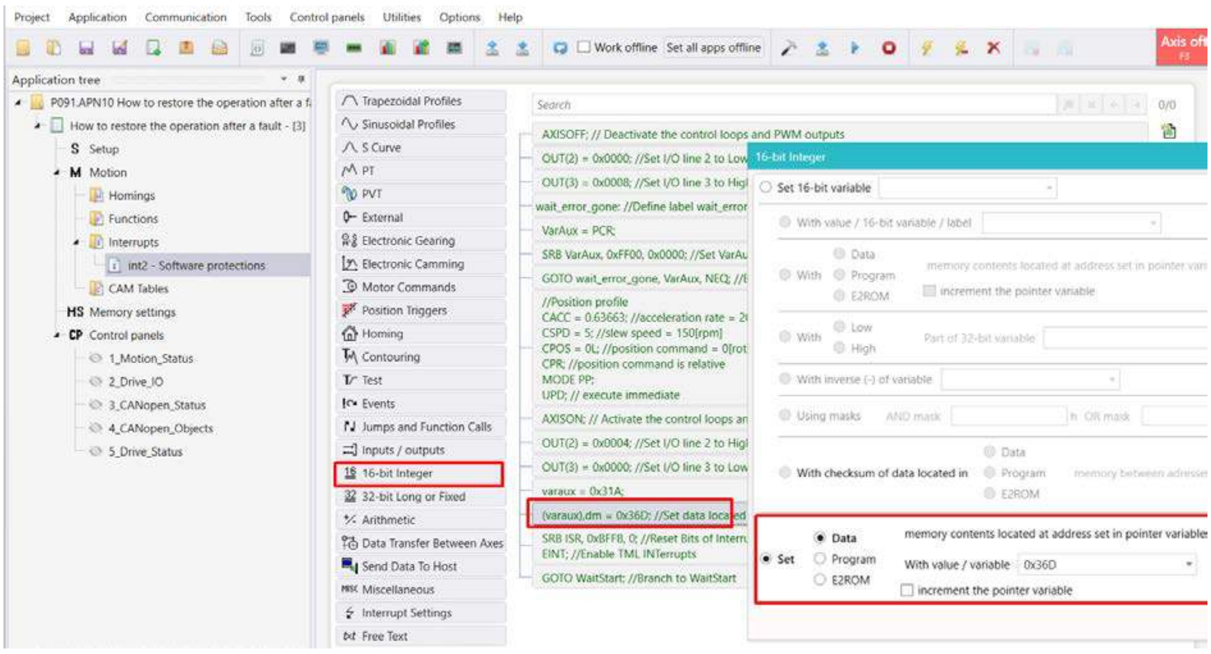
The green LED (Ready – OUT3) is turned on and the red LED (Error – OUT2) is switched OFF, using the “I/O (Inputs/Outputs)” dialogue.

Since in this particular case (error recovery) it is required that the TML program flow is resumed always from the same specific point (in this example, from the WaitStart label) we’ll have to use the unconditional jump (GOTO) instruction to do so.

However, unlike any previous GOTOs used in the program, there is a particularity of any interrupt, function or homing routine that must be considered first: whenever they are started, the firmware is saving the “return address” of the next TML instruction from the program that was currently being executed before passing control to the interrupt / function / homing. In this way, when they are finished the drive will continue the execution of the on-going TML program.

This save operation implies using a reserved RAM area of the drive called the “TML stack” which is limited (depending on the drive used it can have either 11 or 16 locations) and hence its handling is very important. This is why every interrupt / function / homing procedure must end with the special TML instruction RETI (RETurn from Interrupt) or RET (RETurn from function / homing).

So, since we are going to finish the ISR2 in an abnormal way (i.e. instead of the RETI, we’ll use a GOTO) we need to reset the TML stack manually. This is done by writing the value 0x36D at the address 0x31A and it is done using the “16-bit integer” wizard



```
AXISOFF; // Deactivate the control loops and PWM outputs
OUT(2) = 0x0000; //Set I/O line 2 to Low
OUT(3) = 0x0008; //Set I/O line 3 to Higl
wait_error_gone: //Define label wait_error
VarAux = PCR;
SRB VarAux, 0xFF00, 0x0000; //Set VarAu
GOTO wait_error_gone, VarAux, NEQ; //E
//Position profile
CACC = 0.63663; //acceleration rate = 2(
CSPD = 5; //slew speed = 150[rpm]
CPOS = 0L; //position command = 0[rot
CPR; //position command is relative
MODE PP;
UPD; // execute immediate
AXISON; // Activate the control loops an
OUT(2) = 0x0004; //Set I/O line 2 to Higl
OUT(3) = 0x0000; //Set I/O line 3 to Low
varaux = 0x31A;
(varaux).dm = 0x36D; //Set data located
SRB ISR, 0xBFFB, 0; //Reset Bits of Interru
EINT; //Enable TML INTerrupts
GOTO WaitStart; //Branch to WaitStart
```

16-bit Integer

( ) Set 16-bit variable

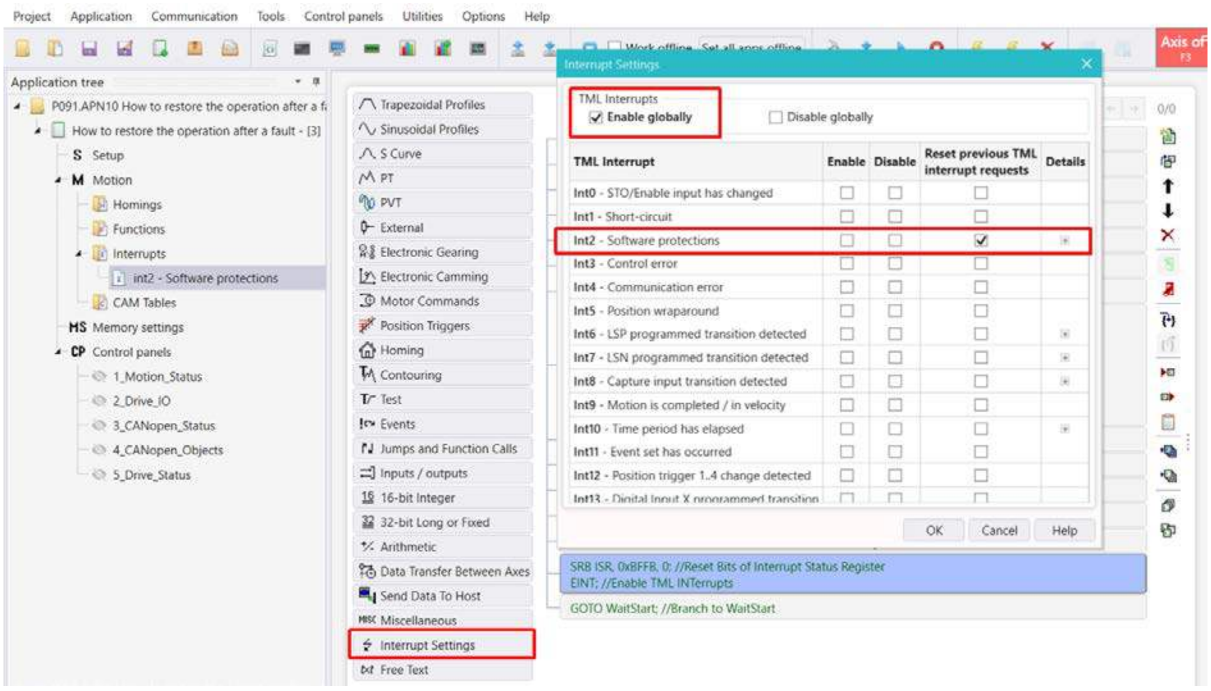
[✓ Set] [✓ Data] Program E2ROM memory contents located at address set in pointer variable

With value / variable: 0x36D

☐ increment the pointer variable

Figure 17 – Indirect write operation to RAM address 0x31A of the value 0x36D to reset the TML stack

The last thing to do before exiting the ISR2 is to globally re-enable the interrupt (operation which is normally done by RETI instruction) and also reset any possible occurrence of an additional INT2 in the mean time.



TML Interrupts
[✓] Enable globally
[ ] Disable globally

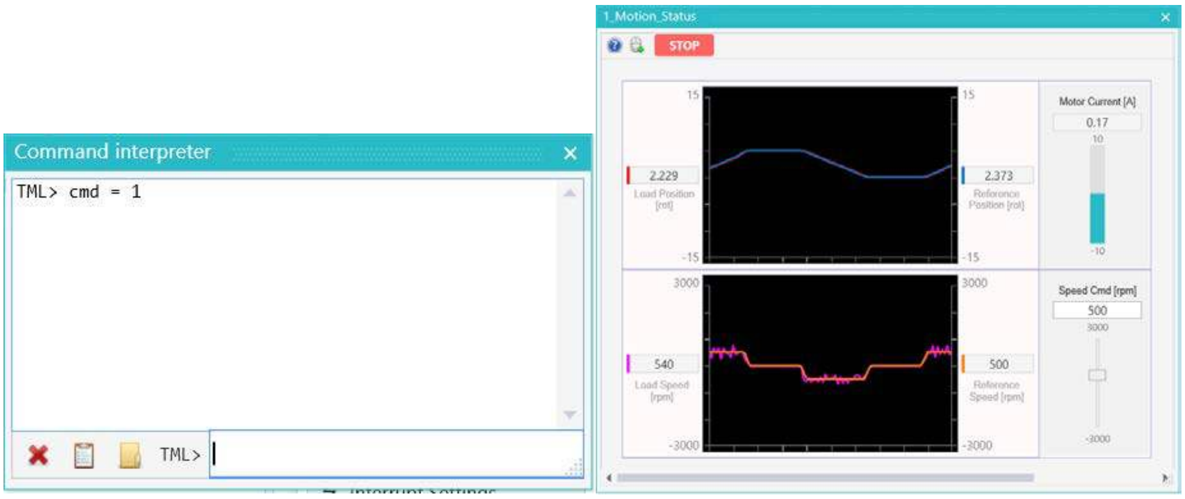
TML Interrupt	Enable	Disable	Reset previous TML interrupt requests	Details
Int0 - STO/Enable input has changed	☐	☐	☐	
Int1 - Short-circuit	☐	☐	☐	
Int2 - Software protections	☐	☐	✓	
Int3 - Control error	☐	☐	☐	
Int4 - Communication error	☐	☐	☐	
Int5 - Position wraparound	☐	☐	☐	
Int6 - LSP programmed transition detected	☐	☐	☐	
Int7 - LSN programmed transition detected	☐	☐	☐	
Int8 - Capture input transition detected	☐	☐	☐	
Int9 - Motion is completed / in velocity	☐	☐	☐	
Int10 - Time period has elapsed	☐	☐	☐	
Int11 - Event set has occurred	☐	☐	☐	
Int12 - Position trigger 1..4 change detected	☐	☐	☐	
Int13 - Digital Input X programmed transition	☐	☐	☐	

```
SRB ISR, 0xBFFB, 0; //Reset Bits of Interrupt Status Register
EINT; //Enable TML INTerrupts
GOTO WaitStart; //Branch to WaitStart
```

Figure 18 – Globally enable interrupts and reset INT2 previous requests

3.3 Application evaluation, using the over-voltage protection

After the application runs, the drive will wait for the master to send the start command. To simulate the master, the “Command Interpreter” window can be used to set the “cmd” variable to value 1. The motor will start to perform the motion profiles inside the “MachineProgram” loop. This can be monitored, using the “1_Motion Status” control panel.

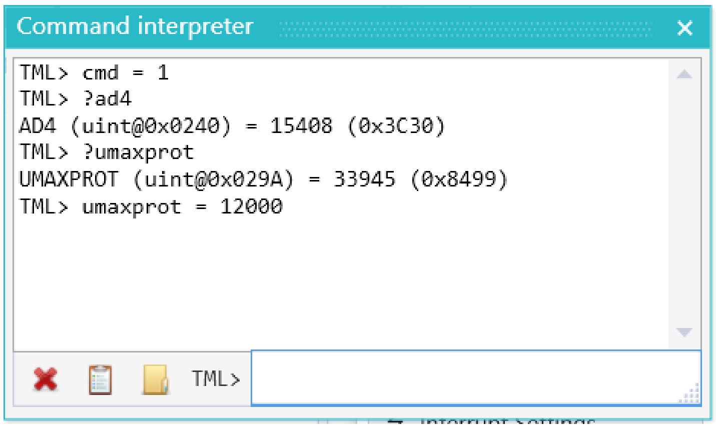


```
TML> cmd = 1
```

Figure 19 – Starting the motion when “cmd” becomes different than 0

The drive is continuously monitoring the motor supply voltage (read via the AD4 variable). When this value goes over the set threshold (UMAXPORT), the over-voltage protection triggers and the drive executes the TML code from the ISR2 (Interrupt Service Routine).

To simulate an over-voltage situation, “UMAXPROT” should be set lower than AD4. With the help of the “command interpreter” we can get the current value of AD4 and UMAXPROT and then we can set UMAXPROT to an arbitrary value.

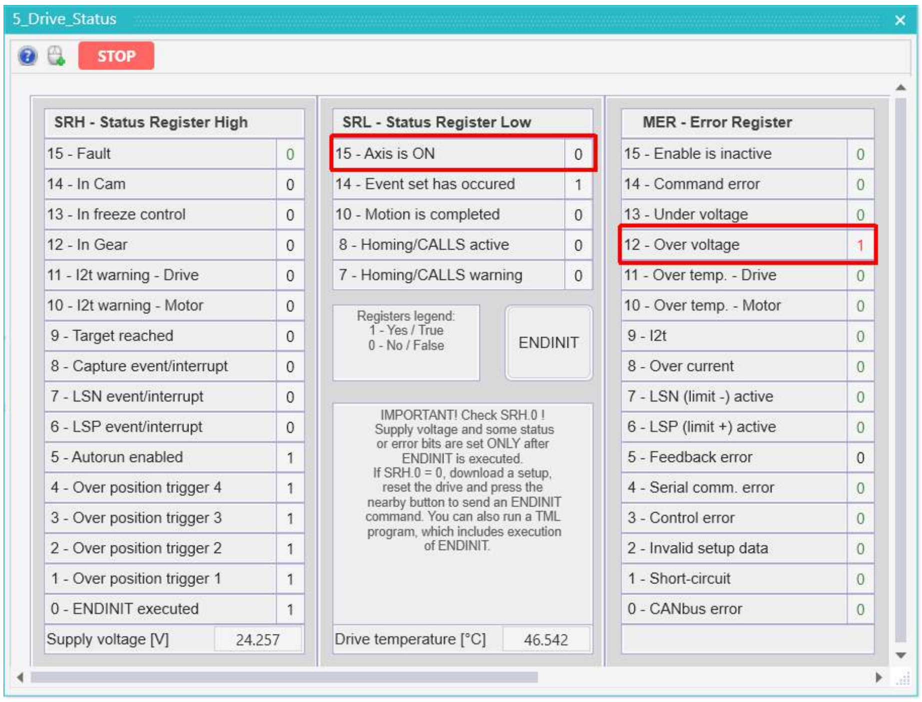


```
TML> cmd = 1
TML> ?ad4
AD4 (uint@0x0240) = 15408 (0x3C30)
TML> ?umaxprot
UMAXPROT (uint@0x029A) = 33945 (0x8499)
TML> umaxprot = 12000
```

Figure 20 – Trigger the over-voltage protection

The bit 12 in the MER error register (over-voltage) will be triggered and the drive will execute the code in the software protections interrupt – the motor will be disabled and the red led will be switched on, while the green led will be off. The status of the drive can be easily seen using the control panel “5_Drive_Status”

To open the 5_Drive_Status window in EasyMotion Studio II, navigate to the "Control panels" menu, then select "5_Drive_Status" or press "Ctrl+5." Once the panel is open, right-click on the window area, then click "START" to activate the control panel.



SRH - Status Register High	
15 - Fault	0
14 - In Cam	0
13 - In freeze control	0
12 - In Gear	0
11 - I2t warning - Drive	0
10 - I2t warning - Motor	0
9 - Target reached	0
8 - Capture event/interrupt	0
7 - LSN event/interrupt	0
6 - LSP event/interrupt	0
5 - Autorun enabled	1
4 - Over position trigger 4	1
3 - Over position trigger 3	1
2 - Over position trigger 2	1
1 - Over position trigger 1	1
0 - ENDINIT executed	1
Supply voltage [V]	24.257

SRL - Status Register Low	
15 - Axis is ON	0
14 - Event set has occured	1
10 - Motion is completed	0
8 - Homing/CALLS active	0
7 - Homing/CALLS warning	0
Drive temperature [°C]	46.542

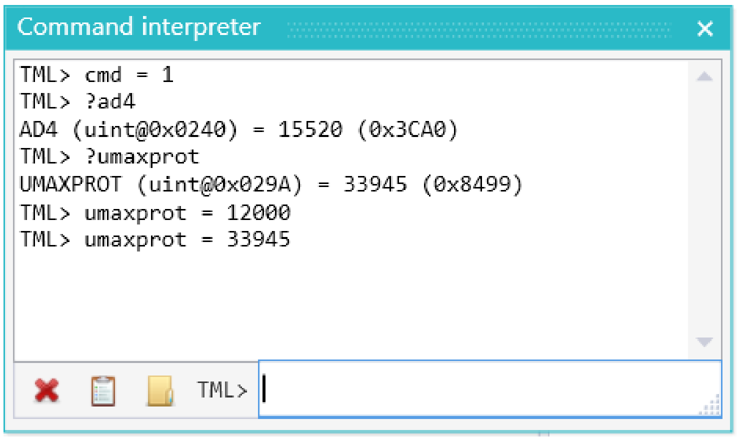
Registers legend:
1 - Yes / True
0 - No / False

IMPORTANT! Check SRH.0 !
Supply voltage and some status or error bits are set ONLY after ENDINIT is executed.
If SRH.0 = 0, download a setup, reset the drive and press the nearby button to send an ENDINIT command. You can also run a TML program, which includes execution of ENDINIT.

MER - Error Register	
15 - Enable is inactive	0
14 - Command error	0
13 - Under voltage	0
12 - Over voltage	1
11 - Over temp. - Drive	0
10 - Over temp. - Motor	0
9 - I2t	0
8 - Over current	0
7 - LSN (limit -) active	0
6 - LSP (limit +) active	0
5 - Feedback error	0
4 - Serial comm. error	0
3 - Control error	0
2 - Invalid setup data	0
1 - Short-circuit	0
0 - CANbus error	0

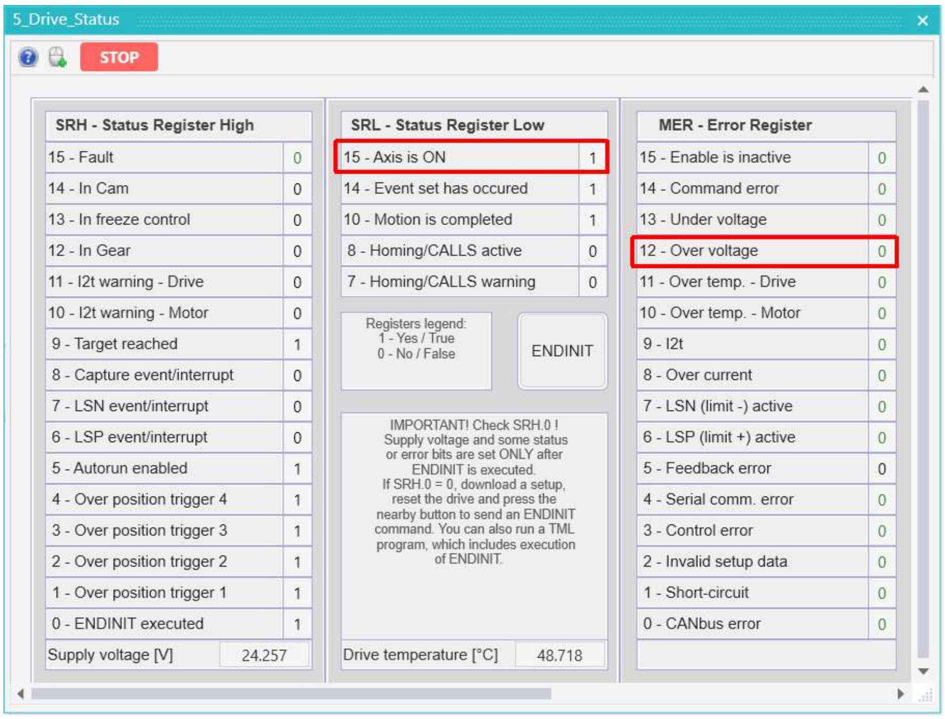
Figure 21 – Over voltage condition

Setting the “UMAXPROT” parameter to the default value, the over-voltage state will disappear and the drive will finish the error recovery procedure: programming the motor to hold position, re-enabling the power stage, switching on the green led / off the red led, resetting the TML stack, the INT2 previous request, enabling the interrupts globally, resetting the faults (MER / DER registers) and jumping to the main program.



```
TML> cmd = 1
TML> ?ad4
AD4 (uint@0x0240) = 15520 (0x3CA0)
TML> ?umaxprot
UMAXPROT (uint@0x029A) = 33945 (0x8499)
TML> umaxprot = 12000
TML> umaxprot = 33945
```

Figure 22 - Reset UMAXPROT to the initial value to exit the over voltage condition



SRH - Status Register High	
15 - Fault	0
14 - In Cam	0
13 - In freeze control	0
12 - In Gear	0
11 - I2t warning - Drive	0
10 - I2t warning - Motor	0
9 - Target reached	1
8 - Capture event/interrupt	0
7 - LSN event/interrupt	0
6 - LSP event/interrupt	0
5 - Autorun enabled	1
4 - Over position trigger 4	1
3 - Over position trigger 3	1
2 - Over position trigger 2	1
1 - Over position trigger 1	1
0 - ENDINIT executed	1
Supply voltage [V]	24.257

SRL - Status Register Low	
15 - Axis is ON	1
14 - Event set has occured	1
10 - Motion is completed	1
8 - Homing/CALLS active	0
7 - Homing/CALLS warning	0

Registers legend:
1 - Yes / True
0 - No / False

IMPORTANT! Check SRH.0 !
Supply voltage and some status or error bits are set ONLY after ENDINIT is executed.
If SRH.0 = 0, download a setup, reset the drive and press the nearby button to send an ENDINIT command. You can also run a TML program, which includes execution of ENDINIT.

Drive temperature [°C]	48.718

MER - Error Register	
15 - Enable is inactive	0
14 - Command error	0
13 - Under voltage	0
12 - Over voltage	0
11 - Over temp. - Drive	0
10 - Over temp. - Motor	0
9 - I2t	0
8 - Over current	0
7 - LSN (limit -) active	0
6 - LSP (limit +) active	0
5 - Feedback error	0
4 - Serial comm. error	0
3 - Control error	0
2 - Invalid setup data	0
1 - Short-circuit	0
0 - CANbus error	0

Figure 23 - The error state is reset, operation resumed, motor is holding position

Once the error recovery procedure is executed and the fault state is reset, the drive will return to the “WaitStart” loop (in the main program) and wait again for the master command.