Dokumentation
der TwinCAT Library
„EfficientIO Communicator“

Inhalt

Change Log 3
Systemvoraussetzungen 4
Einführung 4
Erstellen des Accounts und herunterladen der Bibliothek 5
Installation der Bibliothek 6
Verwendung 7
Begrenzungen Symbolname und Kommentar 7
TLS Zertifikat 8
TMC File 8
Funktionsblöcke 9
FB_VarInfo 9
FB_ReadWrite 11
FB_MqttCommunicator 12
Verwendete Datentypen 13
Unterstützte Datentypen 16
Enumeration 17
Struktur 18
Funktionsblöcke 19
Beispiele 20
Licht ein und ausschalten 20
Dimmbares Licht 21
Temperatursensor 22
Statusüberwachung von 2 Förderbandanlagen 23

Change Log

Version	Datum	Änderungen
1.0	29.06.2022	Initialversion EfficientIO

Systemvoraussetzungen

Die EfficientIO Bibliothek funktioniert auf allen Beckhoff Steuerungen welche TwinCat 3.1 4022.0 oder höher installiert haben. Zusätzlich zur EfficientIO Bibliothek, braucht es die TwinCat 3 Funktion TF6701 der Firma Beckhoff, um den Service nutzen zu können. Die Zykluszeit in welcher die Funktionsblöcke laufen, sollte je nach Performance Level der SPS passend ausgewählt werden. (Siehe Infos direkt bei den Funktionsblöcken)

Die Zykluszeit sollte allerdings nicht kleiner als 1ms gewählt werden. Es wird empfohlen die Funktionsblöcke in unterschiedlichen Task Referenzen mit unterschiedlichen Zykluszeiten laufen zu lassen.

Diese Bibliothek verlangt die Installation der EfficientIO Standard Bibliothek.

Für die Funktion benötigt die Beckhoff SPS einen Zugang zum Internet. Bei CE Geräten wird die Vorschaltung einer Hardwarefirewall empfohlen, generell wird dies jedoch bei jedem Gerät empfohlen welches am Internet angeschlossen wird.

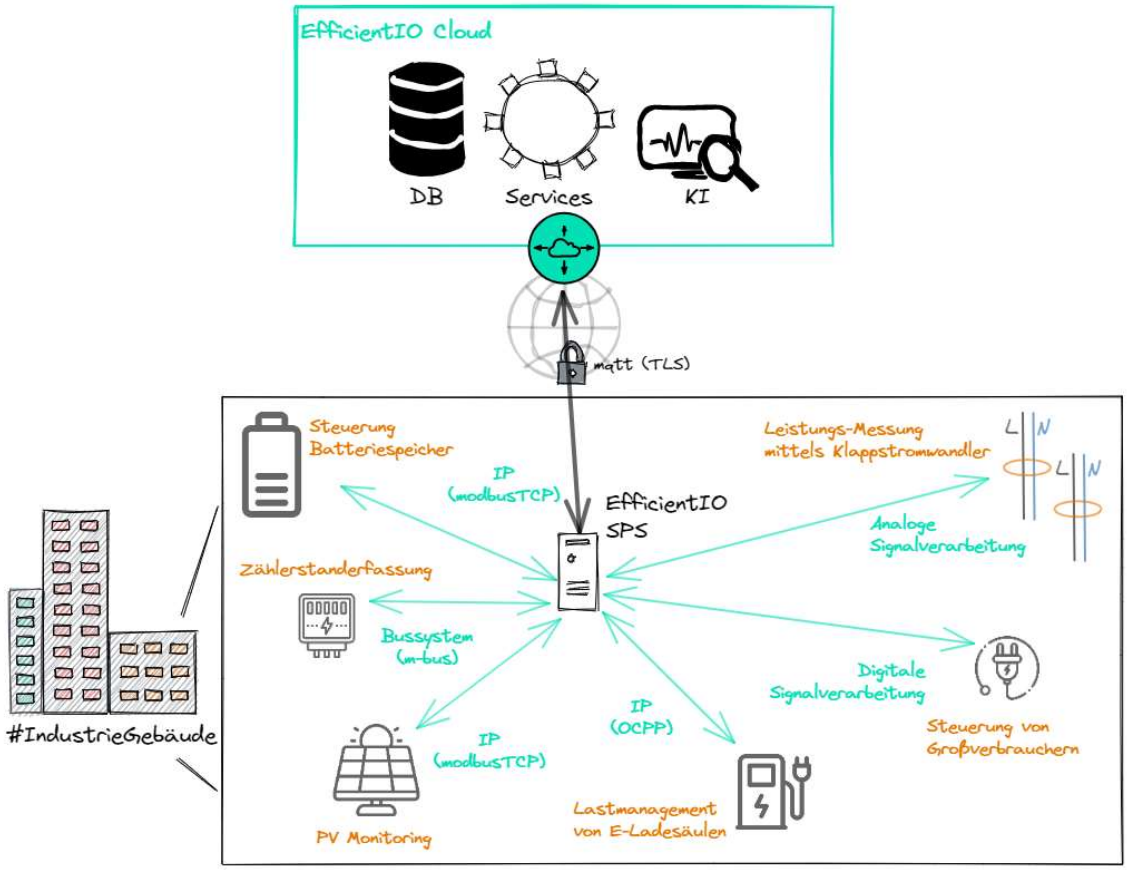
Einführung

Der EfficientIO Communicator wurde entwickelt, um Daten bidirektional über MQTT von einer Beckhoff SPS auf einfache und performante Weise in die EfficientIO Cloud zu bringen. Der SPS Programmierer muss sich um keine Cloud Angelegenheiten kümmern und braucht daher auch keine speziellen Kenntnisse dafür. Er kann sein SPS Programm weiterhin so frei entwickeln und programmieren wie er es gewohnt ist. Der Programmierer muss sich codetechnisch auch keine Gedanken auf der SPS machen, wie die Daten von den Variablen in die Cloud und wieder zurückkommen. Zur Benutzung der Bibliothek muss lediglich ein Account im Dashboard erstellt und anschließend die EfficientIO Bibliothek auf der Beckhoff Steuerung installiert werden.

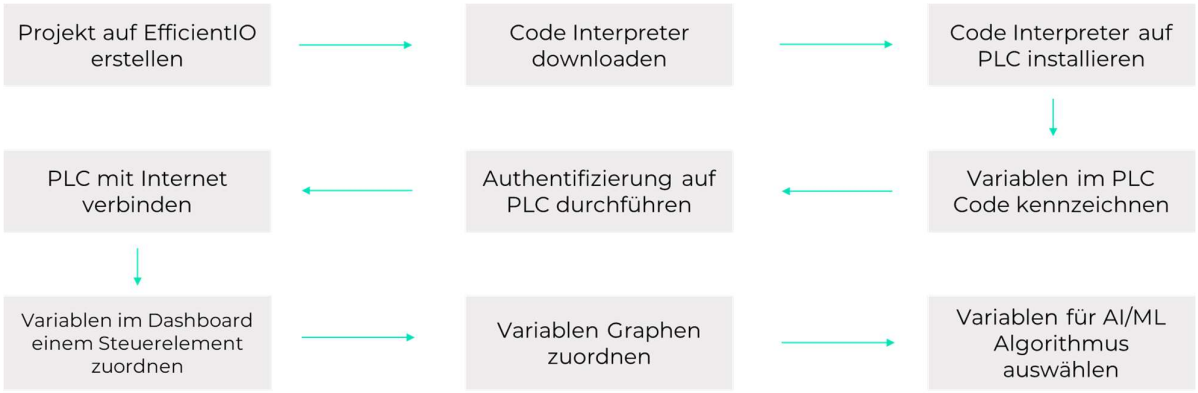
Beim Erstellen des Accounts bekommt der Benutzer die Zugangsdaten für die Cloud, welche im Kommunikations-Funktionsbausteinen angegeben werden müssen. Sobald die Daten der SPS in der Cloud sind, können von der klassischen Gebäudeautomatisierung, über Industrieanwendungen bis hin zu ML verschiedenste Projekte umgesetzt werden.

Durch diese Bibliothek und den anschließenden Services in der Cloud bietet man jedem SPS Programmierer eine einfache Möglichkeit, Daten von der SPS in die Cloud zu bringen, dort über verschiedenste Services wie Visualisierungen, Apps, Alarmierungen, Datenbanken, ML und vieles mehr die Daten zu verarbeiten und anschließend bei Bedarf wieder zurück auf die Steuerung zu schicken.

Funktionsprinzip

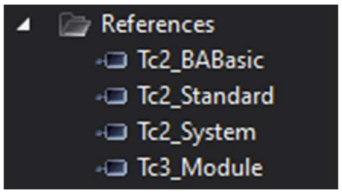


Erstellen des Accounts und herunterladen der Bibliothek

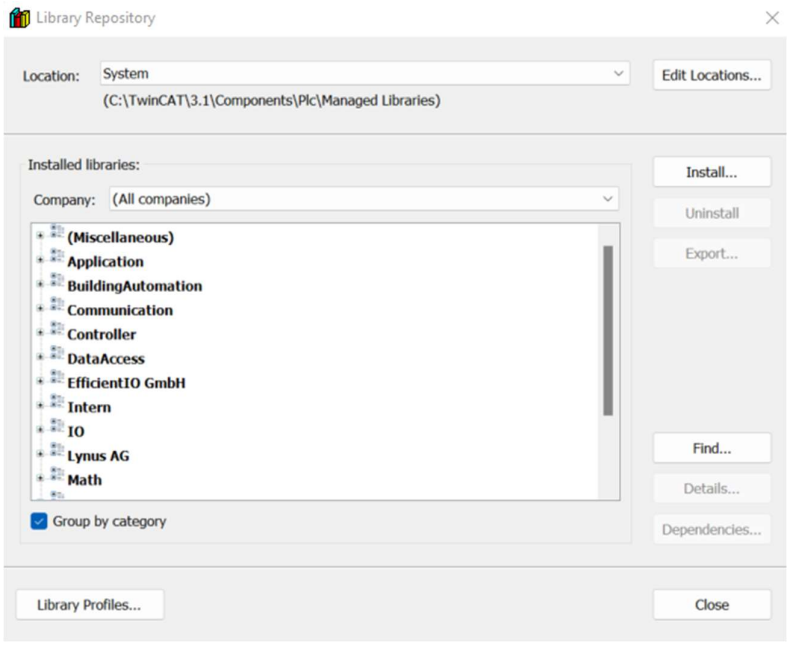


Installation der Bibliothek

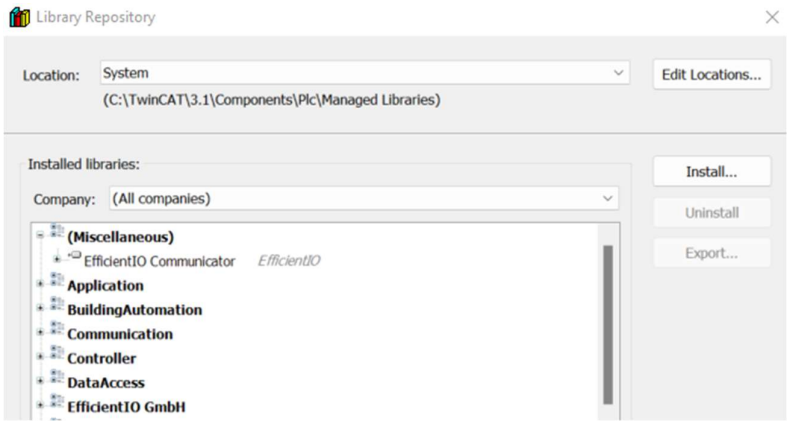
Nachdem die Bibliothek über den erstellten Account heruntergeladen wurde, im SPS Projekt rechtsklick auf References und dann klick auf Bibliotheksrepository:



Danach auf installieren klicken:



Die EfficientIO Bibliothek im abgespeicherten Pfad auswählen und dann auf Öffnen klicken. Nach erfolgreicher Installation erscheint die Bibliothek im Ordner „(Sonstige)“ oder „EfficientIO GmbH“.



Verwendung

Um eine Variable von der SPS in die Cloud schicken zu können, reicht es aus, wenn der SPS Programmierer die Variablen mit einem definierten Kommentar in seinem Code markiert.
Der Kommentar bzw. die Markierung muss wie folgt lauten:

```
//{#lynus.ag#()}
```

Das Kommentar muss zwingend so beginnen und darf auch nicht verändert werden. Ansonsten erkennt der Baustein die Variable nicht. Möchte man hinter der Markierung noch ein Kommentar ergänzen, muss dies zwingend mit einem Leerzeichen beginnen und dann müssen noch 2 „/“ folgen.
z.B. `//{#lynus.ag#()} //Hier kann das Kommentar stehen`

Sollte hinter der Markierung kein Leerzeichen// stehen, gibt der Baustein einen Fehler aus. Ebenfalls Verboten sind im Kommentar folgende 2 Sonderzeichen Zeichen : < und >

```
13      bvar        : BOOL;                     //{#lynus.ag#()}
14      fbGrid      : FB_Grid_Power;            //{#lynus.ag#()}
15      fbGenerator : FB_Generator_OnOff;       //{#lynus.ag#()}
16      fbems       : E_CountrySelection_EMS;   //{#lynus.ag#()}
17      fbWoche     : FB_Montag;                //{#lynus.ag#()} //Das ist ein FB
18  END_VAR
19
```

Alle Funktionsblöcke der Bibliothek müssen im selben Laufzeitsystem aufgerufen werden. Die Variablen, welche kommentiert werden für den Cloudservice, müssen ebenfalls im selben Laufzeitsystem sein.
Das Kommentar kann bei Variablen in Programmaufrufen hinzugefügt werden, aber auch bei Variablen in Globalen Variablenlisten. Zusätzlich kann das Kommentar auch direkt bei Variablen in Funktionsblöcken, Strukturen und Enumerations verwendet werden. Siehe „Unterstützte Datentypen“.

Begrenzungen Symbolname und Kommentar

Beim eindeutigen Symbolnamen, also wie die Variablen in der SPS genannt werden und beim Kommentar, gibt es Begrenzungen in der Länge, die für einen fehlerfreien Betrieb einzuhalten sind.

- Die Begrenzung für den Symbolnamen bezieht sich auf maximal 48 Zeichen.
- Die Begrenzung für ein Symbol (Funktionsblock, Struktur oder Enumeration) bezieht sich auf maximal 48 Zeichen.
- Die Begrenzung für den Text in einem Datentyp ENUM bezieht sich auf maximal 48 Zeichen.
- Das Kommentar darf eine Gesamtlange von 255 Zeichen nicht überschreiten.

ACHTUNG - Beim Symbolnamen gilt folgendes zu beachten:
Wird das Kommentar direkt in Funktionsblöcken, Strukturen oder Enumerations verwendet, dann ergibt sich die Länge des Symbolnames aus dem Programmaufruf inkl. dem Namen der Funktion und dem Variablennamen.
So kann ein Symbolname z.B. lauten: `Main.fbTest.bVar1`

➔ Ergibt eine Länge von 17 Zeichen

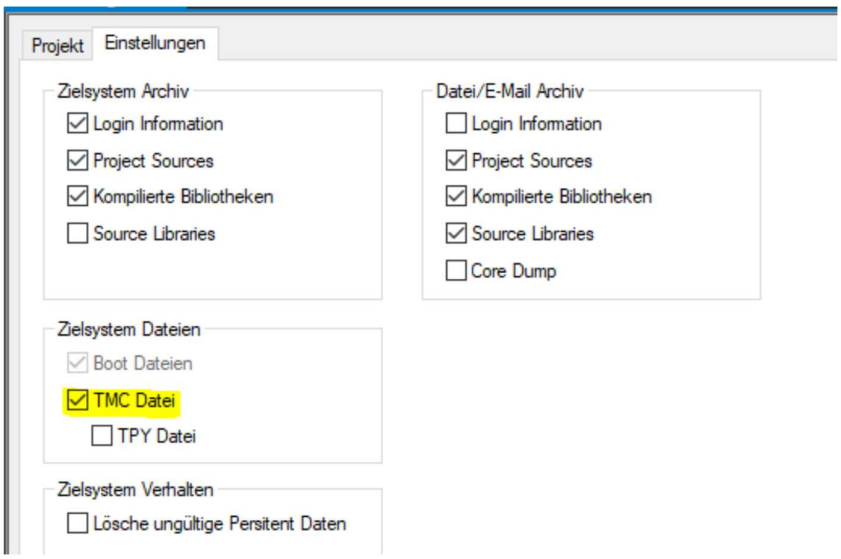
TLS Zertifikat

Das TLS Zertifikat für die Kommunikation in die Cloud erhält man beim Erstellen des Projektes in der Cloud. Dieses Zertifikat kann heruntergeladen werden und muss dann auf das Lokale Laufwerk der SPS kopiert werden. Falls der Ordner „Certificates“ nicht existiert, muss dieser erstellt werden. Es handelt sich dabei um das „MQTT Broker Certificate“ (mqtt.pem).

Der Pfad auf der SPS, wo das Zertifikat gespeichert werden muss, lautet wie folgt:
Windows Embedded Compact => Hard Disk\TwinCAT\3.1\Config\Certificates\
Windows 7 oder Windows 10 => C:\TwinCAT\3.1\Config\Certificates\

TMC File

Ab der Version 2 können Variablen auch direkt in Funktionsblöcken, Strukturen oder Enumerations markiert werden. Dafür muss immer das aktuellste TMC File des SPS Projektes lokal auf die SPS übertragen werden. Werden Änderungen im SPS Code bei Funktionsblöcken, Strukturen oder Enumerations gemacht, die mit dem EfficientIO Communicator verwendet werden sollen, muss das aktuelle TMC File wieder auf die SPS übertragen werden. Das TMC File befindet sich immer direkt im Ordner vom jeweiligen SPS Projekt und kann von dort kopiert werden. Es sollte sich immer nur 1 TMC File im Ordner auf der SPS befinden. Damit das TMC File erzeugt wird, muss in den Einstellungen vom SPS Projekt folgendes Häkchen gesetzt werden.



Sollten keine Kommentare in Funktionsblöcken, Strukturen oder Enumerations gemacht worden sein, braucht es kein TMC File.

Sollte der Ordner „TMC“ noch nicht existieren auf der SPS, muss dieser erstellt werden.
Der Pfad auf der SPS, wo das TMC File gespeichert werden muss, lautet wie folgt:
Windows Embedded Compact => Hard Disk\TwinCAT\3.1\Config\TMC\
Windows 7 oder Windows 10 => C:\TwinCAT\3.1\Config\TMC\

Funktionsblöcke

In der EfficientIO Bibliothek findet man grundsätzlich nur 3 Funktionsblöcke:

- Ein Funktionsblock dient dem Sammeln der Informationen der markierten, bzw. kommentierten Variablen.
- Ein weiterer Funktionsblock kümmert sich um das Lesen bzw. Schreiben der kommentierten Variablen.
- Der letzte Baustein dient der Kommunikation in die EfficientIO Cloud.

Jeder Baustein, bis auf den ‚FB_ReadWrite', darf pro SPS nur einmal vorkommen. Sollten mehrere Instanzen der Funktionsblöcke auf einer SPS vorkommen, kommt es zu einem Fehler am Funktionsbaustein, und es wird ein entsprechender Fehlercode ausgegeben. Der ‚FB ReadWrite' darf maximal 10-mal im SPS Code vorkommen. Generell gilt, je schneller die Task Zeit, umso schneller auch der Ablauf der Applikationen.

FB_VarInfo

Der ‚FB_VarInfo' sammelt die Informationen über alle kommentierten Variablen in der SPS. Der Baustein wird nach dem Start der Laufzeit aufgerufen und gestartet. Alle anderen FB's der Bibliothek starten erst nach dem erfolgreichen Durchlauf dieses Funktionsblockes. Nach jedem Neustart oder einem Online Change auf der SPS beginnt die Suche von neuem, um zu prüfen, ob sich an den Variablen etwas geändert hat. Dieser Funktionsblock sollte in einer Zykluszeit <= 10ms laufen.

Eingänge

Name	Typ	Beschreibung
bReadInfo	BOOL	Über diesen Eingang startet die Auslesung der Variablen von manuellem neu. Wird im Normalfall aber nicht benötigt, da die Auslesung bei Änderungen und Neustart der SPS selbstständig neu startet. Bei Bestätigung stoppt das lesen und schreiben der Variablen und die Kommunikation in die Cloud.
eTMCFile	ESELECTTMCFILE	Mit diesem ENUM kann ausgewählt werden ob ein TMC File verwendet werden soll oder nicht. Das TMC File muss dafür vom Projektordner auf die SPS kopiert werden. Wird nur benötigt, wenn Markierungen direkt in Funktionsblöcken, Strukturen oder Enumerations gemacht werden. Ansonsten wird dies nicht benötigt.
eSavePositionTMCFile	ESAVEPOSITIONFILE	Hier muss ausgewählt werden, wo das TMC File auf der SPS abgespeichert ist. Hard Disk\ (Bei CE Geräten) oder C:\ (Bei Win 7 oder Win 10 Geräten)

Ausgänge

Name	Typ	Beschreibung
udiCounterFoundVariables	UDINT	Gibt die Anzahl der gefundenen kommentierten Symbole wieder. Dies sollten am Ende des Scannens mit der Anzahl der kommentierten Symbole im SPS Projekt übereinstimmen.
bError	BOOL	Wird True, wenn während dem Ausführen des Bausteines ein Fehler aufgetreten ist.
bBusy	BOOL	True solange der Baustein aktiv ist.
byProgress	BYTE	Gibt den Fortschritt des Bausteines an in 0%-100%
udiErrorCodeOut	UDINT	Gibt den internen Beckhoff Fehler wieder
eErrorVarInfo_Out	EERRORVARINFO	Gibt den spezifischen Fehler des Funktionsblockes wieder.
sVarWithError	STRING(255)	Wenn es sich bei dem Fehler um einen Fehler mit einem Symbol handelt, wird hier das jeweilige Symbol ausgegeben.

FB_ReadWrite

Der ‚FB_ReadWrite' dient nach dem Scannen der Symbole dem Lesen und Schreiben der Werte.
Die gelesenen Werte werden in Richtung Cloud geschickt und die schreibenden, kommend von der Cloud, werden auf den Wert der SPS geschrieben. Das Schicken der Daten in beide Richtungen ist Event basiert, sprich Daten werden nur verschickt, wenn sich der Wert ändert. Der Code wurde intern so angepasst, dass ‚FB_ReadWrite' je nach Anwendungsfall skaliert werden kann. Das heißt, es können bis zu 10 Instanzen dieses FB's auf der Steuerung vorkommen, um das Lesen und Schreiben der Werte zu beschleunigen. Dies macht dann Sinn, wenn viele Werte in der SPS vorkommen die in die Cloud geschickt werden müssen. Durch diese Skalierbarkeit hat der Programmierer die Möglichkeit die Applikation immer auf seinen Anwendungsfall passend zu integrieren und die Ressourcen der SPS optimal auszunutzen. Dieser Funktionsblock sollte in einer Zykluszeit <= 5ms laufen. Bei Performance Level 20 oder 30 der SPS sollten je nach freien Performance Ressourcen maximal 5 dieser Funktionsblöcke pro SPS Projekt verwendet werden.

Wichtig: Der ‚FB_ReadWrite' beginnt erst zu arbeiten, nachdem der ‚FB_VarInfo' erfolgreich abgearbeitet wurde und der ‚FB_MqttCommunicator' eine erfolgreiche Verbindung zur Cloud herstellen konnte.

Ausgänge

Name	Typ	Beschreibung
bBusy	BOOL	Wird True sobald der Baustein beginnt zu arbeiten.
bError	BOOL	Wird True, wenn ein Fehler auftritt
udiErrorCodeOut	UDINT	Gibt den internen Beckhoff Fehler wieder
eErrorReadWrite_Out	EERRORREADWRITE	Gibt den spezifischen Fehler des Funktionsblockes wieder

FB_MqttCommunicator

Der ‚FB_MqttCommunicator' dient der bidirektionalen Kommunikation zwischen der Cloud und der SPS. Auf diesem Wege findet der Datenaustausch statt. Dieser Funktionsblock sollte in einer Zykluszeit <= 20ms laufen.

Wichtig: Der ‚FB_MqttCommunicator' startet erst, sobald der ‚FB_VarInfo' erfolgreich abgearbeitet wurde.

Eingänge

Name	Typ	Beschreibung
bConnect	BOOL	Wenn True dann verbindet sich der Baustein mit der Cloud.
sUserName	STRING(36)	Benutzername. Dieser kann nach Erstellen eines Accounts in der Cloud heruntergeladen werden und muss hier angegeben werden.
sUserPassword	STRING(36)	Benutzer Passwort. Dieses kann nach Erstellen eines Accounts in der Cloud heruntergeladen werden und muss hier angegeben werden.
eSavePositionCertificate	ESAVEPOSITIONFILE	Hier muss ausgewählt werden, wo das TLS Zertifikat auf der SPS abgespeichert ist. Hard Disk\ (Bei CE Geräten) oder C:\ (Bei Win 7 oder Win 10 Geräten)

Ausgänge

Name	Typ	Beschreibung
bConnected	BOOL	Wird True, sobald die Kommunikation erfolgreich zur Cloud hergestellt werden konnte.
bError	BOOL	Wird True, wenn ein Fehler auftritt
hrErrorCode	HRESULT	Gibt den internen Beckhoff Code wieder, wenn bError True ist
eConnectionState	ETCIOTMQTTCLIENTSTATE	Gibt den Status der Mqtt Verbindung wieder
eErrorMessage_Out	EERRORMESSAGE	Gibt den spezifischen Fehler des Funktionsblockes wieder

Verwendete Datentypen

Anbei findet man die Beschreibung der verwendeten Datentypen der Bibliothek.

eErrorVarInfo	
Name	**Beschreibung**
eNoError	Kein Fehler
eArraySizeError	Symboldaten zu groß oder zu viele Symbole markiert/kommentiert
eTimoutError	Interner timeout Fehler
eCommentLenghtError	Inhalt des Kommentares ist zu lang
eCommentDescriptionError	Fehler in der Beschreibung des Kommentares (Länge, falsche Zeichen oder überschüssige Leerzeichen)
eSymbolTypeError	Gefundener Datentyp wird nicht unterstützt
eSymbolNameToLongError	Der Symbol Name ist zu lange
eReadWriteFunctionToMuch	Mehr wie 10 Lese/Schreibe Funktion gefunden auf dem Laufzeitsystem
eVarInfoFunctionToMuch	Mehr wie eine Var Info Funktion gefunden auf dem Laufzeitsystem
eSendReceiveFunctionToMuch	Mehr wie ein Sende/Empfang Funktion gefunden auf dem Laufzeitsystem
eMoreReadWriteFuctionsAsSymbols	Es wurden mehr Lese und Schreibe Funktionen gefunden als es Symbole gibt
eNoTMCFileSelected	Kein TMC File ausgewählt, es wurden aber Funktionsblöcke, Strukturen oder Enumerations markiert
eArrayWithNoStandartDatatype	Array markiert welches nicht den Standartdatentypen entspricht
eSymbolNotFoundInTMCFile	Markiertes Symbol wurde im TMC File nicht gefunden. Keine Markierung im Funktionsblock, Struktur oder Enumeration oder kein aktuelles TMC File vorhanden
eSymbolNameWithSubItemIsToLong	Der Symbolname ist zu lang mit allen zusätzlichen Informationen. Siehe „Begrenzungen Symbolname und Kommentar“
eToMuchSubItemsInSymbol	Mehr als 50 Einträge pro Funktionsblock oder Struktur
eToMuchSymbols	Gesamthaft zu viele markierte Symbole mit den Inhalten der Funktionsblöcke oder Strukturen

eSubItemsInfoError	Fehler beim Ermitteln der Daten einer Variable im Funktionsblock oder Struktur
eNoSendReceiveFunction	Es wurde keine Send und Empfangsfunktion FB_Mqtt_Communicator gefunden.
eNoReadWriteFunction	Es wurde keine Lese und Schreibe Funktion FB_ReadWrite gefunden.
eNoSymbolsFound	Es wurden keine markierten Symbole gefunden.
eTMCFileTimoutError	Timout Fehler bei der TMC Funktion
eTMCFileContentOfSymbolIsToBig	Inhalt eines Symbols im TMC File ist zu Groß
eTMCFileSymbolNameToLongError	Name eines Symbols(Funktionsblockes, Struktur oder Enumeration) im TMC File ist zu lange. Siehe „Begrenzungen Symbolname und Kommentar“
eTMCFileToMuchSubItemsInOneSymbol	Zu viele Symbol Einträge in einem Funktionsblock oder Struktur
eTMCFileSubItemNameToLongError	Symbolname in einem Funktionsblock, Struktur oder Enumeration ist zu lange. Siehe „Begrenzungen Symbolname und Kommentar“
eTMCFileErrorWhenOpeningTheFile	Fehler beim Öffnen des TMC Files. TMC File nicht vorhanden oder falscher Namen.
eTMCFileErrorWhenReadingTheFile	Fehler beim lesen des TMC Files.
eTMCFileErrorWhenClosingTheFile	Fehler beim Schließen des TMC Files.
eTMCFileToMuchNoStandartDatatypes	Zu viele markiert Funktionsblöcke, Strukturen oder Enumerations
eTMCFileEnumNameToLongError	Text im Datentyp Enum ist zu lange. Siehe „Begrenzungen Symbolname und Kommentar“
eTMCFileEnumIndexToMuchSizes	Index eines Enums ist zu Groß. Datentyp INT
eTMCFileArraySizeError	Die Größe eines markierten Arrays wurde überschritten
eTMCFileStringSizeError	Die Größe eines markierten Strings wurde überschritten
eTMCFileCheckPositionError	Fehler beim Ermitteln der Größe des TMC Files
eTMCFileSearchPositionError	Fehler bei der Positionsfindung im TMC File
eTMCFileMissingSubItems	Fehlende Unterstruktur die in einem Funktionsblock oder Struktur vorkommt
eTMCFileFunctionblockAccessNotAllowed	Funktionsblock als Unterstruktur in anderen Funktionsblöcken oder Strukturen nicht erlaubt
eTMCFileToMuchSubItemsInOtherSubItems	Zu viele Unterstrukturen in einem Funktionsblock oder Struktur gefunden
eTMCFileArrayWithNoStandartDatatype	Array gefunden das keinen Standard Datentyp enthält

eErrorReadWrite	
Name	**Beschreibung**
eNoError	Kein Fehler
eErrorRead	Ein Fehler beim Lesen der Daten ist aufgetreten
eErrorWrite	Ein Fehler beim Schreiben der Daten ist aufgetreten
eErrorSizeFloatValueRead	Fehler beim Lesen einer Float Zahl aus der SPS
eErrorSizeValueWrite	Fehler beim Schreiben eines empfangenen Wertes

eErrorMessage	
Name	**Beschreibung**
eNoError	Kein Fehler
eReceivedMessageWithError	Empfangene Nachricht hat einen Fehler
eReceivedMessageIncompatibility	Empfangene Nachricht stimmt nicht mit vorhandenen Daten auf der SPS überein. Das Symbol wurden nicht gefunden.
eCreatedMessageWithError	Nachricht konnte auf Grund eines Fehlers nicht erzeugt werden

eSavePositionCert	
Name	**Beschreibung**
ePostionHardDisk	Das TLS Zertifikat liegt auf dem Lokalen Datenträger Hard Disk (CE Geräten)
ePostionC	Das TLS Zertifikat liegt auf dem Lokalen Datenträger Hard Disk (Win 7 oder Win 10)

eSavePositionOnPLC	
Name	**Beschreibung**
eDontUseTMCFile	Das TMC File wird nicht verwendet. Es können nur Standard Datentypen markiert werden.
eUseTMCFile	Das TMC File soll verwendet werden. Es können Standard Datentypen, Funktionsblöcke, Strukturen und Enumerations markiert werden.

Unterstützte Datentypen

Der EfficientIO Communicator unterstützt in der aktuellen Version folgende Datentypen und Grenzen:

BOOL	Wertebereich (0 - 1)
BYTE	Wertebereich (0 - 255)
WORD	Wertebereich (0 - 65535)
DWORD	Wertebereich (0 - 4294967295)
SINT	Wertebereich (-128 - 127)
USINT	Wertebereich (0 - 255)
INT	Wertebereich (-32768 - 32767)
UINT	Wertebereich (0 - 65535)
DINT	Wertebereich (-2147483648 - 2147483647)
UDINT	Wertebereich (0 - 4294967295)
LINT	Wertebereich (-9223372036854775808 - 9223372036854775807)
ULINT	Wertebereich (0 - 18446744073709551615)
LWORD	Wertebereich (0 - 18446744073709551615)
REAL	Wertebereich (≈-3.40282326E+38 - ≈3.40282326E+38)
LREAL	Wertebereich (≈-1.7976931348623157E+307 - ≈ 1.7976931348623157E+307)
STRING(30)	Wertebereich (0 - 30 Zeichen)
TIME	Wertebereich (T#0S - T#71582m47s295ms)
TIME_OF_DAY	Wertebereich (TOD#00:00 - TOD#1193:02:47.295)
DATE	Wertebereich (D#1970-01-01- D#2106-02-06)
DATE_AND_TIME	Wertebereich (DT#1970-01-01-00:00- DT#2106-02-06-06:28:15)

Der EfficientIO Communicator unterstützt generell auch die oben genannten Datentypen in Ein-Dimensionalen Arrays mit einer Größe bis maximal 200 Einträgen. Allerdings ist diese Funktion standardmäßig begrenzt auf ein Array mit nur 1 Eintrag. Sollte Sie mehr benötigen, nehmen Sie direkt mit uns Kontakt auf.

Wichtig: Ab Version 2 können Markierungen auch in Funktionsblöcken, Strukturen und Enumerations vorgenommen werden. Die Verwendung von zwei oder drei dimensionalen Arrays ist momentan nicht möglich. Ebenfalls nicht unterstützt werden Arrays die nicht den oben aufgeführten Datentypen entsprechen.

Achtung: Die Zeichenfolge #EMPTY# darf beim Datentype String(30) nicht verwendet werden.

Achtung: Bei den Datentypen REAL und LREAL handelt es sich bei den Ober- und Untergrenzen um ca. Angaben. Fließkommazahlen können nicht immer exakt auf dieselbe Kommastelle dargestellt werden durch Rundungsfehler, Konvertierungen usw.

Enumeration

Ab Version 2 können unter Verwendung des TMC Files auch Datentypen vom Typ ENUM markiert werden. Das ENUM wird intern als INT Variable gehandhabt. Der Wert eines ENUM ist im Cloudservice somit auch nur als Zahl sichtbar. Die Zahl entspricht dem Wert, wie er auf der SPS definiert wurde.

Für das ENUM können folgende Werte vergeben werden : -32768 bis 32767
Sollte ein ENUM für die Kommunikation in die Cloud markiert werden, muss intern nur 1 Wert markiert werden. Dadurch steht danach das komplette ENUM zur Verfügung. Damit das Ganze funktioniert, muss das intern markierte ENUM danach entweder in einem Funktionsblock, Struktur oder direkt in einem Programmaufruf zusätzlich markiert werden.

Markierung intern im ENUM:

```
{attribute 'qualified_only'}
{attribute 'strict'}
TYPE eEnumTest :
(
    eValue1 := 1,       //{#lynus.ag#()}
    eValue2 := 2,
    eValue3 := 3
);
END_TYPE
```

Aufruf direkt in einem Programm:

```
PROGRAM MAIN
VAR
    eEnumMain           : eEnumTest;                //{#lynus.ag#()}
END_VAR
```

Aufruf in einer Struktur:

```
TYPE st_Test :
STRUCT
    bVar1           : BOOL;         //{#lynus.ag#()}
    eEnum           : eEnumTest;    //{#lynus.ag#()}
END_STRUCT
END_TYPE
```

Struktur

Ab Version 2 können unter Verwendung des TMC Files auch Datentypen vom Typ STRUCT markiert werden. Bei Strukturen ist es auch möglich, nur bestimmte Variablen einer Struktur für die Cloud-Kommunikation zu markieren. Jede Struktur darf maximal eine weitere Unterstruktur haben.
Funktionsblöcke dürfen in Strukturen nicht vorkommen. Verwendet werden darf in Strukturen jeder Standard Datentyp aus „Unterstützte Datentypen“. Damit das Ganze funktioniert, muss die intern markierte Struktur danach entweder in einem Funktionsblock, einer weiteren Struktur oder direkt in einem Programmaufruf zusätzlich markiert werden.

Markierung Intern in der Struktur:

```
TYPE st_Test :
STRUCT
    bVar1           : BOOL;         //{#lynus.ag#()}
    eEnum           : eEnumTest;    //{#lynus.ag#()}
END_STRUCT
END_TYPE
```

Aufruf direkt in einem Programm:

```
PROGRAM MAIN
VAR

    ststruct            : st_Test;                  //{#lynus.ag#()}

END_VAR
```

Markierung intern nur von bestimmten Variablen:

```
TYPE st_Test :
STRUCT
    bVar1           : BOOL;         //{#lynus.ag#()}
    eEnum           : eEnumTest;
END_STRUCT
END_TYPE
```

Markierung intern in einer Struktur:

```
TYPE st_Input :
STRUCT
    bOn             : BOOL;         //{#lynus.ag#()}
    bOff            : BOOL;         //{#lynus.ag#()}
    rDimmlevelSet   : REAL;         //{#lynus.ag#()}
END_STRUCT
END_TYPE
```

Markierung in einem Funktionsblock:

```
FUNCTION_BLOCK FB_Dimmer
VAR_INPUT
    stInput         :st_Input;      //{#lynus.ag#()}
END_VAR
VAR_OUTPUT
    bLightState     : BOOL;         //{#lynus.ag#()}
    rDimmlevelState : REAL;         //{#lynus.ag#()}
    stError         : st_Dimmer;    //{#lynus.ag#()}
END_VAR
VAR
    FPON            : R_TRIG;
    FPOFF           : R_TRIG;
END_VAR
```

Funktionsblöcke

Ab Version 2 können unter Verwendung des TMC Files auch Datentypen vom Typ FUNCTIONBLOCK markiert werden. Bei Funktionsblöcken ist es auch möglich, nur bestimmte Variablen eines Funktionsblockes für die Cloud-Kommunikation zu markieren. Jeder Funktionsblock darf maximal eine weitere Unterstruktur haben. Verwendet werden darf in Strukturen jeder Standard Datentyp aus „Unterstützte Datentypen“. Funktionsblöcke in Funktionsblöcken zu markieren ist nicht erlaubt.

Damit das Ganze funktioniert, muss der Funktionsblock danach in einem Programmaufruf zusätzlich markiert werden.

Markierung der Symbole intern im Funktionsblock:

```
FUNCTION_BLOCK FB_Dimmer
VAR_INPUT
    stInput         :st_Input;      //{#lynus.ag#()}
END_VAR
VAR_OUTPUT
    bLightState     : BOOL;         //{#lynus.ag#()}
    rDimmlevelState : REAL;         //{#lynus.ag#()}
    stError         : st_Dimmer;    //{#lynus.ag#()}
END_VAR
VAR
    FPON            : R_TRIG;
    FPOFF           : R_TRIG;
END_VAR
```

Aufruf direkt in einem Programm:

```
PROGRAM MAIN
VAR

    fbDimmer            : FB_Dimmer;                //{#lynus.ag#()}

END_VAR
```

Beispiele

Folgend finden Sie einige einfache Beispiele, wie man den EfficientIO Communicator auf der SPS anwenden kann.

Licht ein und ausschalten

Der Funktionsblock FB_Light beinhaltet die Logik des Programmierers, um ein Licht ein und auszuschalten. Zusätzlich möchte man über die Variablen `bSwitchON` und `bSwitchOFF` das Licht über die Cloud steuern. Die Variable `bStateLight` kann die Rückmeldung in die Cloud Liefern ob das Licht auf der SPS Ein oder Aus ist.

```
PROGRAM prgKitchen
VAR
    fbLight         : FB_Light;
    bSwitchON       : BOOL;                 //{#lynus.ag#()}
    bSwitchOFF      : BOOL;                 //{#lynus.ag#()}
    bStateLight     : BOOL;                 //{#lynus.ag#()}
END_VAR
```

```
fbLight(bOn:= bSwitchON, bOff:= bSwitchOFF, bLightState=> bStateLight);
```

Aufruf direkt im Funktionsblock::

```
FUNCTION_BLOCK FB_Light
VAR_INPUT
    bOn             : BOOL;         //{#lynus.ag#()}
    bOff            : BOOL;         //{#lynus.ag#()}
END_VAR
VAR_OUTPUT
    bLightState     : BOOL;         //{#lynus.ag#()}
END_VAR
```

Aufruf im Programm:

```
PROGRAM prgKitchen
VAR
    fbLight         : FB_Light;     //{#lynus.ag#()}
END_VAR
```

```
fbLight(bOn:= , bOff:= , bLightState=> );
```

Dimmbares Licht

Dieser Funktionsblock unterscheidet sich vom vorigen Beispiel dadurch, dass dieses Licht zusätzlich noch gedimmt werden kann. Über die Variable `rDimmlevelSet` kann ein Dimmwert von 0%-100% an den Funktionsblock übermittelt werden.

```
PROGRAM prgKitchen
VAR
    fbLight         : FB_Dimmer;
    bSwitchOn       : BOOL;         //{#lynus.ag#()}
    bSwitchOff      : BOOL;         //{#lynus.ag#()}
    rDimmlevelSet   : REAL;         //{#lynus.ag#()}
    bLightState     : BOOL;         //{#lynus.ag#()}
    rDimmlevelState : REAL;         //{#lynus.ag#()}
END_VAR
```

```
fbLight(
    bOn:= bSwitchOn,
    bOff:= bSwitchOff,
    rDimmlevelSet:= rDimmlevelSet,
    bLightState=> bLightState,
    rDimmlevelState=> rDimmlevelState,
    stError=> );
```

Aufruf direkt im Funktionsblock::

```
FUNCTION_BLOCK FB_Dimmer
VAR_INPUT
    bOn             : BOOL;         //{#lynus.ag#()}
    bOff            : BOOL;         //{#lynus.ag#()}
    rDimmlevelSet   : REAL;         //{#lynus.ag#()}
END_VAR
VAR_OUTPUT
    bLightState     : BOOL;         //{#lynus.ag#()}
    rDimmlevelState : REAL;         //{#lynus.ag#()}
    stError         : st_Dimmer;    //{#lynus.ag#()}
```

Aufruf im Programm:

```
PROGRAM prgKitchen
VAR
    fbLight         : FB_Dimmer;    //{#lynus.ag#()}
END_VAR
```

```
fbLight(
    bOn:= ,
    bOff:= ,
    rDimmlevelSet:= ,
    bLightState=> ,
    rDimmlevelState=> ,
    stError=> );
```

Temperatursensor

Mit diesem Beispiel wird gezeigt, wie ein Temperaturwert aus der SPS in die Cloud geschickt werden kann. `rValue` spiegelt den Temperaturwert in der Küche wieder.

```
PROGRAM prgKitchen
VAR
    fbTemp          : FB_TempSensor;
    rValue          : REAL;             //{#lynus.ag#()}
END_VAR
```

```
fbTemp(rTempValue=> rValue);
```

Aufruf direkt im Funktionsblock::

```
FUNCTION_BLOCK FB_TempSensor
VAR_INPUT
END_VAR
VAR_OUTPUT
    rTempValue      : REAL;     //{#lynus.ag#()}
END_VAR
```

Aufruf im Programm:

```
PROGRAM prgKitchen
VAR
    fbTemp          : FB_TempSensor;    //{#lynus.ag#()}
END_VAR
```

```
fbTemp(rTempValue=> );
```

Statusüberwachung von 2 Förderbandanlagen

Dieses Beispiel zeigt, wie eine Förderbandanlage mit 2 Förderbändern überwacht werden kann. Eine Steuerung der Förderbänder wäre auch möglich, ist hier zur Veranschaulichung einfachheitshalber weggelassen.

```
PROGRAM prgMotion
VAR
    fbConveyorPartsEntry        : FB_MotionControl;
    fbConveyorPartsExit         : FB_MotionControl;
    bErrorConveyorEntry         : BOOL;             //{#lynus.ag#()}
    bErrorConveyorExit          : BOOL;             //{#lynus.ag#()}
    iSateConveyorEntry          : INT;              //{#lynus.ag#()}
    iSateConveyorExit           : INT;              //{#lynus.ag#()}
    udiErrorCodeConveyorEntry   : UDINT;            //{#lynus.ag#()}
    udiErrorCodeConveyorExit    : UDINT;            //{#lynus.ag#()}
END_VAR
```

```
fbConveyorPartsEntry(
    bStartStop:= ,
    bDirection:= ,
    rSpeed:= ,
    iStateMotion=> iSateConveyorEntry,
    bError=> bErrorConveyorEntry,
    udiErrorCode=> udiErrorCodeConveyorEntry);

fbConveyorPartsExit(
    bStartStop:= ,
    bDirection:= ,
    rSpeed:= ,
    iStateMotion=> iSateConveyorExit,
    bError=> bErrorConveyorExit,
    udiErrorCode=> udiErrorCodeConveyorExit);
```

Aufruf direkt im Funktionsblock::

```
FUNCTION_BLOCK FB_MotionControl
VAR_INPUT
    bStartStop      : BOOL;
    bDirection      : BOOL;
    rSpeed          : REAL;
END_VAR
VAR_OUTPUT
    iStateMotion    : INT;              //{#lynus.ag#()}
    bError          : BOOL;             //{#lynus.ag#()}
    udiErrorCode    : UDINT;            //{#lynus.ag#()}
```

Aufruf im Programm:

```
PROGRAM prgMotion
VAR
    fbConveyorPartsEntry        : FB_MotionControl;     //{#lynus.ag#()}
    fbConveyorPartsExit         : FB_MotionControl;     //{#lynus.ag#()}
END_VAR
```

```
fbConveyorPartsEntry(
    bStartStop:= ,
    bDirection:= ,
    rSpeed:= ,
    iStateMotion=> ,
    bError=> ,
    udiErrorCode=> );

fbConveyorPartsExit(
    bStartStop:= ,
    bDirection:= ,
    rSpeed:= ,
    iStateMotion=> ,
    bError=> ,
    udiErrorCode=> );
```