Graco PD2K Dual Siemens SDK

Release 0.2.1

Graco Inc.

Feb 20, 2023

CONTENTS

CHAPTER

ONE

OVERVIEW

The PD2K Dual Panel Siemens Software Development Kit (SDK) is a collection of sample code for easily controlling a PD2K proportioner through a Siemens PLC. The code is designed to simplify many common integration challenges, allowing you to build a working system very quickly.

1.1 About the PD Platform

The ProMix PD Platform is a family of electric dosing pump systems utilizing positive displacement (PD) technology. More information can be found on the Graco product page.

This SDK is specifically designed for the ProMix PD2K Dual Panel Proportioner for Automatic Spray Applications, which includes two independent 2-component mix units controlled from a Communication Gateway Module (CGM). SDKs for the other ProMix PD products can be found at help.graco.com.

1.1.1 Related Manuals

These manuals are available for download at www.graco.com.

Manual number	Description
3A4486	ProMix PD2K Dual Fluid Panel Electric Proportioner for Automatic Spray Applications, Operation
332458	ProMix PD2K Proportioner for Automatic Spray Applications, Installation
334494	ProMix PD2K CGM Installation Kit, Instructions/Parts

1.2 Compatibility

The examples in this SDK are built using the Siemens TIA Portal platform V17 Update 4 with STEP 7 Professional. Examples for earlier versions are not provided at this time.

All code is compatible with the latest hardware/firmware products for the S7-1200 and S7-1500 families of CPUs. The example project is built specifically for CPU 1211C DC/DC/Rly (article number 6ES7 211-1HE40-0XB0). backwards compatibility with obsolete/end-of-life products in the S7-1200/1500 family may be possible but is not guaranteed.

All code is written in either SCL or LAD language.

CHAPTER

TWO

GETTING STARTED

This guide walks through setting up a Siemens PLC project using the SDK.

2.1 Getting the SDK

The SDK is packaged as a downloadable .zip archive found on help.graco.com. The latest version (as of writing) is GracoPd2kDual_Siemens_SDK_v0.2.1.zip.

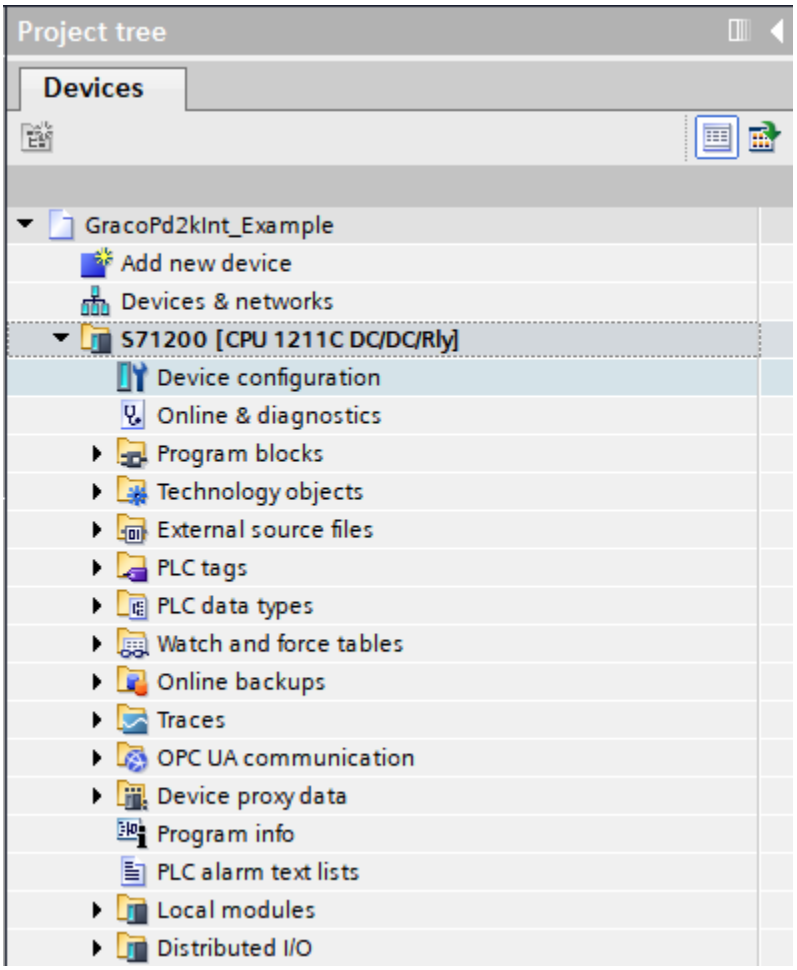
2.2 Using the example project as a template

The SDK comes with a Siemens example project providing some basic functionality right out of the box. Using this project as a starting point is a good way to get to a working system quickly.

The example project can be found in the GracoPd2kDual_Example directory; the project file is GracoPd2kDual_Example.ap17. Since this is a V17 project, make sure you have the correct version of TIA Portal installed on your PC.

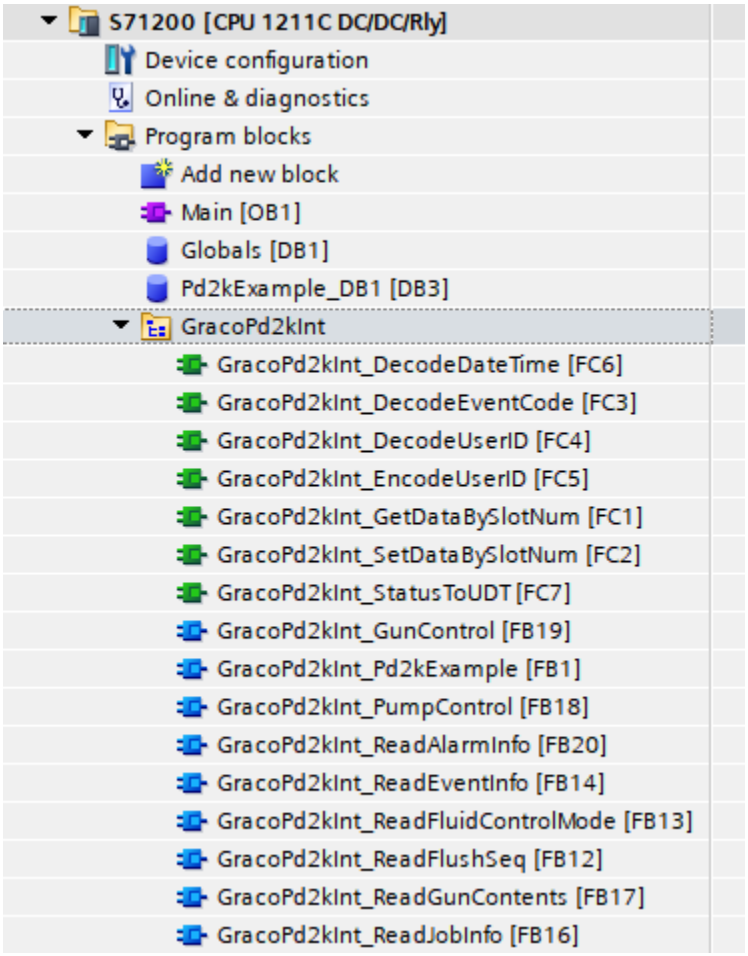
2.2.1 Configure the PLC

The example project comes pre-loaded with a 1211C CPU program:



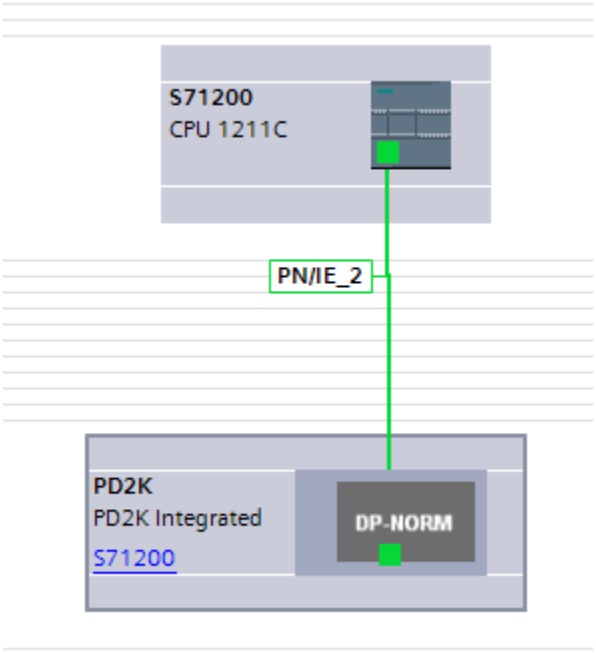
This can be downloaded directly to a matching physical PLC, or changed to a different model by right-clicking on the CPU and selecting “change device”.

All of the FBs, FCs, and UDTs are included in the program under a “GracoPd2kDual” group folder:

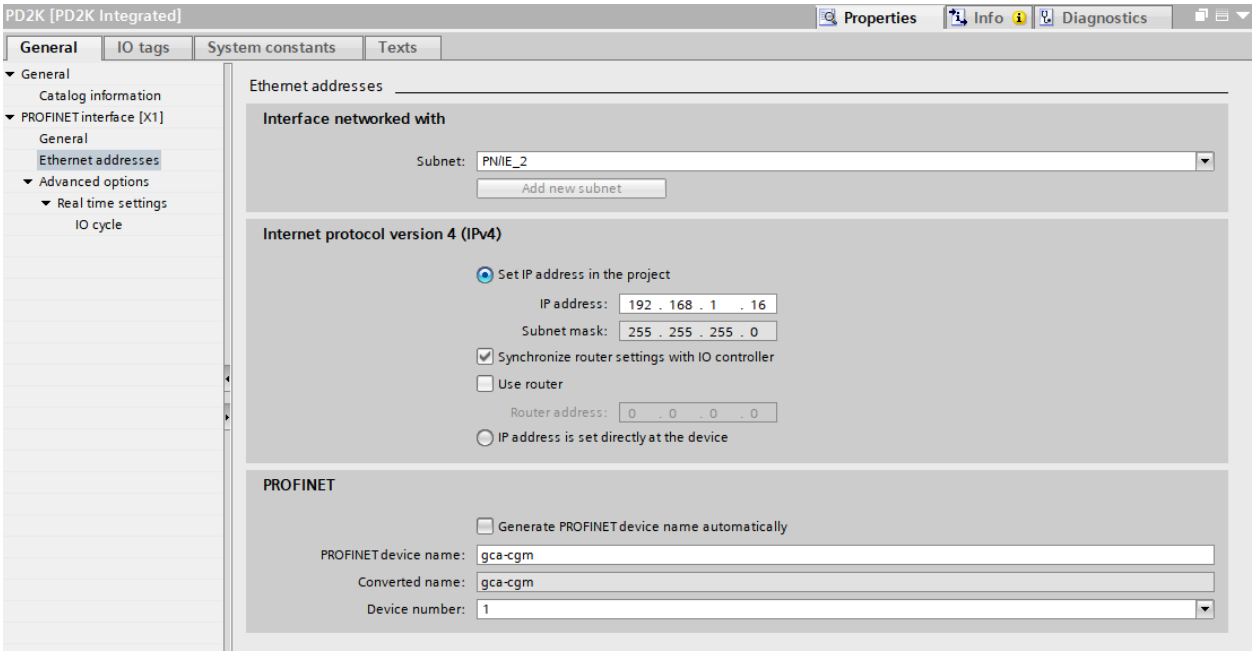


2.2.2 Configure the PD2K

In the network view of the example project, a default PD2K device is connected to the PLC:



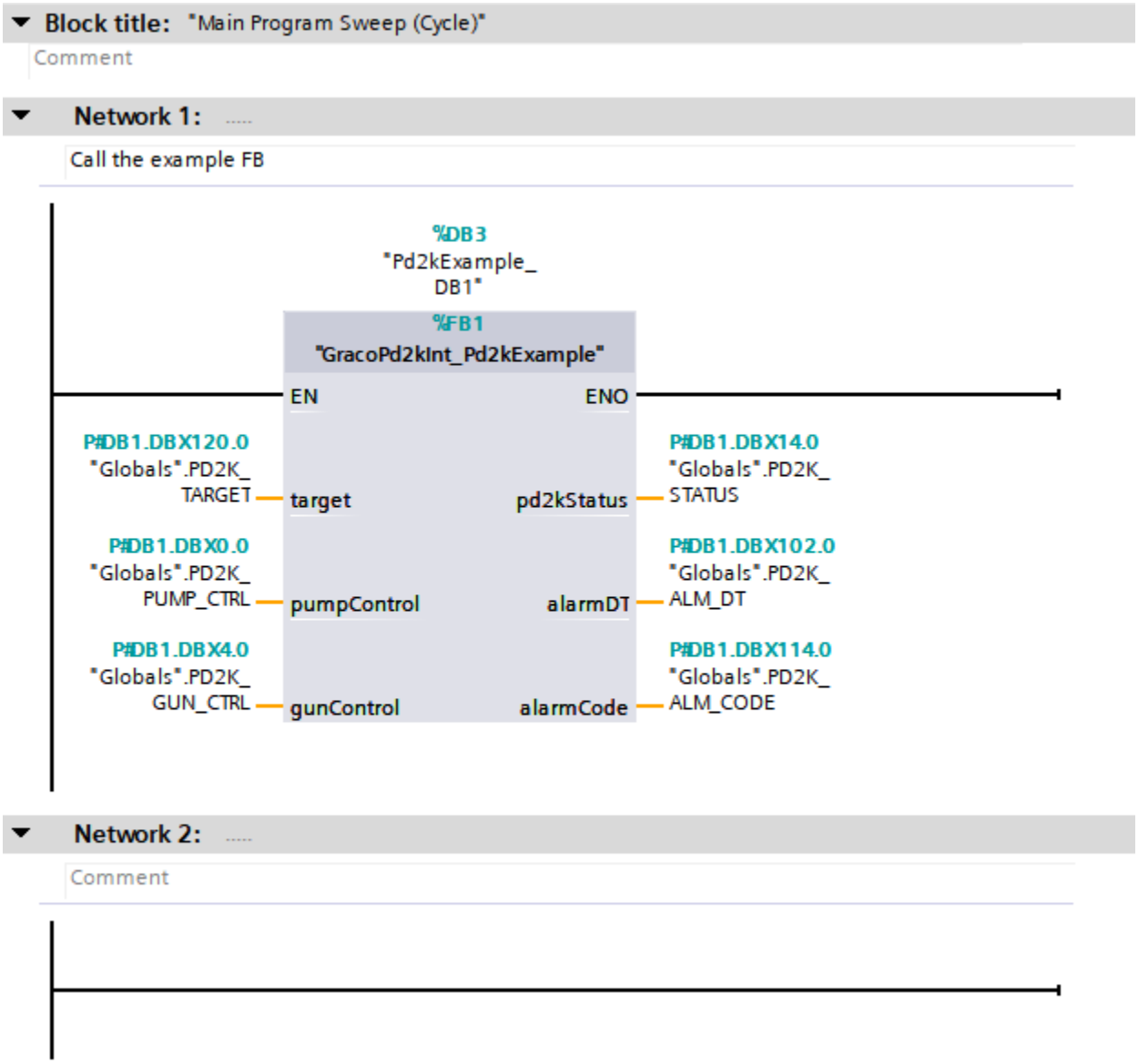
Make sure to set the Profinet settings to match the actual device - these are in Device view -> General -> PROFINET interface -> Ethernet addresses:



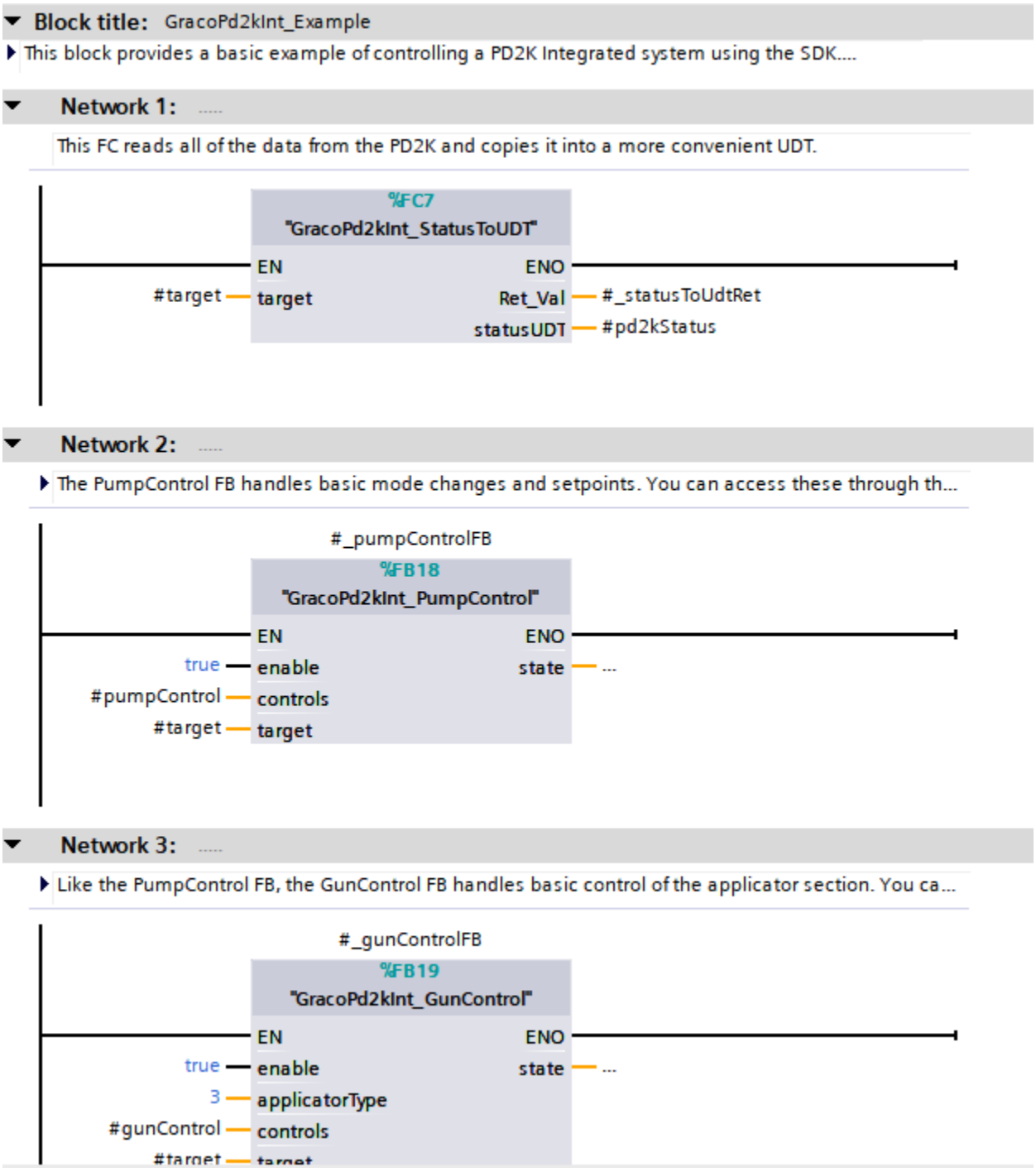
Once the network configuration is setup to match your real device network, the project can be downloaded to your physical PLC.

2.2.3 Project Structure

The Main (OB1) block is the entrypoint for this project. This block simply calls an instance of the Pd2kExample FB:



All of the logic for this example is wrapped in this FB. Open the FB definition to see its code:



The FB is quite simple - it calls a few other FBs/FCs included with the SDK. Each block handles one particular task; for example, the StatusToUDT FC reads data from the PD2K and copies it into a UDT for more convenient lookup. The PumpControl block provides an interface for controlling the system, e.g. writing mode commands via boolean variables. You can read the network comments for more information.

The example FB uses an instance datablock named “Pd2kExample_DB1”. Typically, you do not need to touch this, as all of the relevant data is passed through parameters.

At the root of the project, there is a datablock named “Globals”. This block contains everything you as the user can access for controlling the PD2K:

Globals

	Name	Data type	Offset	Start value	Re
1	Static				
2	PD2K_PUMP_CTRL	"GracoPd2kInt_TypePumpControls"	0.0		
3	PD2K_GUN_CTRL	"GracoPd2kInt_TypeGunControls"	4.0		
4	PD2K_STATUS	"GracoPd2kInt_TypePd2kIntStatus"	14.0		
5	PD2K_ALM_DT	DTL	102.0	DTL#1970-01-01-(	
6	PD2K_ALM_CODE	String[4]	114.0	''	
7	PD2K_TARGET	"GracoPd2kInt_TypeTarget"	120.0		

Each of these variables correspond to parameters for the Pd2kExample FB called in Main.

The PD2K_STATUS variable contains all of the status data arranged in a convenient UDT form. This tends to be much easier compared to working with the *I* process data directly. For more info, see *StatusToUDT*.

The PD2K_PUMP_CTRL_MU1 and PD2K_PUMP_CTRL_MU2 variables provide access to the system mode commands and setpoints. For more info, see *PumpControl*.

The PD2K_ALM_CODE and PD2K_ALM_DT variables are the most recent alarm code and datetime values, respectively. These are updated through the *ReadAlarmInfo* FB.

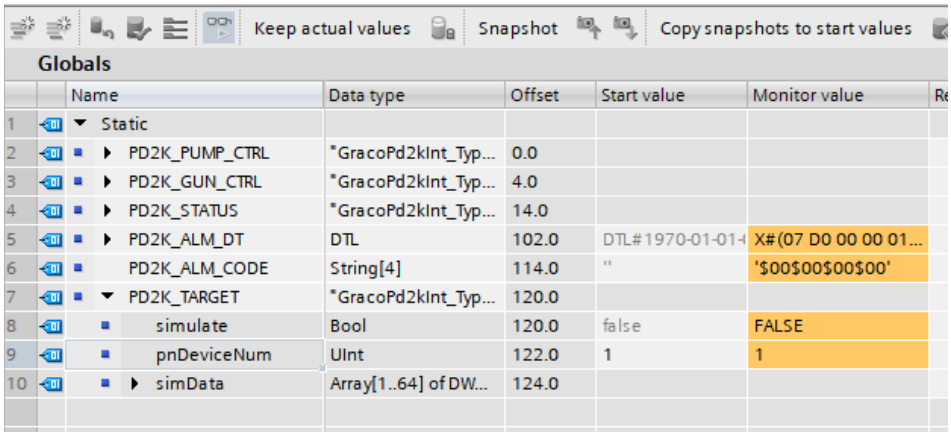
Finally, the PD2K_TARGET variable is the *TypeTarget* instance used to point to the actual PD2K device defined in the project network configuration. see *Using TypeTarget*;

Note that no PLC tags are being used with this example. Because of how the project is structured, you can simply access everything you need through the Globals datablock instead (but you can always add your own tags if you wish).

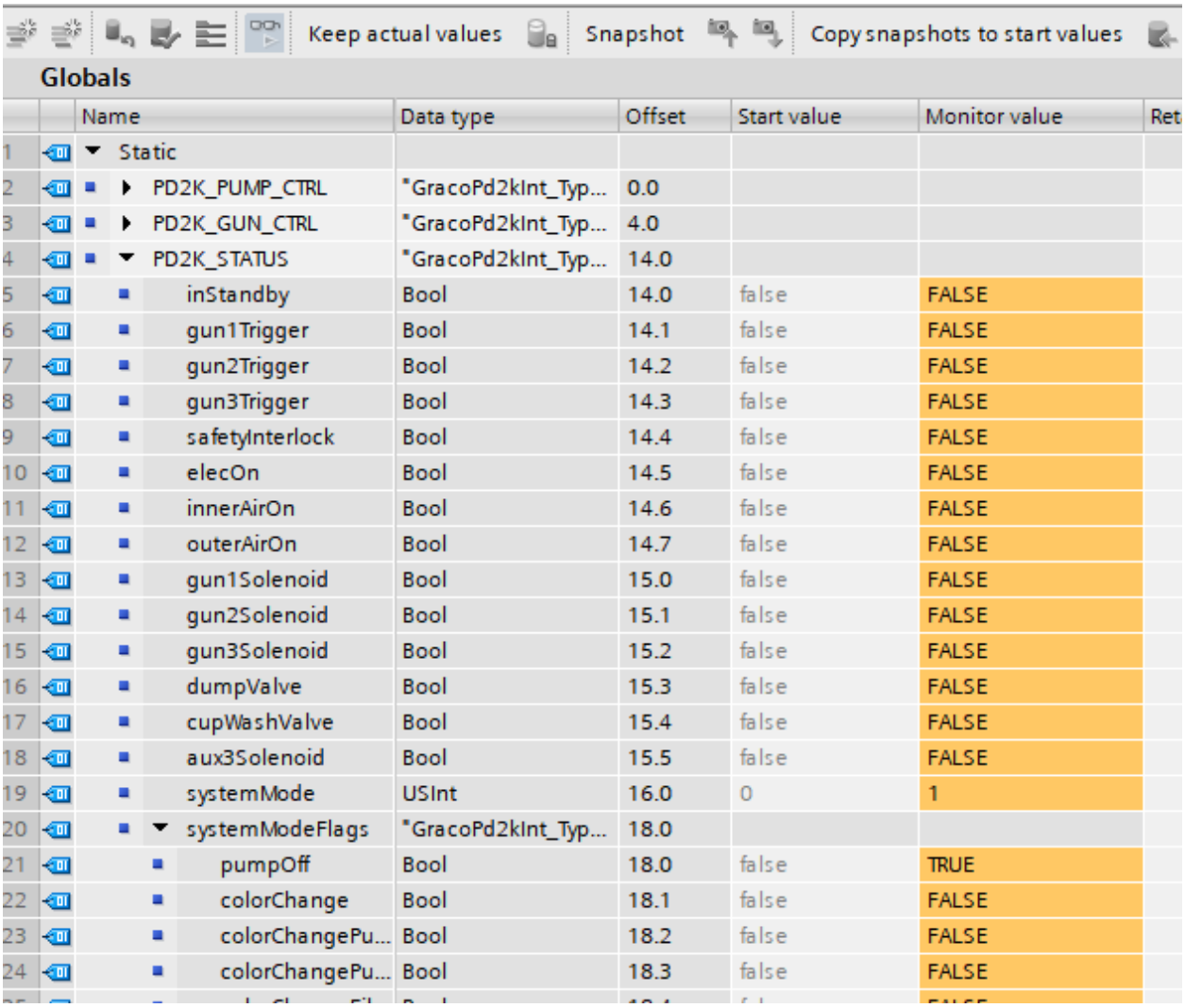
2.2.4 Running the PD2K

Once you have downloaded the example project to a PLC, you can start running the PD2K using the provided code.

While online with the PLC, open the Global DB. Expand the PD2K_TARGET variable. Confirm that the `simulate` variable is set to FALSE. Make sure the `pnDeviceNum1` and `pnDeviceNum2` are set to the Profinet device numbers assigned to the PD2K CGMs:

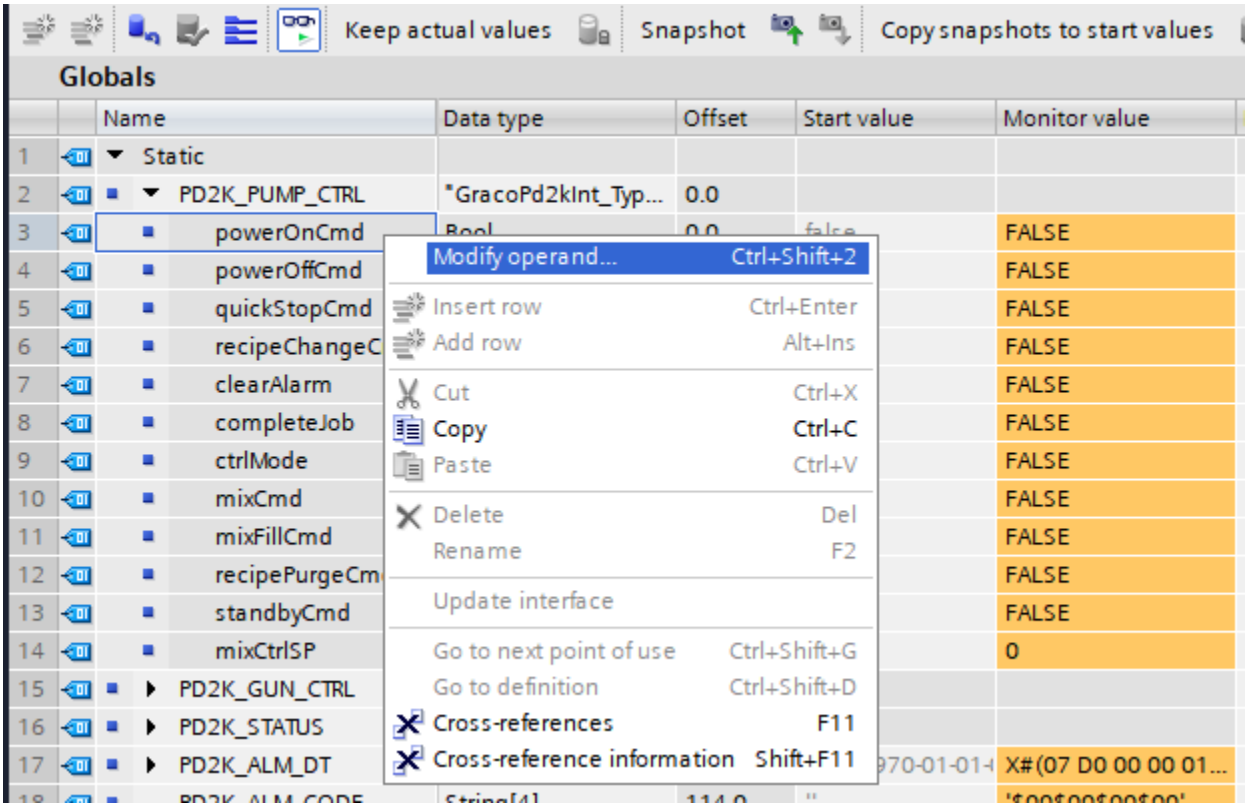


Expand the PD2K_STATUS variable. Assuming the PD2K is currently in "pump off" mode, you should see a "1" for the system mode along with the "pumpOff" system mode flag set to TRUE:

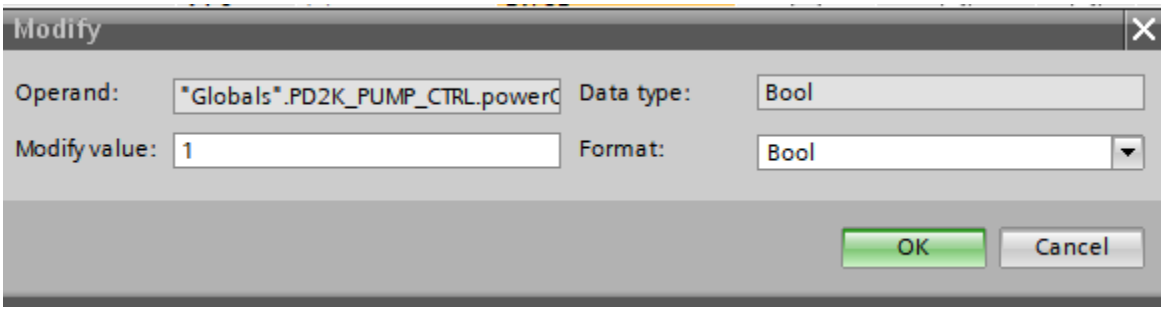


	Name	Data type	Offset	Start value	Monitor value	Ret
1	Static					
2	PD2K_PUMP_CTRL	"GracoPd2kInt_Typ...	0.0			
3	PD2K_GUN_CTRL	"GracoPd2kInt_Typ...	4.0			
4	PD2K_STATUS	"GracoPd2kInt_Typ...	14.0			
5	inStandby	Bool	14.0	false	FALSE	
6	gun1Trigger	Bool	14.1	false	FALSE	
7	gun2Trigger	Bool	14.2	false	FALSE	
8	gun3Trigger	Bool	14.3	false	FALSE	
9	safetyInterlock	Bool	14.4	false	FALSE	
10	elecOn	Bool	14.5	false	FALSE	
11	innerAirOn	Bool	14.6	false	FALSE	
12	outerAirOn	Bool	14.7	false	FALSE	
13	gun1Solenoid	Bool	15.0	false	FALSE	
14	gun2Solenoid	Bool	15.1	false	FALSE	
15	gun3Solenoid	Bool	15.2	false	FALSE	
16	dumpValve	Bool	15.3	false	FALSE	
17	cupWashValve	Bool	15.4	false	FALSE	
18	aux3Solenoid	Bool	15.5	false	FALSE	
19	systemMode	USInt	16.0	0	1	
20	systemModeFlags	"GracoPd2kInt_Typ...	18.0			
21	pumpOff	Bool	18.0	false	TRUE	
22	colorChange	Bool	18.1	false	FALSE	
23	colorChangePu...	Bool	18.2	false	FALSE	
24	colorChangePu...	Bool	18.3	false	FALSE	

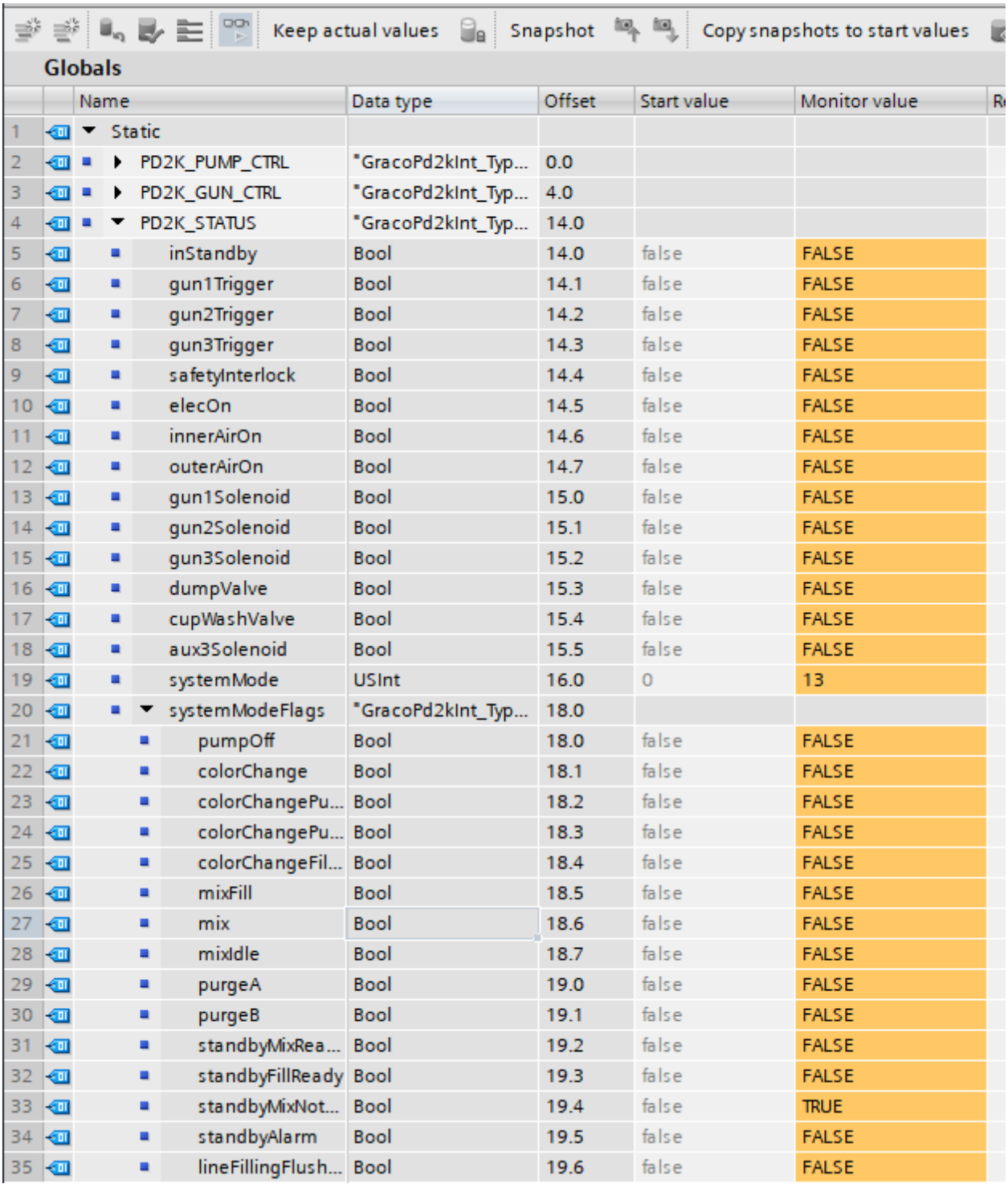
Next, expand the PD2K_PUMP_CTRL variable. Right-click the powerOnCmd member and select “modify operand”:



Set the modify value to 1 (i.e. TRUE) and click OK:



The *PumpControl* FB sees this boolean and pulses the system mode command to “power on” (note the FB also clears the powerOnCmd boolean). If there are no alarms present, the pumps should power on, followed by the system mode status showing “standby”:



	Name	Data type	Offset	Start value	Monitor value
1	Static				
2	PD2K_PUMP_CTRL	"GracoPd2kInt_Typ...	0.0		
3	PD2K_GUN_CTRL	"GracoPd2kInt_Typ...	4.0		
4	PD2K_STATUS	"GracoPd2kInt_Typ...	14.0		
5	inStandby	Bool	14.0	false	FALSE
6	gun1Trigger	Bool	14.1	false	FALSE
7	gun2Trigger	Bool	14.2	false	FALSE
8	gun3Trigger	Bool	14.3	false	FALSE
9	safetyInterlock	Bool	14.4	false	FALSE
10	elecOn	Bool	14.5	false	FALSE
11	innerAirOn	Bool	14.6	false	FALSE
12	outerAirOn	Bool	14.7	false	FALSE
13	gun1Solenoid	Bool	15.0	false	FALSE
14	gun2Solenoid	Bool	15.1	false	FALSE
15	gun3Solenoid	Bool	15.2	false	FALSE
16	dumpValve	Bool	15.3	false	FALSE
17	cupWashValve	Bool	15.4	false	FALSE
18	aux3Solenoid	Bool	15.5	false	FALSE
19	systemMode	USInt	16.0	0	13
20	systemModeFlags	"GracoPd2kInt_Typ...	18.0		
21	pumpOff	Bool	18.0	false	FALSE
22	colorChange	Bool	18.1	false	FALSE
23	colorChangePu...	Bool	18.2	false	FALSE
24	colorChangePu...	Bool	18.3	false	FALSE
25	colorChangeFil...	Bool	18.4	false	FALSE
26	mixFill	Bool	18.5	false	FALSE
27	mix	Bool	18.6	false	FALSE
28	mixIdle	Bool	18.7	false	FALSE
29	purgeA	Bool	19.0	false	FALSE
30	purgeB	Bool	19.1	false	FALSE
31	standbyMixRea...	Bool	19.2	false	FALSE
32	standbyFillReady	Bool	19.3	false	FALSE
33	standbyMixNot...	Bool	19.4	false	TRUE
34	standbyAlarm	Bool	19.5	false	FALSE
35	lineFillingFlush...	Bool	19.6	false	FALSE

You should now be able to set any of the other controls found in PD2K_PUMP_CTRL and control the system. For example, write a desired flow rate value to PD2K_PUMP_CTRL.mixCtrlSP, then set recipeChangeCmd to TRUE to perform a recipe change.

2.2.5 Where to go from here

Now that you have explored the example project, you should have a better understanding of how to use the SDK in your own larger applications. For example:

- The variables provided in the Globals DB can be connected to an HMI for touchscreen control of the system.
- You can use the Globals DB variables within your own custom application sequence. For example, the PLC can monitor the pot life time of the current recipe and automatically trigger a purge when neccessary.
- Because of the modular design of the FBs/FCs, you can expand your program to control multiple PD2K systems from the same PLC. Simply create a *TypeTarget* variable for each instance, assign that instance's profinet device number, and pass that into your function calls.

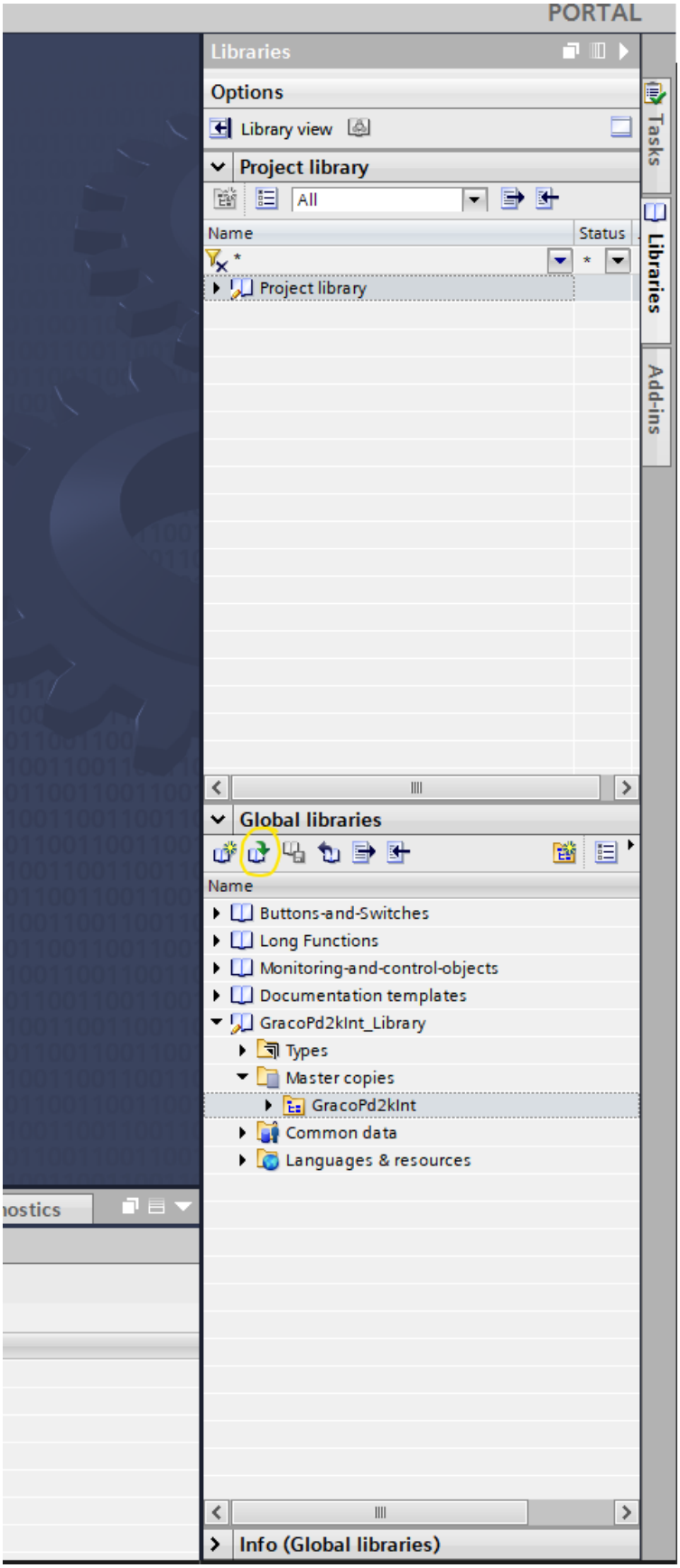
For more information about all the available program blocks, see the *API Documentation*.

2.3 Adding the SDK to an existing project

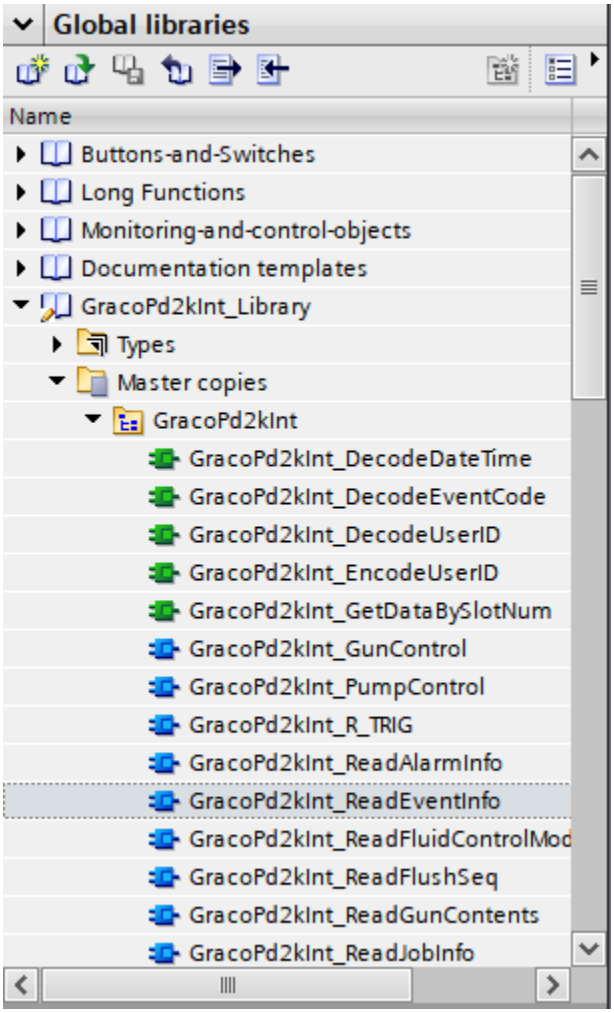
Besides using the example project, the contents of the SDK can be pulled into an existing TIA Portal project as well. The recommended way is using the provided global library named "GracoPd2kDual_Library". This contains all of the FBs/FCs/UDTs from the example project, each of which can be added to an existing TIA Portal project.

2.3.1 Using the global library

With a TIA Portal project open, select the Libraries pane and click the "open global library" button:


PORTAL
Libraries
Options
Library view
Project library
All
Name
Status
Project library
Global libraries
Name
Buttons-and-Switches
Long Functions
Monitoring-and-control-objects
Documentation templates
GracoPd2kInt_Library
Types
Master copies
GracoPd2kInt
Common data
Languages & resources
Info (Global libraries)
Tasks
Libraries
Add-ins

Select the GracoPd2kDual_Library.al17 file in the file navigator and click Open. The blocks can be found in this library under the "Master copies" folder:



Drag any of the program blocks into the project to use them. Note that the blocks must be dropped into their given folders, e.g. FBs/FCs must go in Program blocks, UDTs must go in PLC data types. Once copied over, these blocks can be used in the program.

Note: The blocks in the global library are stored as "master copies", meaning they are not directly linked to the copies in the PLC program. Any changes made in the PLC will not affect the global library (and vice-versa).

"Library types", on the other-hand, provide a link between the the global library and the blocks in the PLC. However, we do not utilize this feature in the SDK because it conflicts with the Siemens Version Control Interface (VCI), and SDK development is done using that tool instead.

CHAPTER

THREE

DESIGN

This chapter discusses various decisions made in the design of the SDK.

3.1 Command model standard

Each of the FBs/FCs in this SDK follow a convention called the "Command model standard". This convention is loosely based on the PLCopen design recommendations for motion control blocks.

Generally speaking, every FB/FC is designed to perform a particular command or set of commands. Their interfaces all follow the same conventions, so all blocks of a given model type are called in the same way.

There are three different command models:

- Function model
- Execute model
- Enable model

The function model is for simple stateless operations (i.e. FCs), while the execute and enable models run over time and may have internal memory (i.e. FBs). These models will be covered in more detail below.

3.1.1 The state output parameter

Each command can be thought of as a state machine, where each instance of the command exists in exactly one state at any point in time. These states are represented by a UDT output parameter called `state`. Within this UDT, there is a member `state.code` that represents the current state as a 16-bit Word value.

The values for `state.code` are standardized such that the most-significant byte corresponds to a particular category. These categories are defined in the following table:

State category	State value (hex)	Description
IDLE	16#0xxx	The command is not running (enable model only)
BUSY	16#1xxx	The command is running (execute model only)
DONE	16#2xxx	The command completed successfully (execute/function models only)
VALID	16#3xxx	The command is running without issues/errors (enable model only)
ABORTED	16#4xxx	The command was aborted (either locally or by a higher-priority command)
ERROR	16#8xxx	An error occurred while processing the command
ERROR_BUSY	16#9xxx	An error occurred, but the command is busy attempting to recover (enable model only)

There can be multiple independent states within each category - for example, states 16#8000, 16#8001, and 16#8002 would represent distinct error conditions for that command.

The `state` parameter also expresses the command's state in the form of boolean flags, each corresponding to the bits in `state.code`'s most significant byte. The flags are defined in the following table:

State flag	bit number	Description
BUSY	0 (2#0001)	The command is running
DONE	1 (2#0010)	The command completed successfully (execute/function models only)
VALID	1 (2#0010)	The command is running without issues/error (enable model only)
ABORTED	2 (2#0100)	The command was aborted (either locally or by a higher-priority command)
ERROR	3 (2#1000)	An error occurred while processing the command

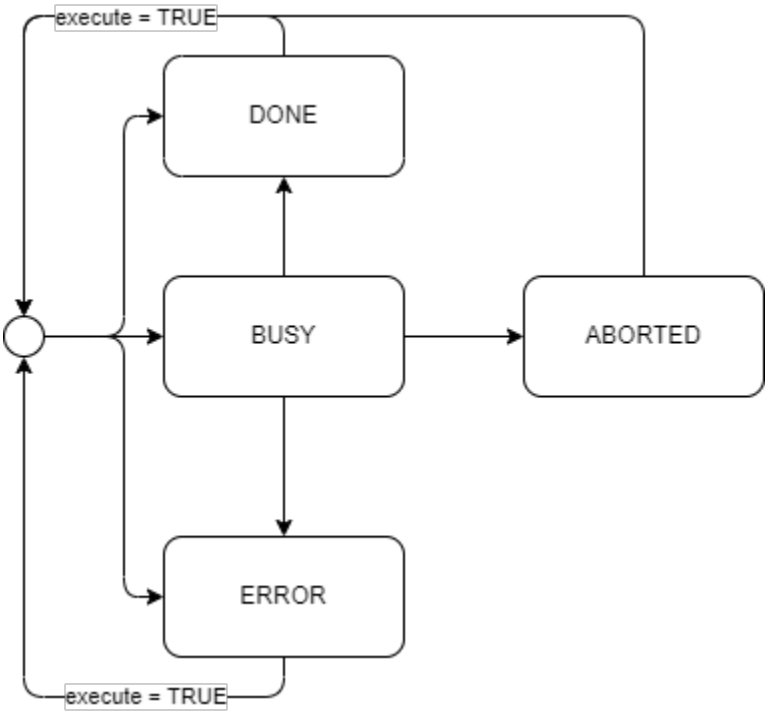
For example, if `state.code` is set to 16#8000, the ERROR flag would be on and the rest would be off. In many cases, using the flags is more convenient than the state code value. More details regarding the state categories and flags for each of the command models will be covered in the next sections.

3.1.2 Function Model

The function model represents a simple, stateless operation that processes immediately. All FCs in the SDK follow this model, since FCs, by definition, have no internal memory. An example is DecodeDateTime - this command reads the source input parameters and computes a native datetime value.

Function model commands typically use only DONE and ERROR states. If something goes wrong during execution (e.g. a parameter is out of range), an error code can be returned to notify the caller. The state codes are normally passed through the return value of the FC, but can also be passed as an output parameter. If a command has no error conditions, then the state code can be omitted altogether.

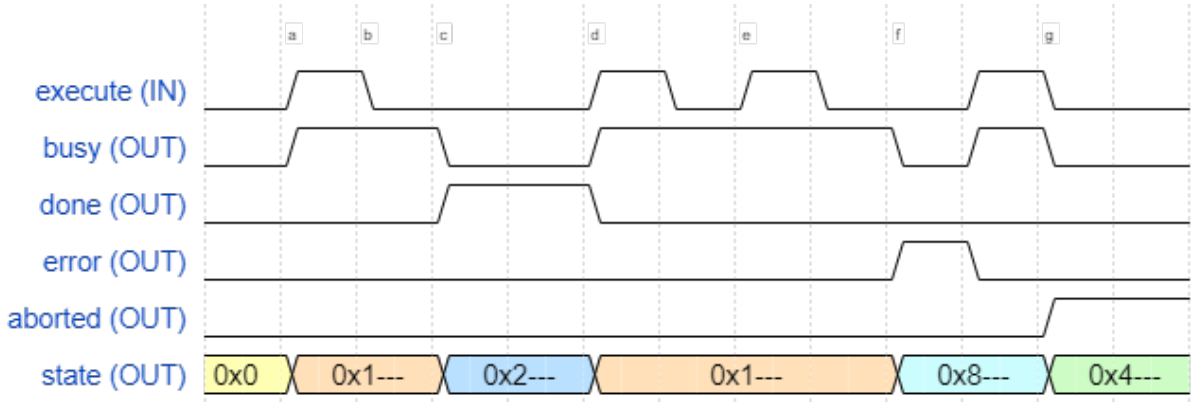
3.1.3 Execute model



The execute model is used for commands that run from start to finish. Once started, the command runs until it either completes the operation (DONE), experiences an error (ERROR), or is aborted (ABORTED). This model is stateful; its behavior depends on previous state and can change over time.

Execute model commands are triggered using an `execute` boolean input parameter. Each rising edge on this parameter executes the command once. The command only re-triggers once the command is in a "terminal" state - i.e., DONE, ERROR, or ABORTED. Triggering the command while already running does nothing.

Here is a timing diagram for the execute model:

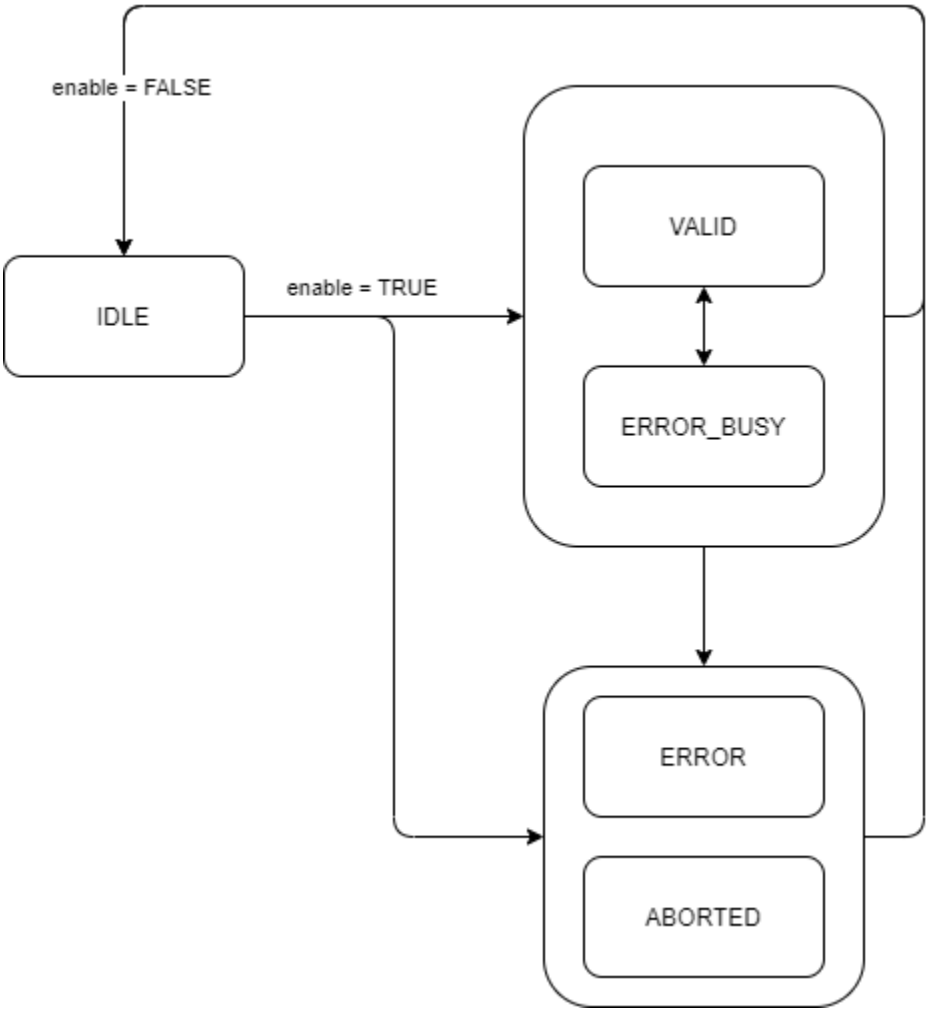


a.	A rising edge on execute starts the command, setting the busy flag to true.
b.	Execute is automatically cleared by the command.
c.	When the command completes, busy is set to false and done is set to true.
d.	When command is executed again, done is set to false and busy is set to true.
e.	Additional execute signals while busy are ignored, rather than restarting the command.
f.	If a problem occurs while processing the command, the error flag is set to true.
g.	Similarly, if something interrupts the command, the aborted flag is set to true.

Note that the `execute` input will automatically be cleared by the FB; there is no need to reset the trigger from outside logic. This provides additional flexibility, e.g. when controlling from browser-based HMIs that lack momentary push buttons. The input can still be written from a regular coil - either way, the command will behave the same.

Some commands (but not all) may include a `cancel` input for stopping a command early. `cancel` inputs work the same way as `execute`, except they trigger a transition from BUSY to ABORTED.

3.1.4 Enable model

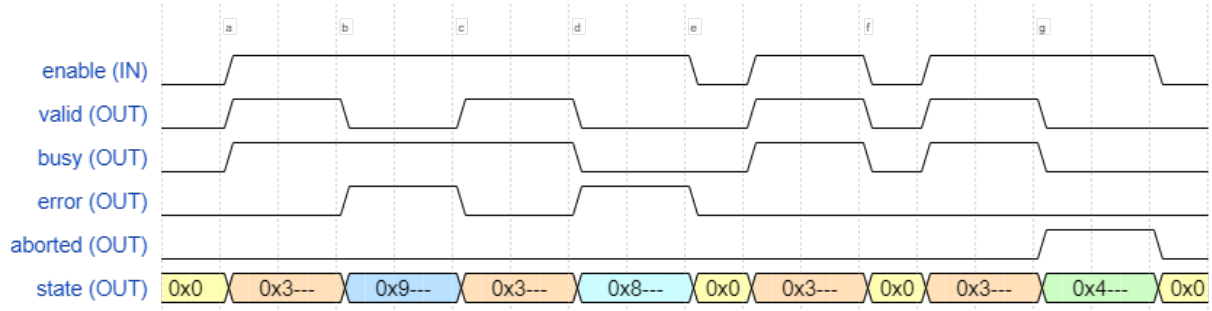


The enable model is used for commands that run indefintely. This differs from the execute model in that the enable model does not have a DONE condition. Instead, this model refers to normal operation as VALID. The enable model is useful for things like jog operations.

Enable model commands use an `enable` boolean input to control their state. When `enable` is false, the command is IDLE and does nothing. When true, the command runs its logic sequence and writes to its outputs.

An example of the enable model is the PumpControl FB. When enabled, the FB responds to inputs and updates the system mode accordingly. Disabling the FB allows it to be "detached" from the device's registers, which could then be controlled through some other logic.

Here is a timing diagram for the enable model:



a.	A rising edge on enable starts the command. The busy and valid flags are set to true.
b.	When an error occurs, error is set to true and valid is set to false. In this case, the block logic can automatically recover from the error, so the busy flag stays at true.
c.	The block logic handled the error - valid is set back to true and error is set back to false.
d.	In this case, an error occurred that requires user intervention. Error is set to true, and both valid and busy are set to false.
e.	Setting enable to false resets the block and clears the error.
f.	Setting enable to false also sets both valid and busy to false.
g.	The command may also be aborted, causing the aborted flag to be true and valid/busy to be false.

Unlike the execute model, the state flags for the enable model are not all mutually-exclusive. Under normal operation, the VALID and BUSY flags are true, meaning the command is running without issues. If an issue does occur, there are two possibilities - either it is something the FB can deal with automatically, or it is something outside of its control.

In the former scenario, the command enters an ERROR_BUSY state, which sets both the ERROR and BUSY flags to true. This means the command has identified an issue and is working to resolve it, e.g. retrying a failed message.

In the latter scenario, the command enters an ERROR state, without the BUSY flag. The command has experienced an issue that it cannot fix, and it is up to the calling code to resolve it. When this happens, the command must be reset by setting `enable` to false. It can then be re-enabled once the issue has been dealt with.

The ABORTED state works the same way as the ERROR state, but this is typically used for situations where another process in the program has overwritten some value the command was using. Because enable model commands can be disabled at any point, they do not need a `cancel` input, like in the execute model.

Note that, while enable model commands are designed to be stopped at any time, there may be several cycles where the command performs some cleanup logic before returning to IDLE state.

3.2 Sharing access to device registers

Many of the FBs/FCs in this SDK write to the same sets of Profinet device registers. For example, The ReadAlarmInfo and ReadRecipe FBs both call SendDCS internally, which uses the DCS registers to query data from the PD2K. Only one DCS request can be sent at a time, so care must be taken to avoid writing the registers from multiple locations in logic at the same time. For example, if ReadAlarmInfo and ReadRecipe were both executed at the same time and allowed to write to the DCS registers, they would interfere with one another and result in a race condition.

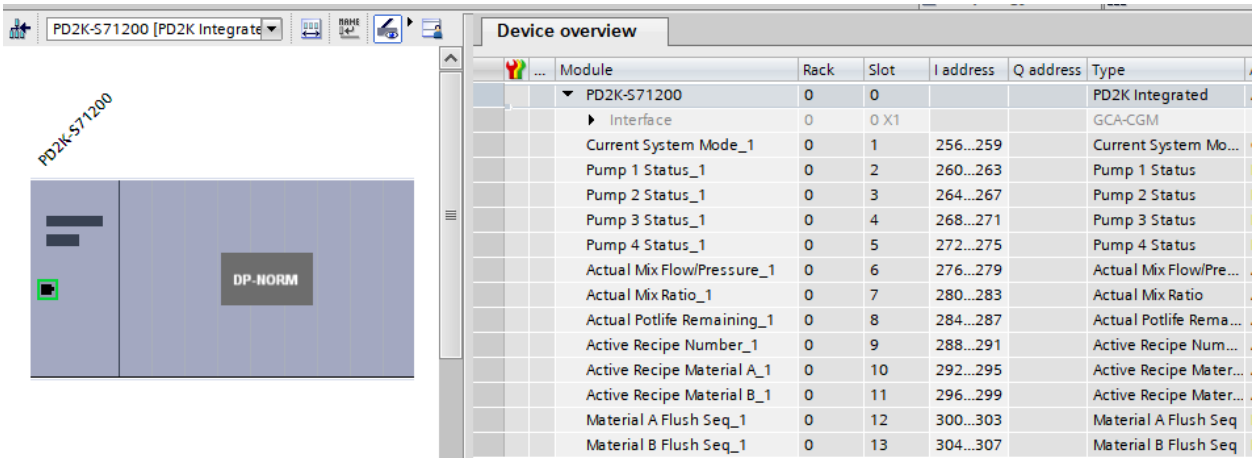
To overcome this, each of the FBs in this SDK are designed using a concept known as a semaphore. In general, semaphores prevent simultaneous writes to a shared resource (like the device registers) using a locked/unlocked mechanism. In short, the FBs will take turns without interfering with one another.

Take the example of executing both the ReadAlarmInfo and ReadRecipe FBs at the same time. Before either FB does anything, they will each read the DCS acknowledge and DCS command register values. If both values are 0 (i.e. no operation, or NOP), then that FB can assume no other FB is currently using the DCS registers and thus, continue its sequence. If either register is non-zero, however, then some other FB has accessed it first, so the FB will go into a wait state until either the registers become available again or the FB times out.

Note that this mechanism only works because all FBs follow the same rules of only accessing registers when they are considered unlocked. Nothing prevents the user's program from writing to a register directly, and doing so will cause conflicts with the SDK FBs.

It's also worth noting that this coordinating behavior works best with only 2-3 active FBs at a time. If many blocks are waiting, It's possible some of them will never be called because the earlier blocks will always take control first. In these situations, it is recommended to structure the program into a sequence where only a couple blocks are called simultaneously.

3.3 Accessing Profinet device data



...	Module	Rack	Slot	I address	Q address	Type
	PD2K-S71200	0	0			PD2K Integrated
	Interface	0	0 X1			GCA-CGM
	Current System Mode_1	0	1	256...259		Current System Mo...
	Pump 1 Status_1	0	2	260...263		Pump 1 Status
	Pump 2 Status_1	0	3	264...267		Pump 2 Status
	Pump 3 Status_1	0	4	268...271		Pump 3 Status
	Pump 4 Status_1	0	5	272...275		Pump 4 Status
	Actual Mix Flow/Pressure_1	0	6	276...279		Actual Mix Flow/Pre...
	Actual Mix Ratio_1	0	7	280...283		Actual Mix Ratio
	Actual Potlife Remaining_1	0	8	284...287		Actual Potlife Rema...
	Active Recipe Number_1	0	9	288...291		Active Recipe Num...
	Active Recipe Material A_1	0	10	292...295		Active Recipe Mater...
	Active Recipe Material B_1	0	11	296...299		Active Recipe Mater...
	Material A Flush Seq_1	0	12	300...303		Material A Flush Seq
	Material B Flush Seq_1	0	13	304...307		Material B Flush Seq

When using the Profinet protocol, the device data is accessed via memory addresses in the process image of the PLC. The addresses can be freely assigned and do not have to be in any particular order. Unfortunately, this makes writing modular code like the SDK library difficult. Our FBs/FCs need some way to locate the registers without hardcoding in the address values.

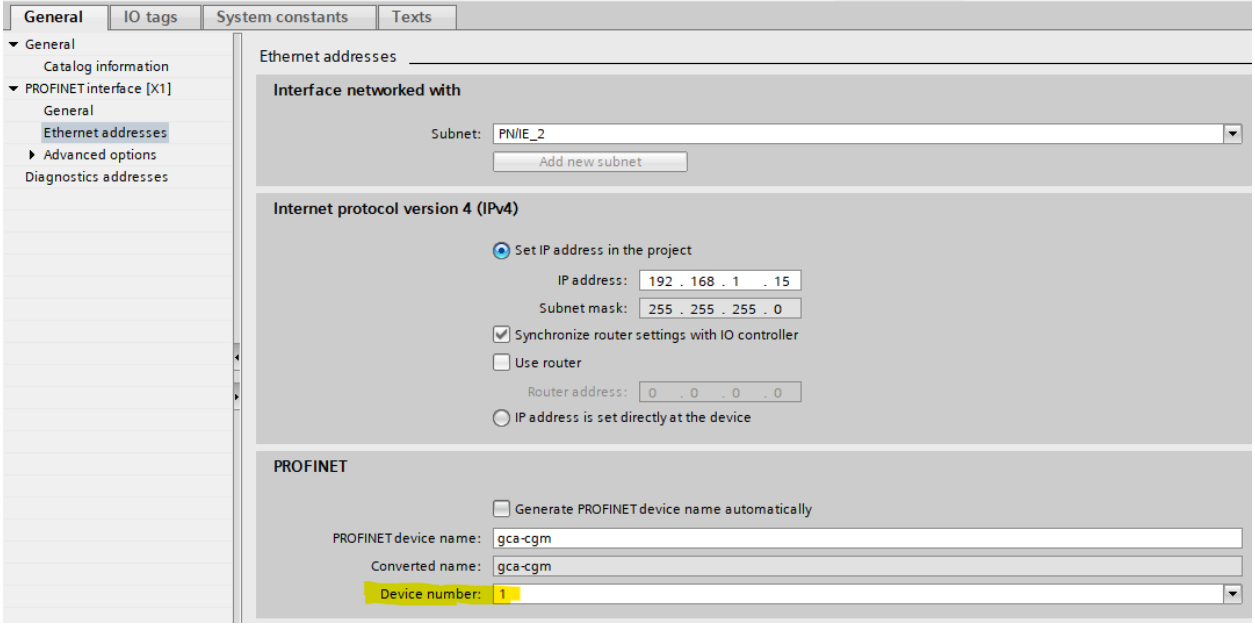
One approach would be to pass each of the required input addresses as parameters to the FB/FC, and to write the output addresses through output parameters. However, this approach becomes tedious when many different registers are needed. For example, the SendDCS FB needs access to all DCS registers - over 16 total. It's also very easy for the programmer to mistakenly connect a register to the wrong parameter, resulting in bugs that are difficult to troubleshoot.

Another approach would be to use PLC tags assigned to the Profinet addresses. This is typical when writing a program for the end user. However, this means the code will depend on a specific set of tags being present in the PLC. Since the tags would be hardcoded into the logic, you would not be able to control multiple PD2Ks from the same PLC without first rewriting the code.

Fortunately, there is a built-in solution that provides easy, modular access to Profinet IO devices that does not involve rewriting any code!

3.3.1 Accessing a Profinet IO device by device number

Within a given Profinet network, every device is assigned a unique device number. This number can be found in its device configuration under General -> Profinet Interface -> Ethernet addresses -> Profinet -> Device number:



Each Profinet device has "slots" for each of its data registers. These slot numbers are fixed for every PD2K system and defined in the GSDML file (e.g. "current system mode" will always be slot 1). Together, the device number and slot number define a "geographic location" for a given piece of data.

Siemens provides a built-in way to look up the memory address of a slot from its geographic location. The solution is using the GEO2LOG FC (for S7-300/400, the equivalent is GEO_LOG). By passing a device number and slot number to this FC, we can look up the slot's corresponding memory location and access it like any other address. Since all Graco products have fixed slot numbers, we can access all device data from a FB/FC just by passing in the device number - no tags required!

This SDK provides two FCs called `GetDataBySlotNum` and `SetDataBySlotNum`. These are wrappers around GEO2LOG/GEO_LOG, and they are used to read/write data from a Profinet device. The device number is passed to the `pnDeviceNum` input parameter, and the desired slot number is passed to `slotNum`. The other FBs/FCs use these FCs internally to access the device registers by their given slot numbers.

3.3.2 Using TypeTarget

The *TypeTarget* UDT is used extensively throughout the SDK code. This type has a member named "pnDeviceNum" should be assigned a specific PD2K Profinet device number. The SDK code uses this value to access device data through the method described above.

Note: *TypeTarget* also supports running in "simulation" mode, where instead of operating on a real Profinet device, all data flows through a "simData" array member in the UDT. This can be useful for testing logic when a real PD2K is not available. Setting the "simulate" member to true enables this mode. Just make sure to set this to "false" when in production, otherwise the system will not operate correctly!

CHAPTER

FOUR

API DOCUMENTATION

This chapter provides a reference of all functions, function blocks, and UDTs defined in the SDK.

Note: All FBs/FCs/UDTs start with the prefix "GracoPd2kDual_" in the SDK project files. This helps prevent naming collisions if multiple SDKs are used in the same PLC program.

4.1 GracoPd2kDual API

4.1.1 Functions (FC)

GetDataBySlotNum

Read data from the specified Profinet device number(defined in target parameter) and a slot number. This FC uses the GEO2LOG and RD_ADDR built-in functions internally to lookup the process data address. The status codes from those functions are forwarded through the return value of this function.

cgmNum (INPUT UInt)
CGM number (1 or 2)

pnSlotNum (INPUT UInt)
Profinet slot number

target (INOUT *TypeTarget*)
target to operate on

data (OUTPUT DWord)
Current process data

State codes

- **16#8000** - Error - invalid CGM number

RETURN (**`GracoCore_TypeFunctionCmdState`_**)

SetDataBySlotNum

Writes data to a specified Profinet device number (defined in target parameter) and a slot number. This FC uses the GEO2LOG and RD_ADDR built-in functions internally to lookup the process data address. The status codes from those functions are forwarded through the return value of this function.

cgmNum (INPUT UInt)
CGM number (1 or 2)

pnSlotNum (INPUT UInt)
Profinet slot number

data (INPUT DWord)
Data to write

target (INOUT *TypeTarget*)
target to operate on

State codes

- **16#8000** - Error - invalid CGM number

RETURN (**`GracoCore_TypeFunctionCmdState`_**)

StatusToUDT

Read status data from a PD2K Integrated device and collect it into a user-defined data type. This provides a more-convenient view of the available data compared to accessing it directly. If an error occurs while attempting to read the device data, the state codes from GetDataBySlotNum will be forwarded through the return value of this function.

target (INOUT *TypeTarget*)
Target to operate on

statusUDT (OUTPUT *TypePd2kStatus*)
Status data packed in a user-defined data type

RETURN (**`GracoCore_TypeFunctionCmdState`_**)

4.1.2 Function Blocks (FB)

PumpControl

An interface into the pump controls for the PD2K Integrated system. This allows you to control the pump system state (e.g. mixing, standby, recipe change), clear alarms, complete a job, and set the mix control setpoint. The control inputs come from a user-defined data type - see TypePumpControlInputs.

enable (INPUT Bool)
Enable command

mixUnitNum (INPUT USInt)
Mix unit number (1-2)

controls (INOUT *TypePumpControls*)
Controls

target (INOUT *TypeTarget*)
Target to operate on

state (OUTPUT `GracoCore_TypeEnableCmdState`_)
Command state

State codes

- **16#3000** - Command is valid, running
- **16#8000** - Error, device number is invalid
- **16#8002** - Internal error
- **16#8001** - Error, invalid mix unit number
- **16#0000** - Command is disabled

ReadAlarmInfo

Read an alarm record from the PD2K. See *SendDCS* for state codes.

indexNum (INPUT USInt)
Index number (0-199)

execute (INOUT Bool)
Execute command

target (INOUT *TypeTarget*)
Target to operate on

eventCode (OUTPUT STRING[4])
Event code

dateTime (OUTPUT DTL)
Event datetime

state (OUTPUT `GracoCore_TypeExecuteCmdState`_)
Command state

ReadEventInfo

Read an event record from the PD2K. See *SendDCS* for state codes.

indexNum (INPUT USInt)
Index number (0-199)

execute (INOUT Bool)
Execute command

target (INOUT *TypeTarget*)
Target to operate on

eventCode (OUTPUT STRING[4])
Event code

dateTime (OUTPUT DTL)
Event datetime

state (OUTPUT `GracoCore_TypeExecuteCmdState`_)
Command state

ReadFlushSequence

Read parameters for a flush sequence. See *SendDCS* for state codes.

flushSeqNum (INPUT USInt)
Flush sequence number (1-5)

execute (INOUT Bool)
Execute command

target (INOUT *TypeTarget*)
Target to operate on

gunPurgeTime (OUTPUT UInt)
Gun purge time (s)

initialFlushVol (OUTPUT UInt)
Initial flush volume (cc)

finalFlushVol (OUTPUT UInt)
Final flush volume (cc)

numWashCycles (OUTPUT USInt)
Number of wash cycles

strokesPerWashCycle (OUTPUT USInt)
Number of strokes per wash cycle

state (OUTPUT `GracoCore_TypeExecuteCmdState`_)
Command state

ReadGrandTotals

Read parameters for a flush sequence. See *SendDCS* for state codes.

execute (INOUT Bool)
Execute command

target (INOUT *TypeTarget*)
Target to operate on

materialATotal (OUTPUT UInt)
Material A total (gal)

materialBTotal (OUTPUT UInt)
Material B total (gal)

materialABTotal (OUTPUT UInt)
Material A+B total (gal)

solventTotal (OUTPUT USInt)
Solvent total (gal)

state (OUTPUT `GracoCore_TypeExecuteCmdState`_)
Command state

ReadJobInfo

Read a job record from the PD2K. See *SendDCS* for state codes.

indexNum (INPUT USInt)
Index number

execute (INOUT Bool)
Execute command

target (INOUT *TypeTarget*)
Target to operate on

dateTime (OUTPUT DTL)
Recorded datetime

jobNum (OUTPUT UInt)
Job number

mixUnitNum (OUTPUT USInt)
Mix unit number

recipeNum (OUTPUT USInt)
Recipe number

totalVol (OUTPUT UDInt)
Total volume of mixed material (cc)

userID (OUTPUT STRING[9])
User ID

state (OUTPUT `GracoCore_TypeExecuteCmdState`_)
Command state

ReadRecipe

Read parameters for a given recipe number. See *SendDCS* for state codes.

mixUnitNum (INPUT USInt)
Mix unit number (1-2)

recipeNum (INPUT USInt)
Recipe number (0-60)

execute (INOUT Bool)
Execute command

target (INOUT *TypeTarget*)
Target to operate on

matNumA (OUTPUT USInt)
Material number of component A

matNumB (OUTPUT USInt)
Material number of component B

flushSeqNumA (OUTPUT USInt)
Flush sequence number for component A

flushSeqNumB (OUTPUT USInt)
Flush sequence number for component B

mixRatioSP (OUTPUT UInt)
Mix ratio setpoint (A*100 : B)

potLifeTimeSP (OUTPUT UInt)
Potlife time setpoint (min)

mixPressureTolPct (OUTPUT UInt)
Mix pressure tolerance (%)

state (OUTPUT `GracoCore_TypeExecuteCmdState`_)
Command state

ReadUserID

Read the current User ID. See *SendDCS* for state codes.

mixUnitNum (INPUT USInt)
Mix unit number (1-2)

execute (INOUT Bool)
Execute command

target (INOUT *TypeTarget*)
Target to operate on

userID (OUTPUT STRING[9])
User ID

state (OUTPUT `GracoCore_TypeExecuteCmdState`_)
Command state

SendDCS

Send a dynamic command structure (DCS) to the PD2K. See the operation manual for more details regarding the DCS.

cmdID (INPUT USInt)
Command ID

arg0 (INPUT DWord)
Argument 0

arg1 (INPUT DWord)
Argument 1

arg2 (INPUT DWord)
Argument 2

arg3 (INPUT DWord)
Argument 3

arg4 (INPUT DWord)
Argument 4

arg5 (INPUT DWord)
Argument 5

arg6 (INPUT DWord)
Argument 6

arg7 (INPUT DWord)
Argument 7

arg8 (INPUT DWord)
Argument 8

execute (INOUT Bool)
Execute command

target (INOUT *TypeTarget*)
Target to operate on

ret0 (OUTPUT DWord)
Return 0

ret1 (OUTPUT DWord)
Return 1

ret2 (OUTPUT DWord)
Return 2

ret3 (OUTPUT DWord)
Return 3

ret4 (OUTPUT DWord)
Return 4

ret5 (OUTPUT DWord)
Return 5

ret6 (OUTPUT DWord)
Return 6

ret7 (OUTPUT DWord)
Return 7

ret8 (OUTPUT DWord)
Return 8

state (OUTPUT `GracoCore_TypeExecuteCmdState`_)
Command state

State codes

- **16#2000** - Command completed
- **16#4000** - Command was aborted
- **16#8000** - Error, command ID must be non-zero
- **16#8001** - Error, device number is invalid
- **16#8002** - Internal error
- **16#8003** - Error, timed out waiting for DCS registers to be available
- **16#8004** - Error, DCS command failed
- **16#8005** - Error, timed out waiting for acknowledge signal

WriteFlushSequence

Write the parameters for a given flush sequence number. See *SendDCS* for state codes.

flushSeqNum (INPUT USInt)
Flush sequence number (1-5)

gunPurgeTime (INPUT UInt)
Gun purge time (0-999 s)

initFlushVol (INPUT UInt)
Initial flush volume (0-9999 cc)

finalFlushVol (INPUT UInt)
Final flush volume (0-9999 cc)

numWashCycles (INPUT USInt)
Number of wash cycles (0-99)

strokesPerWashCycle (INPUT USInt)
Number of strokes per wash cycle (0-99)

execute (INOUT Bool)
Execute command

target (INOUT *TypeTarget*)
Target to operate on

state (OUTPUT `GracoCore_TypeExecuteCmdState`_)
Command state

WriteMaterialReadyFlag

Write a value for the material ready flag. See *SendDCS* for state codes.

mixUnitNum (INPUT USInt)
Mix unit number (1-2)

materialReadyFlag (INPUT Bool)
Material ready flag

execute (INOUT Bool)
Execute command

target (INOUT *TypeTarget*)
Target to operate on

state (OUTPUT `GracoCore_TypeExecuteCmdState`_)
Command state

WriteRecipe

Write the parameters for a given recipe number. See *SendDCS* for state codes.

mixUnitNum (INPUT USInt)
Mix unit number (1-2)

recipeNum (INPUT USInt)
Recipe number (0-21)

matNumA (INPUT USInt)
Material number for component A (0-30)

matNumB (INPUT USInt)
Material number for component B (0, 31-34)

flushSeqNumA (INPUT USInt)
Flush sequence number for component A (1-5)

flushSeqNumB (INPUT USInt)
Flush sequence number for component B (1-5)

mixRatioSP (INPUT UInt)
Mix ratio setpoint (0-5000, A*100 : B)

potLifeTimeSP (INPUT UInt)
Potlife time setpoint (0-999 min)

mixPressureTolPct (INPUT UInt)
Mix pressure tolerance (%)

execute (INOUT Bool)
Execute command

target (INOUT *TypeTarget*)
Target to operate on

state (OUTPUT `GracoCore_TypeExecuteCmdState`_)
Command state

WriteUserID

Write a value for the user ID. See *SendDCS* for state codes.

mixUnitNum (INPUT USInt)
Mix unit number (1-2)

userID (INPUT STRING[9])
User ID

execute (INOUT Bool)
Execute command

target (INOUT *TypeTarget*)
Target to operate on

state (OUTPUT `GracoCore_TypeExecuteCmdState`_)
Command state

4.1.3 UDTs

TypeMixUnit

eventFlag (Bool)

safetyInterlockStatus (Bool)

inStandby (Bool)

systemMode (USInt)
System mode

systemModeFlags (*TypeSystemModeFlags*)
System status flags

actualMixFlow (UInt)
Actual mix flow/pressure (cc/min in flow mode, psi in pressure mode)

actualMixRatio (UInt)
Actual mix ratio (A*100 : B)

actualPotLifeRemaining (UInt)
Actual pot life time remaining (sec)

activeRecipeNum (USInt)
Active recipe number

activeMatNumA (USInt)
Active material number for component A

activeMatNumB (USInt)
Active material number for component B

activeFlushSeqNumA (USInt)
Active flush sequence number for component A

activeFlushSeqNumB (USInt)
Active flush sequence number for component B

activeRatioSP (UInt)
Active mix ratio setpoint (A*100 : B)

activePotLifeSP (UInt)
Active pot life setpoint (min)

currentJobNum (UInt)

currentJobSprayVolA (UInt)

currentJobSprayVolB (UInt)

currentJobSolventVol (UInt)

TypePd2kStatus

PD2K Integrated status data. See *StatusToUDT*.

mixUnit1 (*TypeMixUnit*)

mixUnit2 (*TypeMixUnit*)

gun1Trigger (Bool)
Gun 1 trigger status

gun2Trigger (Bool)
Gun 2 trigger status

pump1 (*TypePump*)
Pump 1 status

pump2 (*TypePump*)
Pump 2 status

pump3 (*TypePump*)
Pump 3 status

pump4 (*TypePump*)
Pump 4 status

dcsAck (USInt)
DCS acknowledge

dcsAckFlags (`GracoCore_TypeDcsAckFlags`_)
DCS acknowledge flags

dcsRet0 (DWord)
DCS return 0

dcsRet1 (DWord)
DCS return 1

dcsRet2 (DWord)
DCS return 2

dcsRet3 (DWord)
DCS return 3

dcsRet4 (DWord)
DCS return 4

dcsRet5 (DWord)
DCS return 5

dcsRet6 (DWord)
DCS return 6

dcsRet7 (DWord)
DCS return 7

dcsRet8 (DWord)
DCS return 8

TypePump

status (USInt)
Pump status

statusFlags (*TypePumpStatusFlags*)
Pump status flags

materialNum (USInt)

actualFlowRate (UInt)
Actual pump flow rate (cc/min)

actualPressure (UInt)
Actual pump pressure (psi)

TypePumpControls

Pump control inputs. See *PumpControl*.

powerOnCmd (Bool)
Power on command

powerOffCmd (Bool)
Power off command

quickStopCmd (Bool)
Quick stop command

recipeChangeCmd (Bool)
Recipe change command

clearAlarm (Bool)
Clear alarm command

completeJob (Bool)
Complete job command

mixCmd (Bool)
Mix mode command

mixFillCmd (Bool)
Mix fill mode command

recipePurgeCmd (Bool)
Recipe purge command

standbyCmd (Bool)
Standby mode command

mixCtrlModeSP (Bool)
Mix control mode setpoint (0=flow mode, 1=pressure mode)

mixCtrlSP (UInt)
Mix control setpoint (cc/min for flow mode, psi for pressure mode)

mixFillCtrlSP (UInt)
Mix fill control setpoint (cc/min for flow mode, psi for pressure mode)

nextRecipeNum (UInt)
Next recipe number

TypePumpStatusFlags

Pump status flags

off (Bool)
Pump is off

standby (Bool)
Pump is in standby

busy (Bool)
Pump is busy

flushing (Bool)
Pump is flushing

priming (Bool)
Pump is priming

TypeSystemModeFlags

System mode flags

pumpOff (Bool)
System is off

colorChange (Bool)
Color change is active

colorChangePurgeA (Bool)
Color change is active, purging component A

colorChangePurgeB (Bool)
Color change is active, purging component B

colorChangeFilling (Bool)
Color change is active, filling

mixFill (Bool)
System in mix fill mode

mix (Bool)
System in mix mode

mixIdle (Bool)
System in mix idle mode

purgeA (Bool)
Purging component A

purgeB (Bool)
Purging component B

standbyMixReady (Bool)
Standby, mix ready

standbyFillReady (Bool)
Standby, ready for fill

standbyMixNotReady (Bool)
Standby, mix is not ready

standbyAlarm (Bool)
Standby, alarm is active

lineFillingFlushing (Bool)
Line is filling/flushing

pumpPrimeFlush (Bool)
Priming/flushing a pump

maintenance (Bool)
Maintenance

solventPush (Bool)
Solvent push

TypeTarget

Defines a target Profinet device. See *Using TypeTarget*.

simulate (Bool)
Simulation mode select (1=on)

pnDeviceNum1 (UInt)
Profinet device number for CGM 1

pnDeviceNum2 (UInt)
Profinet device number for CGM 2

simData1 (Array[1..56] of DWord)
Simulation data for CGM 1

simData2 (Array[1..56] of DWord)
Simulation data for CGM 2

CHAPTER

FIVE

CHANGELOG

5.1 Release version 0.2.1 (latest)

Release date: 2023-02-13

Changed:

- Profinet data access has been largely reworked throughout the code. Instead of relying on UDTs to move data around, we now use GetDataBySlotNum/SetDataBySlotNum along with TypeTarget to access the process data directly. This replaces the original pack/unpack FCs. See the "design" section in the manual for more details.
- All code now supports the S7-1200 series of PLCs. This required removing all STL code from the program, since STL is not supported in S7-1200.
- Migrated all code from TIA Portal V12 to V17.
- TypePd2kStatus has been restructured to group data for each mix unit together.
- TypePd2kStatus now uses more appropriate numeric data types for members (was DWORD for everything)
- TypePd2kStatus now includes additional flags for enumerated values. StatusToUDT is used to populate these values.
- Reworked all state codes to be more consistent with other SDKs. See "design" section in manual for more details.
- Much of the internal logic has been reworked and improved for better readability.
- Manual has been completely re-written. Content has been updated to reflect latest SDK improvements. Formatting has also been improved.
- The PD2K GSDML file has been updated to schema V2.2.

Added:

- PumpControl FB
- StatusToUDT FC - similar to original UnpackStatus FC but outputs to a TypePd2kStatus UDT instead of the full TypePD2K.
- Types EnableCmdState, ExecuteCmdState, and FunctionCmdState - replaces TypeCmdState
- TypeDcsAckFlags
- TypePumpControls - used with PumpControl FB
- Various FCs for working with strings and datetimes - DecodeDateTime, DecodeEventCode, DecodeUserID, EncodeUserID.
- GetDataBySlotNum and SetDataBySlotNum.

- In addition to the PDF manual, an interactive manual has also been included which can be viewed in a web browser.

Removed:

- Removing support for S7-300. Managing both versions of code required a lot of extra development time, and since the vast majority of customers are using S7-1200/1500 PLCs in their installs, the S7-300 compatibility was not adding much value.
- RecipeChange FB - merged into PumpControl
- Manual is no longer provided in .docx format.

5.2 Release version 0.1.1

Release date: 2020-10-23

This is the first release of the SDK.

- Initial release for GracoPd2kDual global library
- Initial release for GracoPd2kDual example program
- Initial release for GracoPd2kDual manual
- Initial release for PackControl FC
- Initial release for PackControlAdvanced FC
- Initial release for UnpackControl FC
- Initial release for UnpackControlAdvanced FC
- Initial release for UnpackStatus FC
- Initial release for UnpackStatusAdvanced FC
- Initial release for ReadAlarmInfo FB
- Initial release for ReadEventInfo FB
- Initial release for ReadFlushSeq FB
- Initial release for ReadGrandTotals FB
- Initial release for ReadJobInfo FB
- Initial release for ReadRecipe FB
- Initial release for ReadUserID FB
- Initial release for RecipeChange FB
- Initial release for SendDCS FB
- Initial release for WriteFlushSeq FB
- Initial release for WriteMaterialReadyFlag FB
- Initial release for WriteRecipe FB
- Initial release for WriteUserID FB