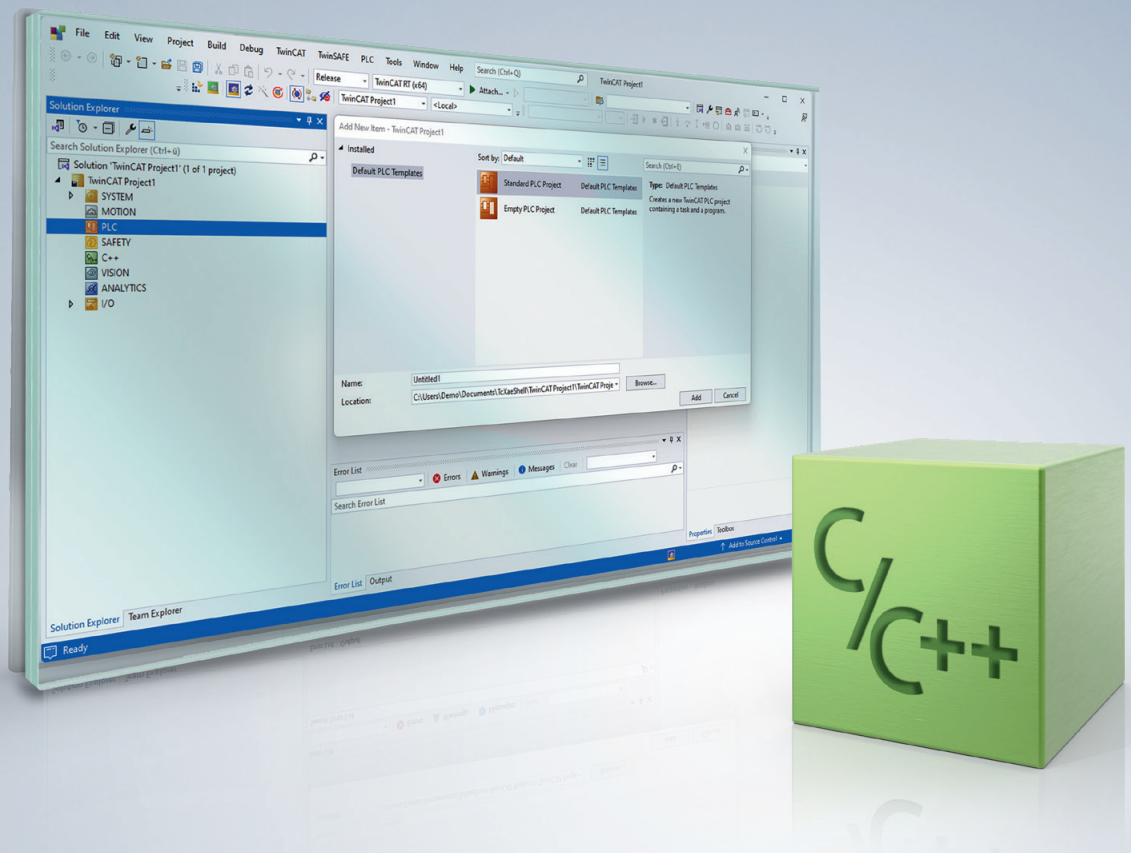
BECKHOFF New Automation Technology

Manual | EN

TwinCAT 3

C/C++



2026-04-27 | Version: 1.19.1

Table of contents

1 Foreword 7
1.1 Notes on the documentation 7
1.2 For your safety 8
1.3 Notes on information security 9
1.4 Documentation issue status 9
2 Overview 10
3 Introduction 11
3.1 Difference between user mode and real time programming 13
4 Installation 17
4.1 System requirements 17
4.2 Installation 17
5 Preparation - only once 20
5.1 Visual Studio - TwinCAT XAE Base toolbar 20
5.2 Module signing 20
5.2.1 TwinCAT 21
5.2.2 Operating system 30
6 Modules 35
6.1 The TwinCAT Component Object Model (TcCOM) concept 35
6.1.1 TwinCAT module properties 37
6.1.2 TwinCAT module state machine 44
6.2 Module-to-module communication 46
7 Modules - Handling 49
7.1 Versioned C++ Projects 49
7.2 Starting Modules 50
7.3 TwinCAT Loader 50
7.3.1 Test signing 51
7.3.2 Encrypting Modules 53
7.3.3 Return Codes 54
7.3.4 TcSignTool - Storage of the certificate password outside the project 55
8 TwinCAT C++ development 56
9 Quick Start 58
9.1 Create TwinCAT 3 project 58
9.2 Create TwinCAT 3 C++ project 59
9.3 TwinCAT 3 C++ Configure project 63
9.4 Implement TwinCAT 3 C++ project 64
9.5 Publish TwinCAT 3 C++ project in version 0.0.0.1 66
9.6 Implement and publish TwinCAT 3 C++ project version 0.0.0.2 66
9.7 Create TwinCAT 3 C++ Module instance 68
9.8 TwinCAT 3 enable C++ debugger 70
9.9 Create a TwinCAT task and apply it to the module instance 71
9.10 Activating a TwinCAT 3 project 73
9.11 TwinCAT 3 C++ Implement project Online Change 74

10 Debugging ... 75
10.1 Details of Conditional Breakpoints ... 78
10.2 Visual Studio tools ... 79
10.3 Working with TwinCAT Task Dumps ... 81
10.4 Examining dump files using WinDbg ... 82
10.5 Debugging the state machine with the Usermode Runtime ... 83

11 Wizards ... 85
11.1 TwinCAT C++ Project Wizard ... 85
11.2 TwinCAT Module Class Wizard ... 86
11.3 TwinCAT Module Class Editor (TMC) ... 88
11.3.1 Overview ... 90
11.3.2 Basic Information ... 91
11.3.3 Data Types ... 91
11.3.4 Modules ... 109
11.4 TwinCAT Module Instance Configurator ... 129
11.4.1 Object ... 130
11.4.2 Context ... 131
11.4.3 Parameter (Init) ... 131
11.4.4 Data Area ... 132
11.4.5 Interfaces ... 132
11.4.6 Interface Pointer ... 132
11.4.7 Data Pointer ... 132
11.5 Customer-specific project templates ... 133
11.5.1 Overview ... 133
11.5.2 Files involved ... 134
11.5.3 Transformations ... 134
11.5.4 Notes on handling ... 136

12 Programming Reference ... 139
12.1 TwinCAT C++ Project properties ... 140
12.1.1 Tc SDK ... 142
12.1.2 Tc Extract Version ... 143
12.1.3 Tc Publish ... 143
12.1.4 Tc Sign ... 144
12.2 File Description ... 145
12.3 Online Change ... 148
12.4 Limitations for TwinCAT C++ modules ... 149
12.5 Memory allocation ... 150
12.6 Static variables ... 152
12.7 Multi-task data access synchronization ... 153
12.7.1 CriticalSections ... 154
12.7.2 Semaphores ... 156
12.7.3 FIFO template classes ... 157
12.8 Interfaces ... 158
12.8.1 Interface ITcPostCyclicCaller ... 159
12.8.2 Return values ... 161

12.8.3 Interface ITcCyclic ... 161
12.8.4 Interface ITcCyclicCaller ... 162
12.8.5 Interface ITcFileAccess ... 164
12.8.6 Interface ITcFileAccessAsync ... 171
12.8.7 Interface ITcIoCyclic ... 172
12.8.8 Interface ITcIoCyclicCaller ... 174
12.8.9 ITComOnlineChange interface ... 175
12.8.10 ITComObject interface ... 177
12.8.11 ITComObject interface (C++ convenience) ... 181
12.8.12 Interface ITcPostCyclic ... 182
12.8.13 Interface ITcRTimeTask ... 183
12.8.14 Interface ITcTask ... 183
12.8.15 Interface ITcTaskNotification ... 187
12.8.16 Interface ITcUnknown ... 187
12.9 ADS Communication ... 189
12.9.1 AdsReadDeviceInfo ... 190
12.9.2 AdsRead ... 192
12.9.3 AdsWrite ... 194
12.9.4 AdsReadWrite ... 196
12.9.5 AdsReadState ... 198
12.9.6 AdsWriteControl ... 200
12.9.7 AdsAddDeviceNotification ... 201
12.9.8 AdsDelDeviceNotification ... 203
12.9.9 AdsDeviceNotification ... 205
12.10 Mathematical Functions ... 206
12.11 Time Functions ... 207
12.12 STL / Containers ... 208
12.13 Error messages - understanding ... 209
12.14 Module messages for the Engineering (logging / tracing) ... 209

13 How to...? ... 213
13.1 Using the Automation Interface ... 213
13.2 Windows 10 as target system up to TwinCAT 3.1 Build 4022.2 ... 213
13.3 Publishing modules on the command line ... 213
13.4 Clone ... 213
13.5 Access Variables via ADS ... 214
13.6 TcCallAfterOutputUpdate for C++ modules ... 214
13.7 Order determination of the execution in a task ... 214
13.8 Setting version/vendor information ... 214
13.9 Renaming TwinCAT C++ projects ... 216
13.10 Delete Module ... 218
13.11 Add revision control and Online Change subsequently ... 219
13.11.1 C++ Project -> Revision control ... 219
13.11.2 C++ Module -> OnlineChange ... 222
13.12 Initialization of TMC-member variables ... 227
13.13 Using PLC strings as method parameters ... 227
13.14 Third Party Libraries ... 228

13.15 Linking via TMC editor (TcLinkTo) ... 228
13.16 Online Change per ADS ... 230

14 Troubleshooting ... 232
14.1 Using C++ classes in TwinCAT C++ module ... 232
14.2 Activation – “invalid object id” (1821/0x71d) ... 232

15 C++-samples ... 233
15.1 Sample01: Cyclic module with IO ... 236
15.2 Sample02: Cyclic C++ logic, which uses IO from the IO Task ... 237
15.3 Sample03: C++ as ADS server ... 237
15.3.1 Sample03: TC3 ADS Server written in C++ ... 238
15.3.2 Sample03: ADS client UI in C# ... 242
15.4 Sample05: C++ CoE access via ADS ... 246
15.5 Sample06: UI-C#-ADS client uploading the symbolic from module ... 247
15.6 Sample07: Receiving ADS Notifications ... 252
15.7 Sample08: provision of ADS-RPC ... 253
15.8 Sample10: module communication: Using data pointer ... 256
15.9 Sample11: Module communication: method call PLC module to C++ module ... 257
15.9.1 TwinCAT 3 C++ module providing methods ... 258
15.9.2 Calling methods offered by another module via PLC ... 272
15.10 Sample11a: Module communication: Method call C++ module to C++ module ... 284
15.11 Sample12: module communication: Using IO mapping ... 285
15.12 Sample13: Module communication: Method call C++ module to PLC module ... 286
15.13 Sample19: Synchronous File Access ... 289
15.14 Sample20: FileIO-Write ... 290
15.15 Sample20a: FileIO-Cyclic Read / Write ... 290
15.16 Sample22: Automation Device Driver (ADD): Access DPRAM ... 292
15.17 Sample23: Structured Exception Handling (SEH) ... 293
15.18 Sample24: Semaphores ... 295
15.19 Sample25: Static Library ... 296
15.20 Sample26: Order of execution in a task ... 298
15.21 Sample30: Timing Measurement ... 300
15.22 Sample31: Functionblock TON in TwinCAT3 C++ ... 301
15.23 Sample35: Access Ethernet ... 302
15.24 Sample37: Archive data ... 303
15.25 TcCOM samples ... 304
15.25.1 TcCOM_Sample01_PlcToPlc ... 304
15.25.2 TcCOM_Sample02_PlcToCpp ... 314
15.25.3 TcCOM_Sample03_PlcCreatesCpp ... 318

16 Appendix ... 323
16.1 ADS Return Codes ... 323
16.2 Retain data ... 326
16.3 Creating and handling C++ projects and modules ... 329
16.4 Creating and handling TcCOM modules ... 333
16.5 Third-party components ... 337

1 Foreword

1.1 Notes on the documentation

This description is intended exclusively for trained specialists in control and automation technology who are familiar with the applicable national standards.
The documentation and the following notes and explanations must be complied with when installing and commissioning the components.
The trained specialists must always use the current valid documentation.

The trained specialists must ensure that the application and use of the products described is in line with all safety requirements, including all relevant laws, regulations, guidelines, and standards.

Disclaimer

The documentation has been compiled with care. The products described are, however, constantly under development.
We reserve the right to revise and change the documentation at any time and without notice.
Claims to modify products that have already been supplied may not be made on the basis of the data, diagrams, and descriptions in this documentation.

Trademarks

Beckhoff®, ATRO®, EtherCAT®, EtherCAT G®, EtherCAT G10®, EtherCAT P®, MX-System®, Safety over EtherCAT®, TC/BSD®, TwinCAT®, TwinCAT/BSD®, TwinSAFE®, XFC®, XPlanar®, and XTS® are registered and licensed trademarks of Beckhoff Automation GmbH.
If third parties make use of the designations or trademarks contained in this publication for their own purposes, this could infringe upon the rights of the owners of the said designations.



EtherCAT® is a registered trademark and patented technology, licensed by Beckhoff Automation GmbH, Germany.

Copyright

© Beckhoff Automation GmbH & Co. KG, Germany.
The distribution and reproduction of this document, as well as the use and communication of its contents without express authorization, are prohibited.
Offenders will be held liable for the payment of damages. All rights reserved in the event that a patent, utility model, or design are registered.

Third-party trademarks

Trademarks of third parties may be used in this documentation. You can find the trademark notices here: https://www.beckhoff.com/trademarks.

1.2 For your safety

Safety regulations

Read the following explanations for your safety.
Always observe and follow product-specific safety instructions, which you may find at the appropriate places in this document.

Exclusion of liability

All the components are supplied in particular hardware and software configurations which are appropriate for the application. Modifications to hardware or software configurations other than those described in the documentation are not permitted, and nullify the liability of Beckhoff Automation GmbH & Co. KG.

Personnel qualification

This description is only intended for trained specialists in control, automation, and drive technology who are familiar with the applicable national standards.

Signal words

The signal words used in the documentation are classified below. In order to prevent injury and damage to persons and property, read and follow the safety and warning notices.

Personal injury warnings

⚠ DANGER
Hazard with high risk of death or serious injury.

⚠ WARNING
Hazard with medium risk of death or serious injury.

⚠ CAUTION
There is a low-risk hazard that could result in medium or minor injury.

Warning of damage to property or environment

NOTICE
The environment, equipment, or data may be damaged.

Information on handling the product



This information includes, for example:
recommendations for action, assistance or further information on the product.

1.3 Notes on information security

The products of Beckhoff Automation GmbH & Co. KG (Beckhoff), insofar as they can be accessed online, are equipped with security functions that support the secure operation of plants, systems, machines and networks. Despite the security functions, the creation, implementation and constant updating of a holistic security concept for the operation are necessary to protect the respective plant, system, machine and networks against cyber threats. The products sold by Beckhoff are only part of the overall security concept. The customer is responsible for preventing unauthorized access by third parties to its equipment, systems, machines and networks. The latter should be connected to the corporate network or the Internet only if appropriate protective measures have been set up.

In addition, the recommendations from Beckhoff regarding appropriate protective measures should be observed. Further information regarding information security and industrial security can be found in our https://www.beckhoff.com/secguide.

Beckhoff products and solutions undergo continuous further development. This also applies to security functions. In light of this continuous further development, Beckhoff expressly recommends that the products are kept up to date at all times and that updates are installed for the products once they have been made available. Using outdated or unsupported product versions can increase the risk of cyber threats.

To stay informed about information security for Beckhoff products, subscribe to the RSS feed at https://www.beckhoff.com/secinfo.

1.4 Documentation issue status

Version	Change
1.19.x	**New:** System requirements [▶ 17] Restrictions for the operation of TC1300 TwinCAT C++ under Beckhoff RT Linux® Examining dump files using WinDbg [▶ 82]
1.18.x	**New:** Working with TwinCAT Task Dumps [▶ 81] Installation [▶ 17] Online Change per ADS [▶ 230] Third-party components [▶ 337]

2 Overview

This chapter deals with the implementation of TwinCAT 3 in C/C++. The most important chapters are:

- Basics
  Which platforms are supported? Are additional programs required to implement TwinCAT 3 C++ modules?
  The answers can be found in Requirements [▶ 17] and Preparation [▶ 20], the following limitations [▶ 149] apply.
- Quick start [▶ 58]
  This is a "less than five minute example" to implement a simple, cyclic counter in C++. The counter value is monitored and overwritten, debugging options are described, etc.
- MODULE [▶ 37]
  Modularity is the basic philosophy of TwinCAT 3. An understanding of the modular concept of TwinCAT 3 is the basic requirement, especially for C++ modules.
  Basic knowledge of the architecture of TwinCAT modules is required.
- Wizards [▶ 85]
  Documentation of the visual components of the TwinCAT C++ environment.
  This includes tools for creating projects and tools for editing modules and configuring module instances.
- Programming reference [▶ 139]
  This chapter contains detailed information on programming in TwinCAT C++. Here you will find interfaces and other TwinCAT functions for ADS communication and auxiliary methods.
- The "How to ...? [▶ 213]" chapter contains useful tips for working with TwinCAT C++.
- Samples
  Some interfaces and their use are described in detail in the form of an executable program, which is provided as a download with source code and solution.

3 Introduction

The method of emulating classic automation devices such as programmable logic controllers (PLC) and numerical controllers (NC) as software on powerful standard hardware has been the state of the art for many years and is now practiced by many manufacturers.

There are many benefits, but the most important is without doubt the fact that the software is mostly hardware-independent. This means, firstly, that the performance of the hardware can be specially adapted to the application and, secondly, that you can automatically benefit from its further development.

This particularly applies to PC hardware, whose performance is still increasingly at a dramatically fast rate. The relative independence from a supplier that results from this separation of software and hardware is also very important for the user.

Since the PLC and Motion Control – and possibly other automation components – remain independent logic function blocks with this method, there are only a few changes in the application architecture in comparison with classic automation technology.

The PLC determines the machine's logical processes and transfers the implementation of certain axis functions to the Motion Control. On account of the improved performance of the controllers and the possibility to use higher-level programming languages (IEC 61131-3), even complex machines can be automated in this way.

Modularization

In order to master the complexity of modern machines and at the same time to reduce the necessary engineering expenditure, many machine manufacturers have begun to modularize their machines. Individual functions, assemblies or machine units are thereby regarded as modules, which are as independent as possible and are embedded into the overall system via uniform interfaces.

Ideally a machine is then structured hierarchically, whereby the lowest modules represent the simplest, continually reusable basic elements. Joined together they form increasingly complex machine units, up to the highest level where the entire machine is created. Different approaches are followed when it comes to the control system aspects of machine modularization. These can be roughly divided into a decentralized and a centralized approach.

In the local approach, each machine module is given its own controller, which determines the PLC functions and possibly also the motion functions of the module.

The individual modules can be put into operation and maintained separately from one another and scaled relatively independently. The necessary interactions between the controllers are coordinated via communication networks (fieldbuses or Ethernet) and standardized via appropriate profiles.

The central approach concentrates all control functions of all modules in the common controller and uses only very little pre-processing intelligence in the local I/O devices. The interactions can occur much more directly within the central control unit, as the communication paths become much shorter. Dead times do not occur and use of the control hardware is much more balanced, which reduces overall costs.

However, the central method also has the disadvantage that the necessary modularization of the control software is not automatically specified. At the same time, the possibility of being able to access any information from other parts of the program in the central controller obstructs the module formation and the reusability of this control software in other applications. Since no communication channel exists between the control units, an appropriate profile formation and standardization of the control units frequently fall by the wayside.

The best of both worlds

The ideal controller for modular machines uses elements from decentralized and centralized control architecture. A central, powerful computer platform of the most general kind possible serves 'as always' as the control hardware.

The benefits of centralized control technology:

- low overall costs
- available

- fast, modular fieldbus system (keyword: EtherCAT)
- and the possibility to access all information in the system without loss of communication

are decisive arguments.

The above-mentioned benefits of a decentralized approach can be implemented in the centralized control system by means of suitable modularization of the control software.

Instead of allowing a large, complex PLC program and an NC with many axes to run, many small ‘controllers’ can co-exist in a common runtime on the same hardware with relative independence from one another. The individual control modules are self-contained and make their functions available to the environment via standard interfaces, or they use corresponding functions of other modules or the runtime.

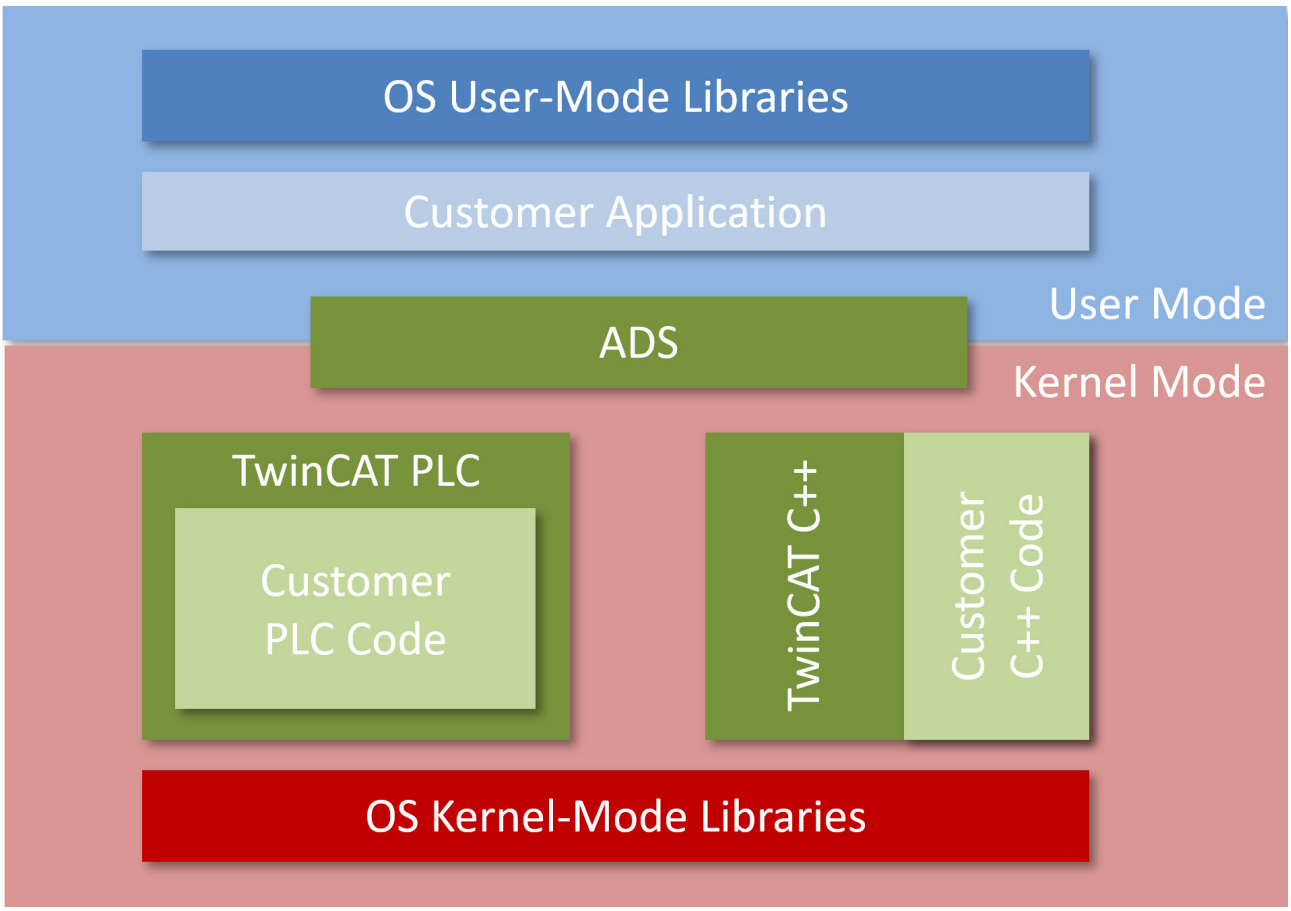
A significant profile is created through the definition of these interfaces and the standardization of the corresponding parameters and process data. Since the individual modules are implemented in a runtime, direct calls of other modules are also possible – once again via corresponding standard interfaces. In this way the modularization can take place within sensible limits without communication losses occurring.

During the development or commissioning of individual machine modules, the associated control modules can be created and tested on any control hardware with the appropriate runtime. Missing connections to other modules can be emulated during this phase. On the complete machine they are then instanced together on the central runtime, which only needs to be dimensioned such that the resource requirements of all instanced modules (memory, tasks and computing power) are fulfilled.

TwinCAT 3 runtime

TwinCAT Runtime offers a software environment in which the TwinCAT modules are loaded, implemented and managed. It provides more basic functions so that system resources (memory, tasks, fieldbus and hardware access, etc.) can be used. The individual modules must not have been created with the same compiler. This means that they can be independent of each other and come from different suppliers.

When the runtime starts, some system modules are automatically loaded so that their properties are available to other modules. However, the properties of the system modules are accessed in the same way as the properties of normal modules, so it does not matter to the modules whether the respective property is provided by a system module or a normal module.

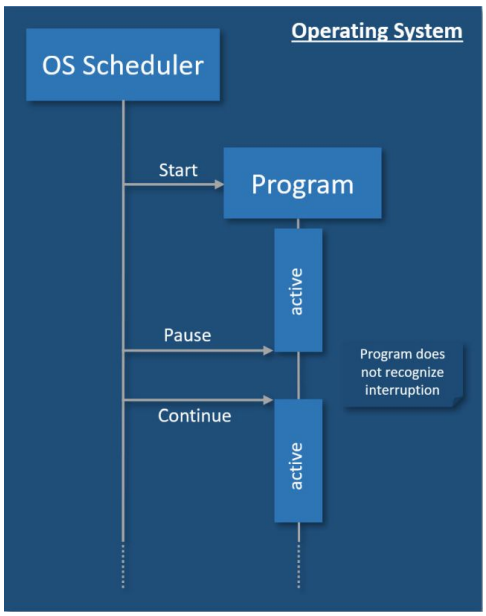


Unlike the PLC, where the user-defined program is executed in a runtime environment, TwinCAT C++ modules are not in such a hosted environment. This causes TwinCAT C++ modules (.tmx) to be executed as kernel modules, which are loaded by TwinCAT.

3.1 Difference between user mode and real time programming

This article describes the conceptual differences between standard user mode programming in a programming language such as C++, C# or Java, and real-time programming in TwinCAT.

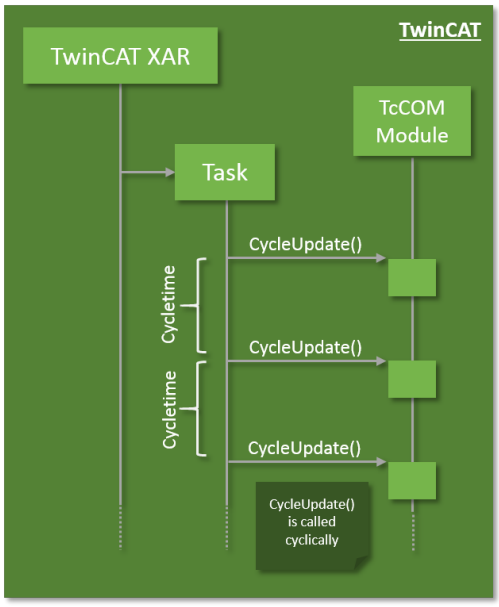
The article particularly focuses on real-time programming with TwinCAT C++, because this is where previous knowledge with C++ programming comes to the fore and the sequence characteristics of the TwinCAT real-time system have to be taken into account.



With conventional user mode programming, e.g. in C#, a program is created, which is then executed by an operating system.

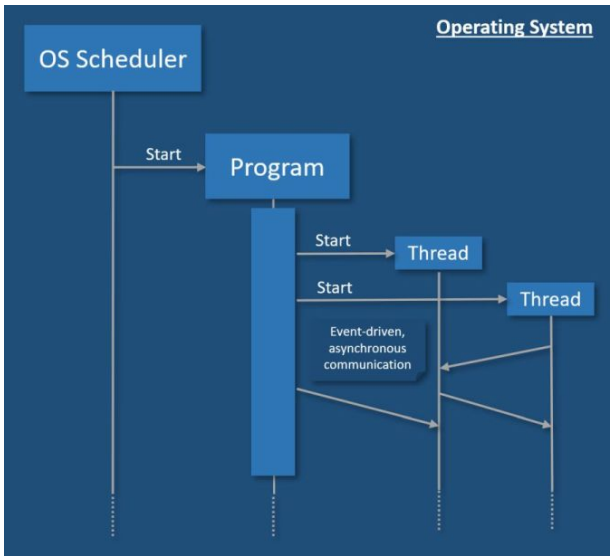
The program is started by the operating system and can run independently, i.e. it has full control over its own execution, including aspects such as threading and memory management. In order to enable multitasking, the operating system interrupts such a program at any time and for any period. The program does not register such an interruption. The operating system must ensure that such interruptions remain unnoticed by the user. The data exchange between the program and its environment is event-driven, i.e. non-deterministic and often blocking.

The behavior is not adequate for execution under real-time conditions, because the application itself must be able to rely on the available resources in order to be able to ensure real-time characteristics (response guarantees).



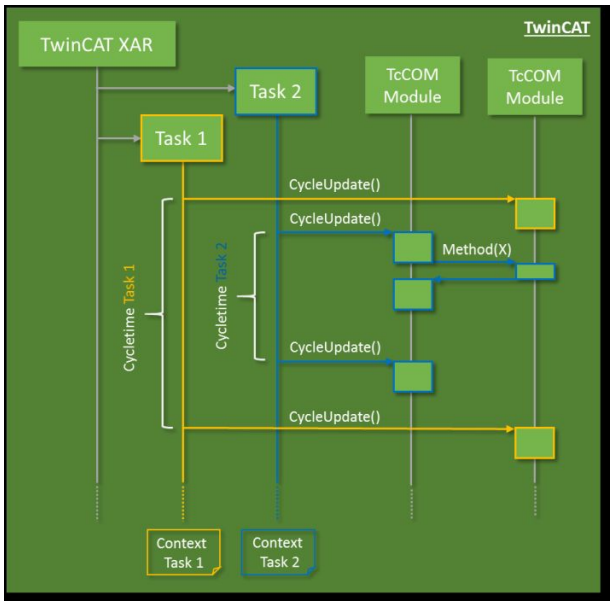
The basic idea of PLC is therefore adopted for TwinCAT C++: the TwinCAT real-time system manages the real-time tasks, handles the scheduling and cyclically calls an entry point in the program code. The program execution must be completed within the available cycle length and return the control. The TwinCAT system makes the data from the I/O area available in the process images, so that consistent access can be guaranteed. This means that the program code itself cannot use mechanisms such as threading.

Concurrency



With conventional programming in user mode, concurrency is controlled by the program. This is where threads are started, which communicate with each other. All these mechanisms require resources, which have to be allocated and enabled, which can compromise the real-time capability. The communication between the threads is event-based, so that a calling thread has no control over the processing time in the called thread.

In TwinCAT, tasks are used for calling modules, which therefore represents concurrency. Tasks are assigned to a core; they have cycle times and priorities, with the result that a higher-priority task can interrupt a lower-priority task. If several cores are used, tasks are executed concurrently in practice.



Modules can communicate with each other, so that data consistency has to be ensured in concurrency mode.
Data exchange across task boundaries is enabled through mapping, for example. When direct data access via methods is used, it must be protected through Critical sections, for example.

Startup/shutdown behavior

The TwinCAT C++ code is executed in the so-called "kernel context" and the "TwinCAT real-time context", not as a user mode application.

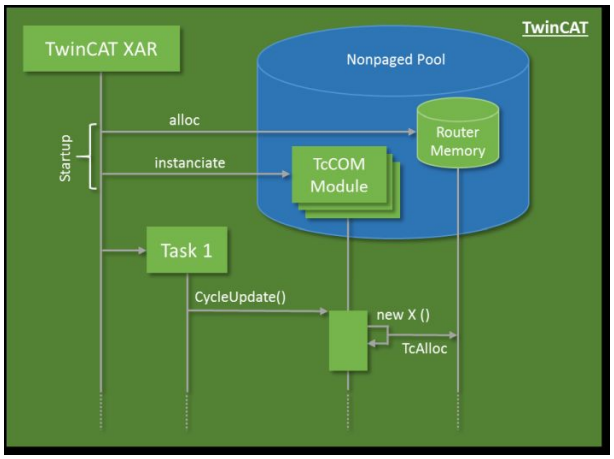
During startup/shutdown of the modules, code for (de)initialization is initially executed in the kernel context; only the last phase and the cyclic calls are executed in the TwinCAT real-time context.

Details are described in chapter Module state machine [▶ 44].

Memory management

TwinCAT has its own memory management, which can also be used in the real-time context. This memory is obtained from what is referred to as the "non-paged pool", which is provided by the operating system. In this memory the TcCOM modules are instantiated with their memory requirement.

In addition, the so-called "router memory" is provided by TwinCAT in this memory area, from which the TcCOM modules can allocate memory dynamically in the real-time context (e.g. with the New operator).



If possible, memory should generally be allocated in advance, not in the cyclic code. During each allocation a check is required to verify that the memory is actually available. For allocations in the cyclic code, the execution therefore depends on the memory availability.

4 Installation

4.1 System requirements

Engineering (XAE)

Technical data	Requirements
Operating system	Windows 10/11
TwinCAT version	TwinCAT 3.1 Build 4024 | Build 4026
Required TwinCAT license	None (however, Visual Studio license from Microsoft)

Runtime (XAR)

Technical data	Requirements
Operating system	Windows 10, TwinCAT/BSD®, Beckhoff RT Linux® (with restrictions, see below)
Target platform	x86, x64, x64-E
TwinCAT version	TwinCAT 3.1 Build 4024 | Build 4026
Required TwinCAT license	TC1300

Restrictions for the operation of TC1300 TwinCAT C++ under Beckhoff RT Linux®

TwinCAT C++ modules cannot currently be used to their full extent under Beckhoff RT Linux®. This concerns the following restrictions:

- TwinCAT XAE Engineering and the "TC1300 C++" workload must be installed in at least version 4026.21 for the build environment to be available.
- It must be compiled for the "TwinCAT OS (x64-E)" or "TwinCAT OS (ARMV8-A)" platforms. These platforms can be added to existing projects via the **TwinCAT Upgrade C++ Project** context menu item.
- Debugging (connection of the TwinCAT Debugger from Engineering) is not available. It can be developed and debugged with other operating systems (Windows, TwinCAT/BSD®) as a runtime system and Beckhoff RT Linux® can be used for execution.
- SEH (Structured Exception Handling) [▶ 293] cannot be used.
- Further restrictions are described in the section Programming Reference [▶ 139].

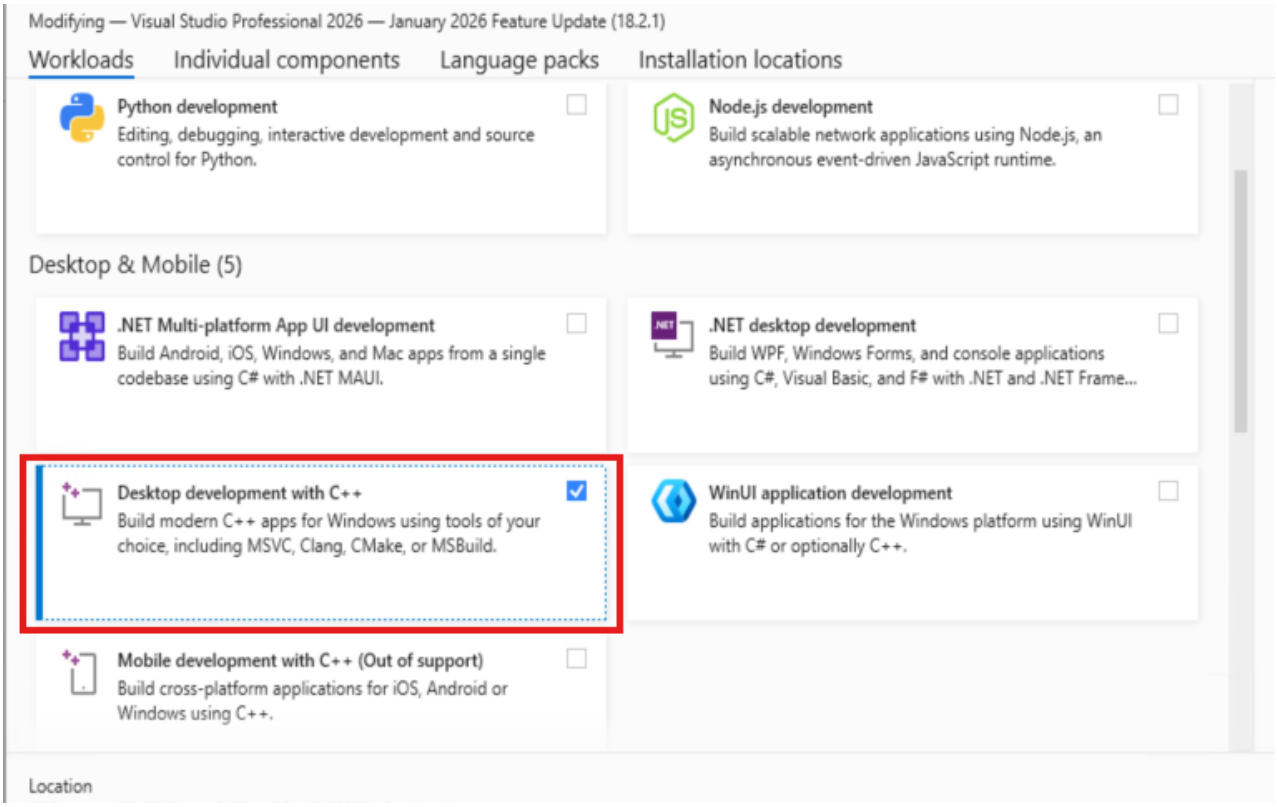
4.2 Installation

Engineering prerequisites

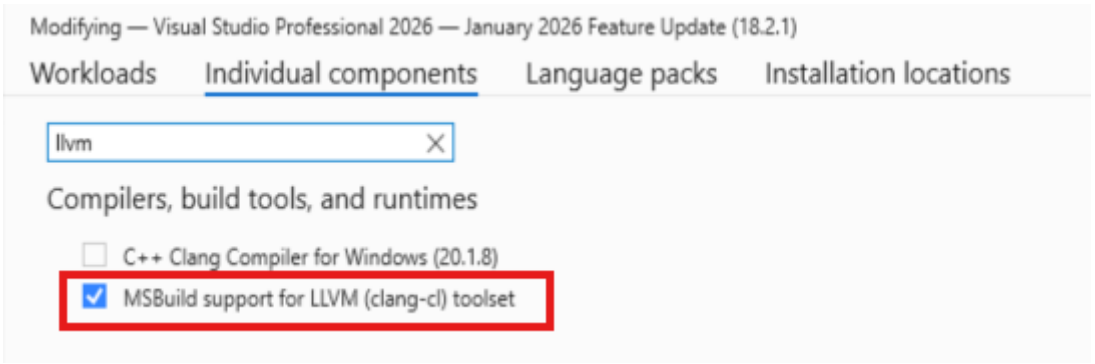
For engineering, Microsoft Visual Studio 2017, 2019, 2022, or 2026 Professional/ Enterprise must be installed on the engineering PC. However, XAE Shell is sufficient for integrating and using existing binary C++ modules in a TwinCAT 3 PLC environment (Visual Studio is not required).

For engineering, additional options must be activated in the Visual Studio installer for the respective Visual Studio version in which TwinCAT is to be integrated:

“**Desktop development with C++**” must be installed as “Workload” in the Visual Studio installer:



In addition, the "**MSBuild support for LLVM (clang-cl) toolset**" should be installed as "Individual Component" in the Visual Studio installer:



Runtime prerequisites

Microsoft Visual Studio **does not** have to be installed.

- For TwinCAT 3.1 Build 4024.x: TwinCAT 3.1 XAR
- For TwinCAT 3.1 Build 4026.x: TwinCAT Standard Runtime

TwinCAT Package Manager: Installation (TwinCAT 3.1 Build 4026)

Detailed instructions on installing products can be found in the chapter Installing workloads in the TwinCAT 3.1 Build 4026 installation instructions.

Install the following workload to use the product:

TC1300 | TwinCAT 3 C/C++

TwinCAT setup: Installation (TwinCAT 3.1 Build 4024 and earlier)

Is included in the TwinCAT 3.1 Full setup.

TwinCAT/BSD®:

no installation required

TwinCAT Runtime for Linux®:

no installation required

5 Preparation - only once

A PC for the engineering of TwinCAT C++ modules must be prepared. You only have to carry out these steps once:

- Configure the TwinCAT Basis [▶ 20] as well as the configuration and platform toolbar.
- Sign modules so that they can be executed; see Documentation for setting up a test signing [▶ 20].

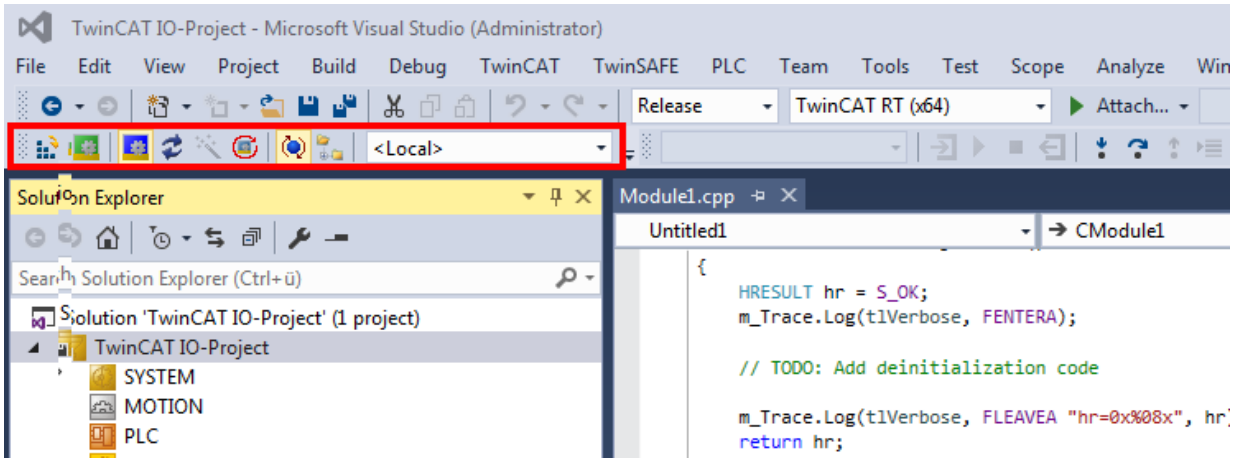
5.1 Visual Studio - TwinCAT XAE Base toolbar

Efficient engineering through TwinCAT XAE base toolbar

TwinCAT 3 integrates its own toolbar in the Visual Studio menu for better efficiency. It assists you in the creation of C++ projects. This toolbar is automatically added to the Visual Studio menu by the TwinCAT 3 setup. If you wish to add it manually, however, do the following:

1. Open the **View** menu and select **Toolbars\TwinCAT XAE Base**

⇨ The selected toolbar appears below the menu.



5.2 Module signing

For Windows and TwinCAT/BSD® runtime systems, TwinCAT C++ modules are signed with a certificate so that they can be executed. Beckhoff RT Linux® systems do not recognize file-based signatures, so this step can be omitted if Beckhoff RT Linux® alone is to be provided.

The signature ensures that only C++ software whose origin can be traced is executed on productive systems.

The C++ modules are loaded by the TwinCAT runtime system and must be signed with a TwinCAT user certificate.

For test purposes, certificates that cannot be verified can be used for signing. However, this is only possible if the operating system is in test mode so that these certificates are not used on productive systems.

No signing for engineering and Beckhoff RT Linux® target systems.

Only execution on Windows and TwinCAT/BSD® requires certificates – the engineering and Beckhoff RT Linux® target systems do not.

Organizational separation of development and production software

Beckhoff recommends working organizationally with (at least) two certificates.

1. A certificate which is not countersigned, thus the test mode is needed for the development process. This certificate can also be issued individually by each developer. The test systems are then set to test mode.

2. Only the software that has passed the corresponding final tests is signed by a countersigned certificate. This software can thus also be installed on machines and delivered.

Such a separation of development and operation ensures that only tested software runs on productive systems.

5.2.1 TwinCAT

Versioned C++ projects are stored as binary in a TMX file (TwinCAT Module Executeable).

For the implementation of TwinCAT 3 C++ modules, this compiled, executable TMX file must be signed with a TwinCAT user certificate if it is to be loaded by the TwinCAT Runtime.

Signing

TwinCAT TMX files can only be loaded after a successful signing.

NOTICE

Signing on 32-bit and 64-bit systems

In contrast to the operating system signing, TwinCAT signing is intended for both 32-bit and 64-bit systems. Thus, the test mode is also assumed for a test signing on 32-bit systems.

For signing a TMX file, a TwinCAT user certificate is required [▶ 21], which is configured accordingly in the project for signing [▶ 144].

Test signing

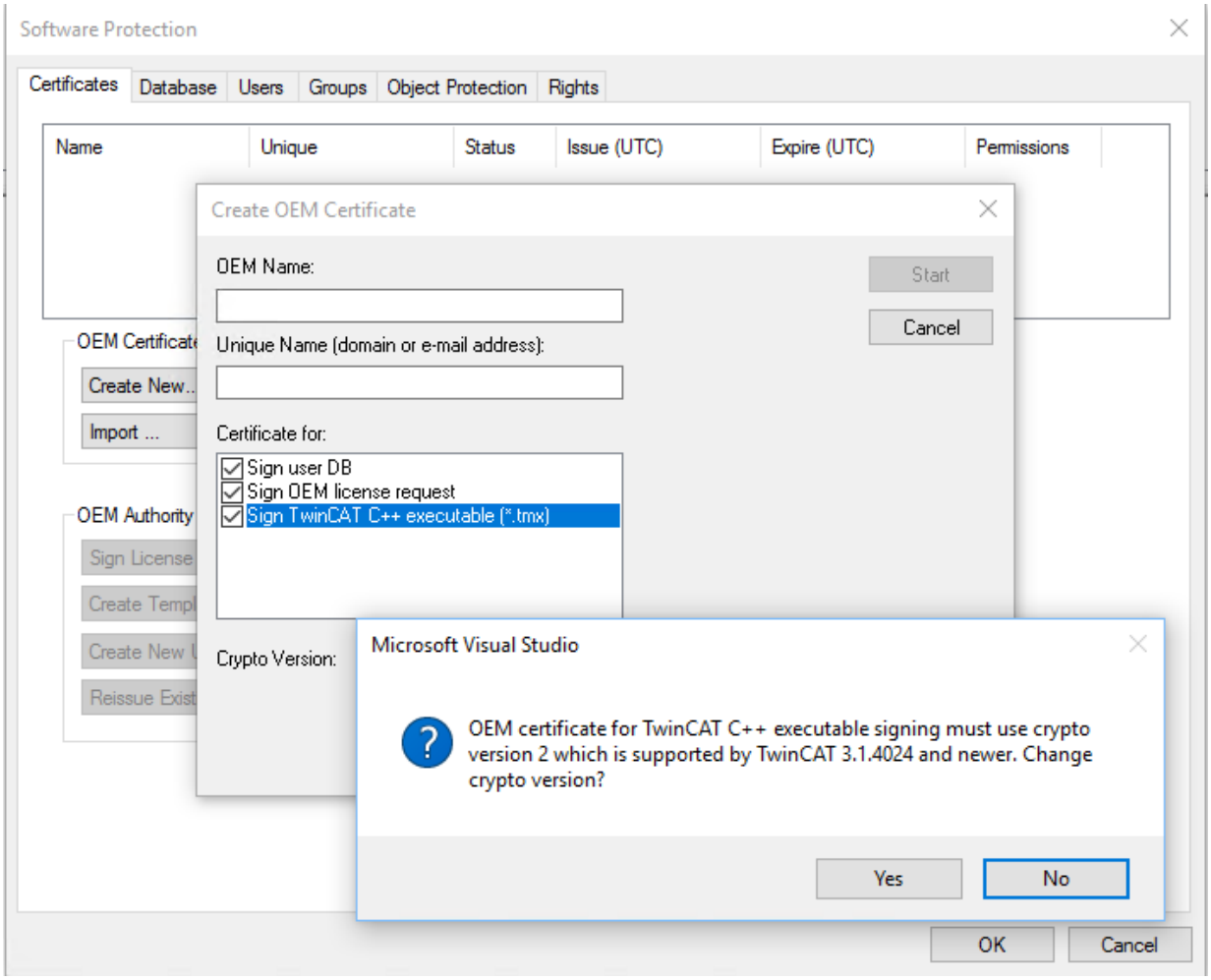
The user certificate can be created locally in TwinCAT. As long as it is not countersigned by Beckhoff, it is necessary to activate the test mode [▶ 21].

As soon as the TwinCAT user certificate has been countersigned by Beckhoff [▶ 23], the test mode can be dispensed with accordingly. It can be deactivated in the same way as it is activated.

5.2.1.1 Test signing

The test signing for TwinCAT can be carried out with the same TwinCAT user certificate as for the actual delivery (see Request TwinCAT 3 user certificate [▶ 24]).

1. For test operation, e.g., during software development, it is sufficient to create a TwinCAT user certificate [▶ 25]. Make sure that you select the purpose "Sign TwinCAT C++ executable (*.tmx)". For this the Crypto version 2 is required, a message appears.

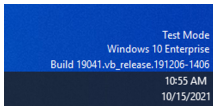


On XAR (and XAE, if it is a local test), activate the test mode so that the operating system can accept the self-signed certificates. This can be done on both engineering systems (XAE) and runtime systems (XAR).

For Windows

Use the administrator prompt to execute the following:
`bcdedit /set testsigning yes`
and reboot the target system.
You may have to switch off "SecureBoot" for this, which can be done in the bios.

If test signing mode is enabled, this is displayed at the bottom right of the desktop. The system now accepts all signed drivers for execution.



For TwinCAT/BSD®

In the file `/usr/local/etc/TwinCAT/3.1/TcRegistry.xml` enter "`<Value Name="EnableTestSigning" Type="DW"1</Value>`" under Key "`System`".

```
<Key Name="System">
  <Value Name="RunAsDevice" Type="DW">1</Value>
  <Value Name="RTimeMode" Type="DW">0</Value>
```

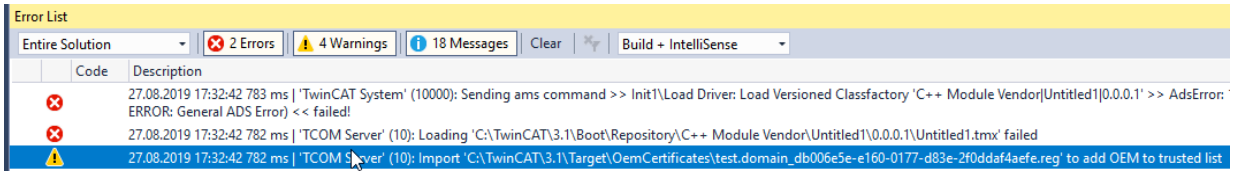
```
  <Value Name="AmsNetId" Type="BIN">052445B00101</Value>
  <Value Name="LockedMemSize" Type="DW">33554432</Value>
  <Value Name="EnableTestSigning" Type="DW">1</Value>
</Key>
```

Then restart the TwinCAT system service:

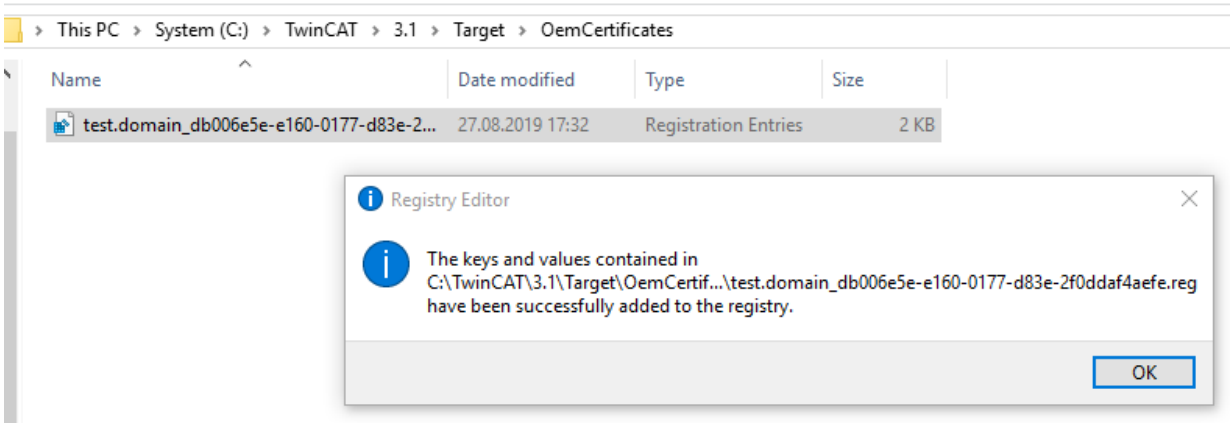
```
doas service TcSystemService restart
```

After the respective procedure, the system accepts all signed drivers for execution.

1. During the first activation (Activate Configuration) with a TwinCAT user certificate, the target system detects that the certificate is not trusted and the activation process is aborted:



For Windows:
A local user with administration rights can trust the certificate via the created REG file by simply executing it:



For TwinCAT/BSD®:
If the "Tcimportcert" package is not installed, install it: `pkg install TcImportCert`
Trust the certificate via `doas tcimportcert /usr/local/etc/TwinCAT/3.1/Target/OemCertificates/<CreatedFile>.reg`.
Then restart the TwinCAT System Service or reboot the system:
`doas service TcSystemService restart`

⇨ This process only enables C++ modules with a signature from the trusted TwinCAT user certificates to run.

2. Following this process you can use the TwinCAT user certificate for signing with the test mode of the operating system.
   This is configured in the project properties [▶ 144].
   Use the TcSignTool [▶ 55] to avoid storing the password of the TwinCAT user certificate in the project, where it would also end up in version management, for example.

If you want to use the TwinCAT user certificate without TestMode for delivery, you must have the certificate countersigned by Beckhoff [▶ 23].

5.2.1.2 User certificates for delivery without test mode

System requirements

- Min. TwinCAT 3.1 Build 4024
- Min. Windows 10 or TwinCAT/BSD® (on the target system)

With TwinCAT Build 4024, Beckhoff offers existing customers the issuing of a "TwinCAT 3 OEM user certificate", which can be used for signing TMX files created with TwinCAT 3 in C++.

- This certificate requires secure validation of the applicant data, since it is used in the Windows environment. TwinCAT 3 user certificates must therefore be officially ordered for validation of the address and contact data, and are only issued to existing Beckhoff customers.
- Order number: **TC0008**
- The issuing of this TwinCAT 3 user certificate is free of charge.
- Directory for saving the certificate: **C:\TwinCAT\3.1\CustomConfig\Certificates**

The TwinCAT 3 user certificate is not required for using the TwinCAT 3 TMX files

The TwinCAT 3 user certificate is used exclusively for the one-time signing of the TMX files and is not required for the use of the TMX files signed with it.

On which computers is the TwinCAT 3 user certificate TC0008 required?

The TwinCAT 3 user certificate should be located exclusively on the engineering computer on which the TMX files are signed - i.e. **NOT** on each target system.

Validity of the TwinCAT 3 user certificate

The validity of the TwinCAT 3 user certificate is limited to two years for security reasons.

What happens if the certificate has expired?

You can no longer sign new TMX files.
However, the use of already signed TMX files is still possible without any restrictions.

You can apply for a renewal of your certificate before the expiry of the two years (and even after that).

To extend a TwinCAT 3 user certificate, the same process applies as for requesting a new certificate. In this case, the certificate must also be ordered (the order numbers for a certificate extension are the same as for a new certificate request).

In contrast to a new certificate, you do not generate a new OEM Certificate Request File but send your existing certificate to the Beckhoff certificate section for renewal. Please notify us in the email that this is a certificate extension and not a new issue. Otherwise, the same criteria apply regarding the content of the email as for the application for a new certificate.

The existing certificate receives a new expiration date, is then re-signed and is valid for another 2 years.

The newly signed certificate is thus fully compatible with the original version.

5.2.1.2.1 Request TwinCAT 3 user certificate

Overview of the ordering and validation process

An official order is required to request a TwinCAT user certificate.

- Order number: **TC0008** (TwinCAT 3 Certificate Extended Validation)
- The issuing (and renewal) of a TwinCAT 3 user certificate is free of charge.
- Since a TwinCAT 3 user certificate is a digital ID card, verification of the inquirer's contact data is required according to the usual market standards.
- A TwinCAT 3 user certificate is therefore only issued to existing Beckhoff customers.

Overview of the ordering and validation process

Your email address must be a company email account (freemailers such as GMail or similar are not permitted) and correspond with the company name of the inquirer.

1. Contact your Beckhoff sales contact and announce the request of a TwinCAT 3 OEM certificate. Order "TC0007" or "TC0008".

2. Important: as the inquirer, please provide your contact details as the delivery address (= contact name and email address) and the area of use of the certificate (company name, address).
3. The contact details provided in the order will be verified and you (the inquirer named in the delivery address) will be contacted by Beckhoff Sales.
4. When requesting a new OEM certificate, Creation of the Certificate Request file for TC0008 [▶ 25].
5. Determine the "File Fingerprint" of the OEM certificate file using TwinCAT Engineering (see Determining the file fingerprint of the OEM certificate file [▶ 28]). Please inform the Beckhoff sales contact of this File Fingerprint as part of your contact data verification. The transmission of the File Fingerprint must be done by a different communication channel than the one used for sending the OEM certificate request file.
6. Now send the "OEM certificate file" to the Beckhoff sales contact.
7. After signing the certificate file at the Beckhoff headquarters, you will receive it by e-mail from your contact person.

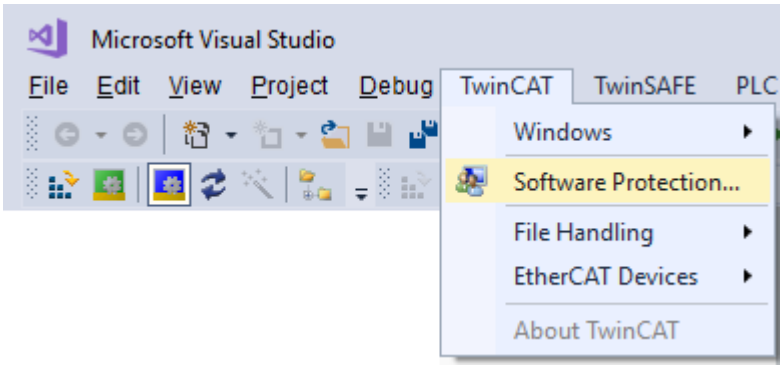
Please note that it may take a few days to validate your contact details and issue the certificate.

5.2.1.2.2 Creation of the Certificate Request file for TC0008

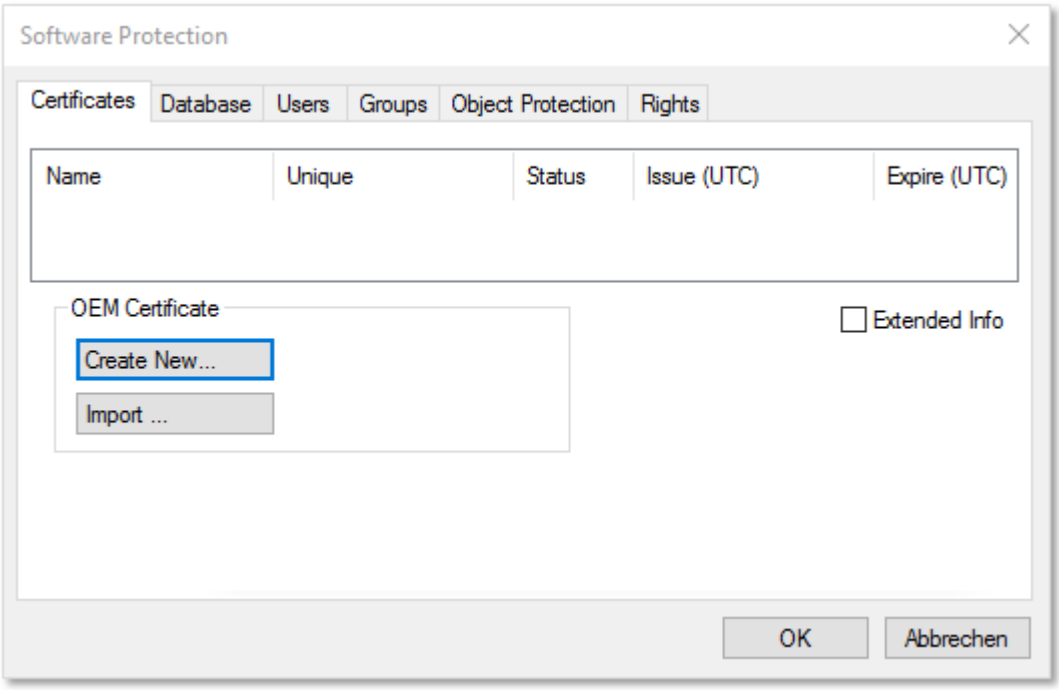
System requirements

- Min. TwinCAT 3.1 Build 4024
- Min. Windows 10 or TwinCAT/BSD® (on the target system)

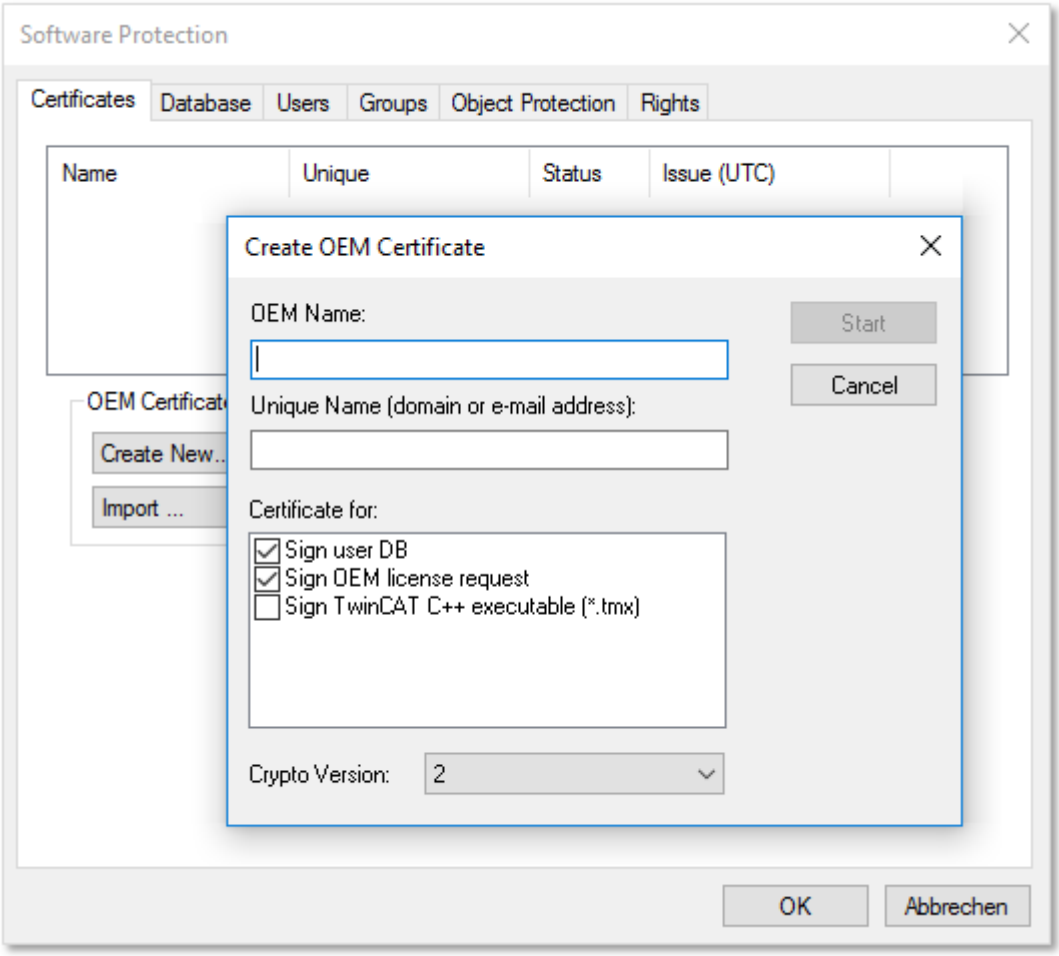
1. Call up the Software Protection configurator. To do this, select the menu item **Software Protection** in the main menu below the item **TwinCAT**:



2. In the window that opens, select the **Certificates** tab.
   Click **Create New....**:

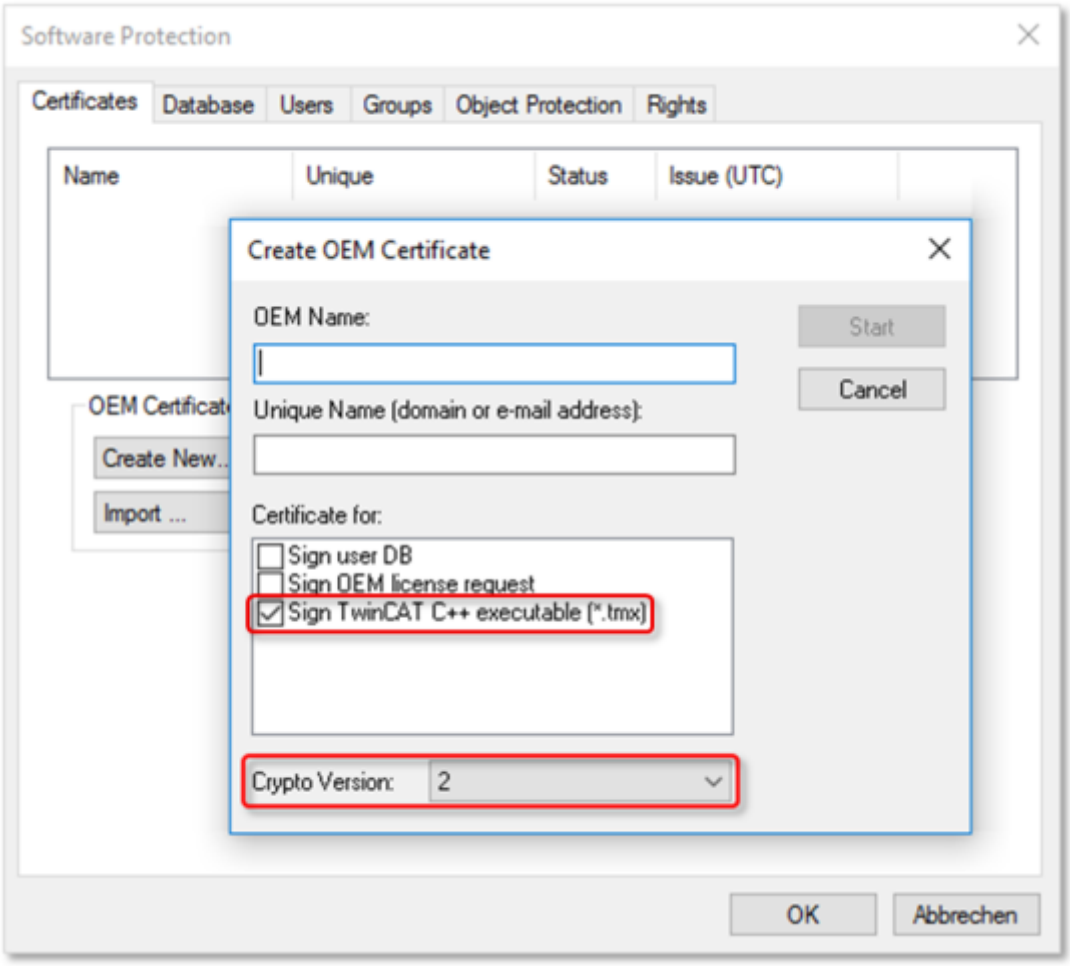


3. The **Create OEM Certificate** input window opens:



4. Enter the required data:
   - Enter your company name in the **OEM Name** text field. The name must have a clear reference to your company or your business unit.

- Enter a **Unique Name**. The "OEM Unique Name" must be a unique name that uniquely identifies the owner of the certificate worldwide, preferably the URL of your company's website or your email address. The email address must be a company email address, i.e. it must be possible to assign it unambiguously to your company.
- Check the checkbox **Sign TwinCAT C++ executables**:

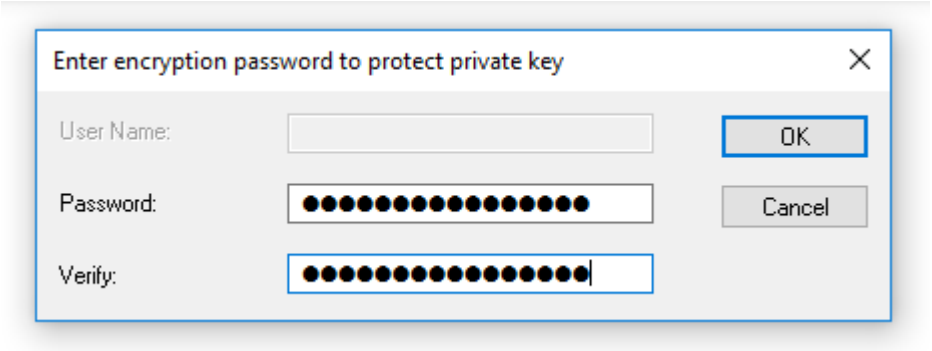


If you only want to sign TwinCAT driver software with this certificate, uncheck the other two checkboxes. (These are only used in the PLC area)

- Make sure that Crypto version 2 (for the encrypted content of the certificate content) is set. (default setting)

5. Once you have entered the data, click **Start** and select a directory to save the file.
   You can simply accept the suggested directory "c:\twincat\3.1\customconfig\certificates". You need the newly created file in this directory in order to be able to read out the "File Fingerprint" for this file [▶ 28] **in a subsequent step.**
   ⇨ A dialog for selecting a password for the OEM Private Key opens.
6. Issue a password for the OEM Private Key**.**
   Password security - Be sure to use a strong password for your certificate! Take suitable measures to protect your password so that it cannot fall into the wrong hands!
   Password cannot be recovered if lost - Beckhoff cannot recover or reset your password. If you forget or lose the password for your certificate, you can no longer use it and have to request a new certificate.

7. Confirm the password by entering it again and close the dialog with **OK**.



⇨ The file is saved.

The "Certificate Request File" generated in this way must now be signed by the Beckhoff certificate section in order to be valid. The procedure is described in chapter Requesting a certificate [▶ 24].

5.2.1.2.3 Determining the file fingerprint of the OEM certificate file

You need this functionality to request a **TwinCAT OEM Certificate Extended Validation** (TC0008).

System requirements

This functionality requires TwinCAT 3.1 Build 4024 of higher.

The OEM Certificate Request File becomes the TwinCAT OEM certificate once it is signed by Beckhoff. The files do not differ except for this signature. For this reason, the term "TwinCAT OEM certificate file" is used for both file versions in the following sections.

Reading the "file fingerprint" of an OEM certificate file via TwinCAT 3 Engineering

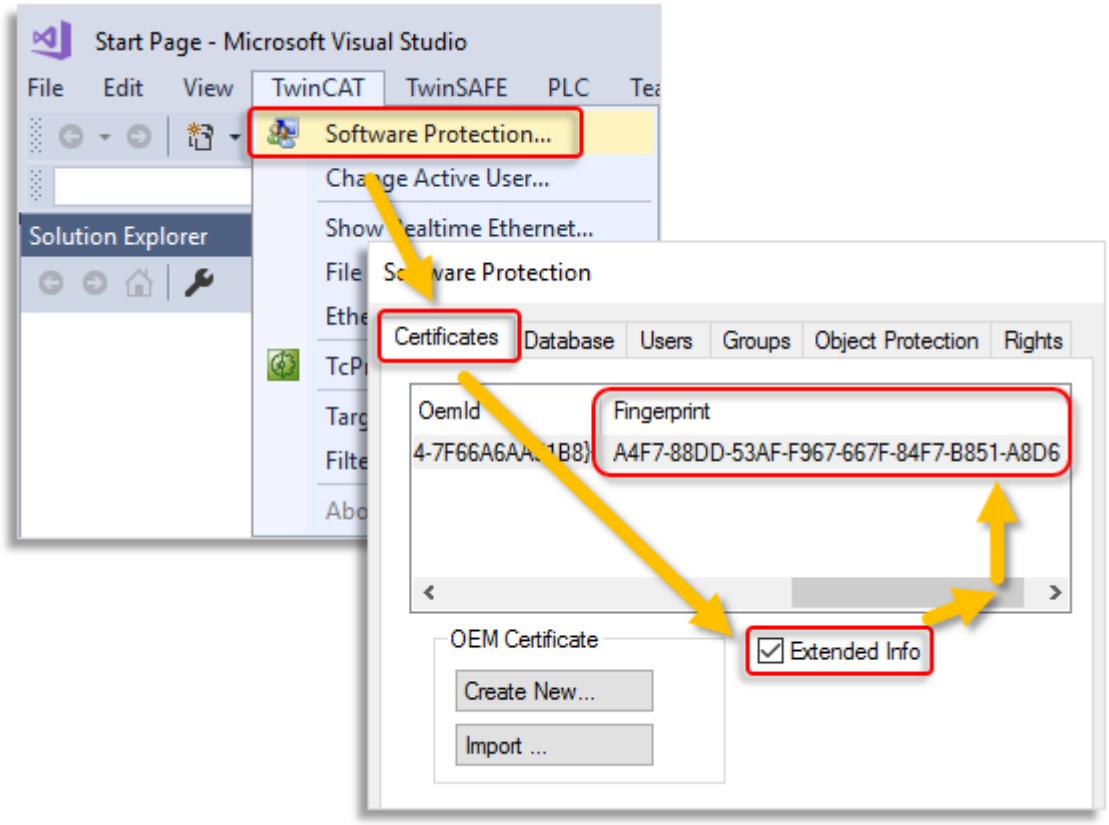
For this function it is necessary that the OEM certificate file is located in this directory: "c:\twincat\3.1\customconfig\certificates".

This directory contains your OEM certificate, if you already have a certificate and want to renew it.

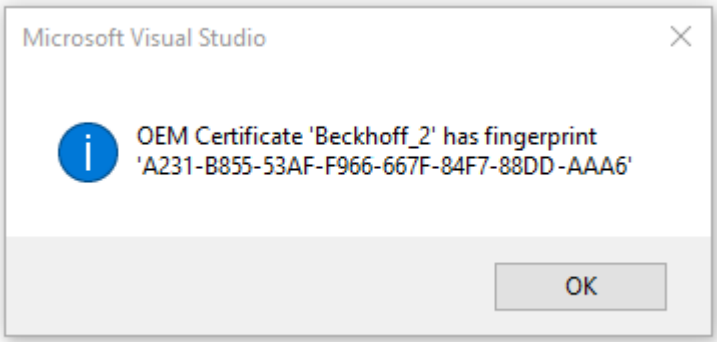
If you did not change the suggested directory when creating the "OEM Certificate Request File", the file is already in this directory.

Procedure:

1. Call up the TwinCAT 3 Software Protection configurator.



2. Select the **Certificates** tab.
3. Check the **Extended Info** checkbox.
4. In the window scroll to the right until you see the **Fingerprint** column. (As an alternative, you can simply double-click the certificate line. The Fingerprint file is then displayed in a pop-up window:



The shortcut [Ctrl] + [C] can be used to copy the fingerprint data from the message window to the Windows clipboard.

5.2.1.2.4 Saving the signed TwinCAT user certificate

Recommended directory for saving the certificate: **C:\TwinCAT\3.1\CustomConfig\Certificates**

System requirements

- Min. TwinCAT 3.1 Build 4024
- Min. Windows 10 or TwinCAT/BSD® (on the target system)

The TwinCAT 3 user certificate is not required for using the TwinCAT 3 TMX files

The TwinCAT 3 user certificate is used exclusively for the one-time signing of the TMX files and is not required for the use of the TMX files signed with it.

On which computers is the TwinCAT 3 user certificate TC0008 required?

The TwinCAT 3 user certificate should be located exclusively on the engineering computer on which the TMX files are signed - i.e. **NOT** on each target system.

5.2.2 Operating system

NOTICE

Migration to TMX with TwinCAT Loader recommended

Since TwinCAT 3.1 4024.0 versioned C++ projects are available, whose binaries can be loaded directly from TwinCAT. Migration is recommended!

For the implementation of TwinCAT 3 C++ modules on x64 platforms, the driver (*.sys file) must be signed with a certificate if it is to be loaded by the operating system.

The signature, which is automatically executed during the TwinCAT 3 build process, is used by 64-bit Windows operating systems for the authentication of the drivers.

A certificate is required to sign a driver. This Microsoft documentation describes the process and background knowledge for obtaining a test and release certificate that is accepted by 64-bit Windows operating systems.

To use such a certificate in TwinCAT 3, configure the step after compiling your x64 build target as documented in "Creating a test certificate for test mode [▶ 30]".

Test certificates

For testing purposes, self-signed test certificates can be created and used without technical limitations.

The following tutorials describe how to enable this option.
To create drivers with real certificates for production machines, this option must be disabled.

- Creating a test certificate for test mode [▶ 30]
- Delete (test) certificates [▶ 32]

Further references:

MSDN, MakeCert test certificates (Windows drivers),
https://docs.microsoft.com/en-us/windows-hardware/drivers/install/makecert-test-certificate

5.2.2.1 Test signing

Overview

Implementing TwinCAT 3 C++ modules for x64 platforms requires signing the driver with a certificate.

This article describes how to create and install a test certificate for testing a C++ driver.

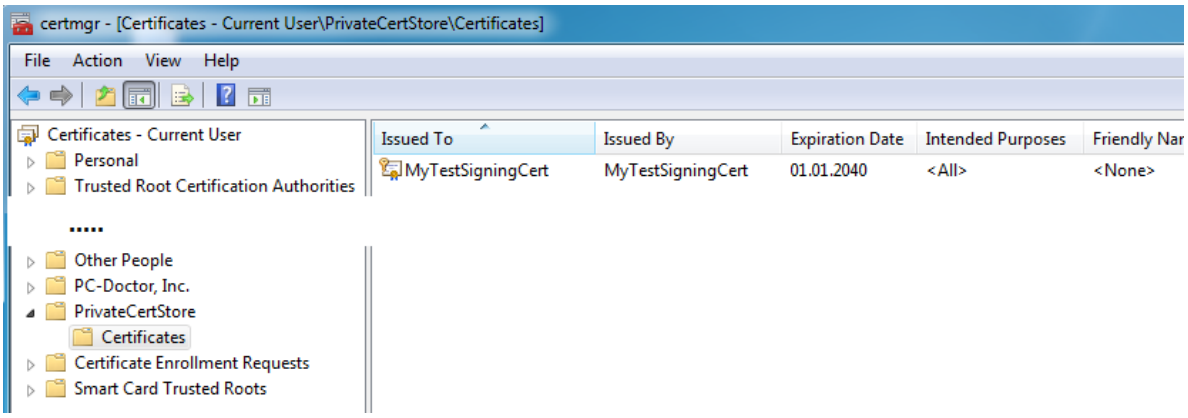
Note the procedure when creating test certificates

Developers may have a wide range of tools for creating certificates themselves. Please follow this description exactly to activate the test certificate mechanism.

The following commands must be executed from a command line that has been opened in either way:

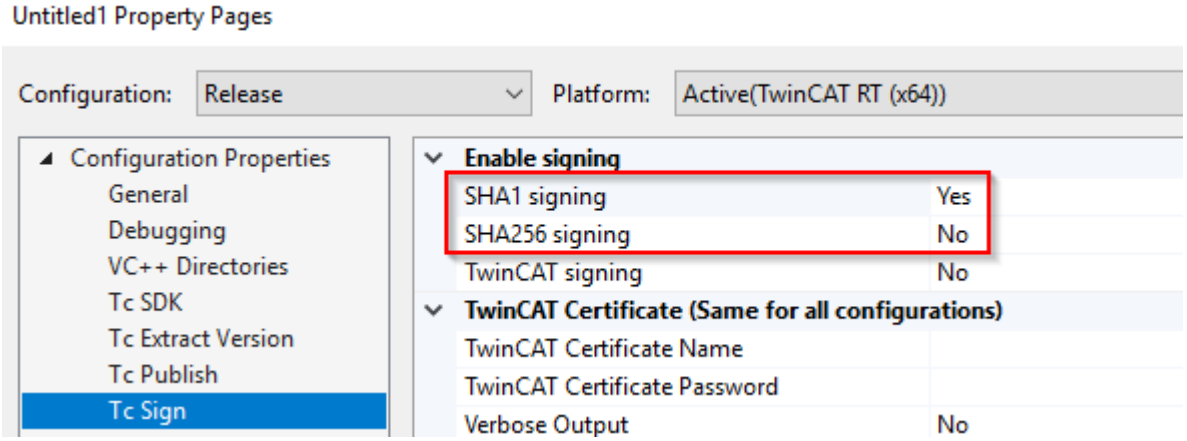
- **Command prompt from Microsoft Visual Studio 2010 / 2012 with administrator rights**. (Via: **All Programs** -> **Microsoft Visual Studio 2010/2012** -> **Visual Studio Tools** -> Visual Studio Command Prompt, then right-click **Run as administrator**)
- **Developer Command Prompt from Microsoft Visual Studio 2017 / 2019 with administrator rights.**. (Via: **All Programs** -> **Visual Studio 2017** -> Visual Studio Command Prompt for VS 2017/2019, then right-click on **Run as administrator**)
- Only if the WINDDK has been installed:
  Normal prompt (**Start** ->Command Prompt) with administrator rights, then change to directory *%WINDDK7%\bin\x86*, which contains the corresponding tools.

1. On XAE:
   In the engineering system, enter the following command in the Microsoft Visual Studio 2010 / 2012 command prompt with administrator rights (see note above):
   ```
   makecert -r -pe -ss PrivateCertStore -n CN=MyTestSigningCert
   MyTestSigningCert.cer
   ```
   (If you do not have access rights to the PrivateCertStore, you can use a different storage location. This must also be used in the PostBuild event [▶ 34]**.)**
   ⇨ This is followed by creation of a self-signed certificate, which is stored in the file "MyTestSigningCert.cer" and in the Windows Certificate Store.
   ⇨ Check the result with mmc (**Use File->Add/Remove Snap-in->Certificates**):



2. On XAE:
   configure the certificate so that it is recognized by TwinCAT XAE on the engineering system.
   Set the environment variable TWINCATTESTCERTIFICATE to "MyTestSigningCert" in the engineering system or edit the post build event of Debug|TwinCAT RT (x64) and Release|TwinCAT RT (x64).
   The name of the variable is NOT the name of the certificate file, but the CN name (in this case MyTestSigningCert).

From TwinCAT 3.1 4024.0, the configuration of the certificate to be used is carried out under Tc Sign in the project properties. To use signing via the operating system, as described here, please pay attention to the project settings.

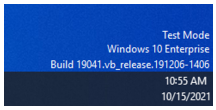


On XAR (and XAE, if it is a local test), activate the test mode so that the operating system can accept the self-signed certificates. This can be done on both engineering systems (XAE) and runtime systems (XAR).

For Windows

Use the administrator prompt to execute the following:
`bcdedit /set testsigning yes`
and reboot the target system.
You may have to switch off "SecureBoot" for this, which can be done in the bios.

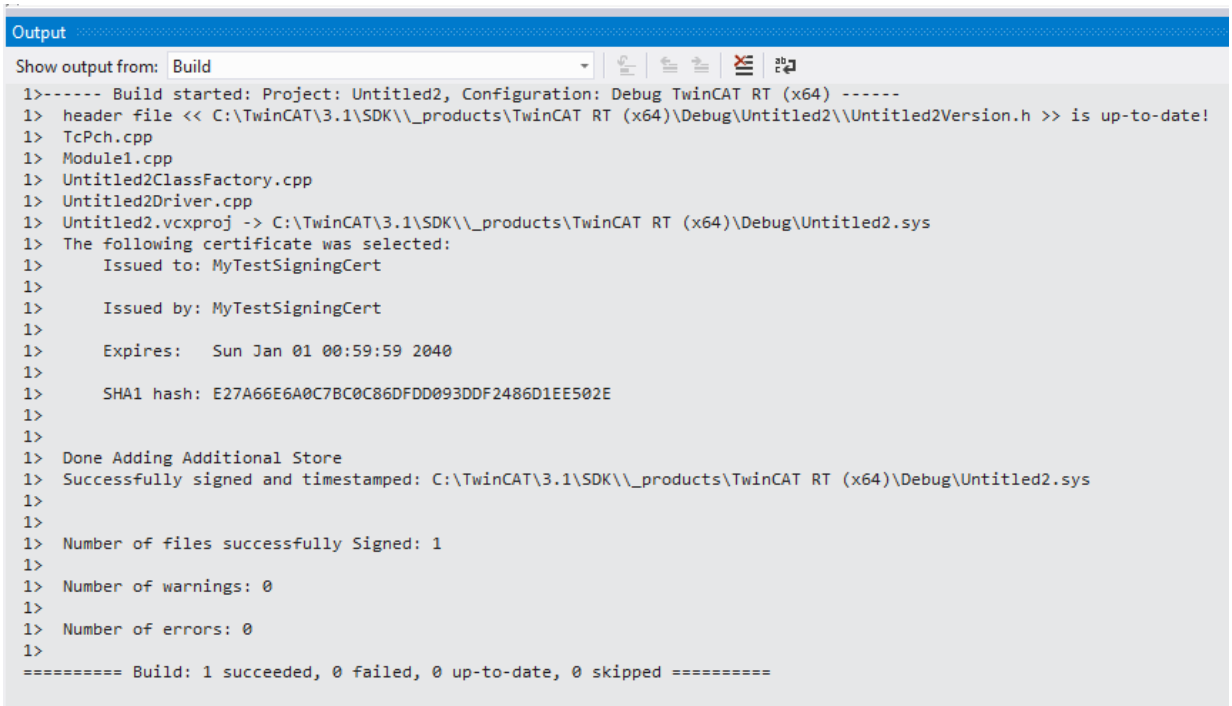
If test signing mode is enabled, this is displayed at the bottom right of the desktop. The system now accepts all signed drivers for execution.



After the respective procedure, the system accepts all signed drivers for execution.

1. Test whether a configuration with a TwinCAT module implemented in a TwinCAT C++ driver can be enabled and started on the target system.

⇨ Compilation of the x64 driver generates the following output:



References:
MSDN, MakeCert test certificates (Windows drivers),
https://docs.microsoft.com/en-us/windows-hardware/drivers/install/makecert-test-certificate

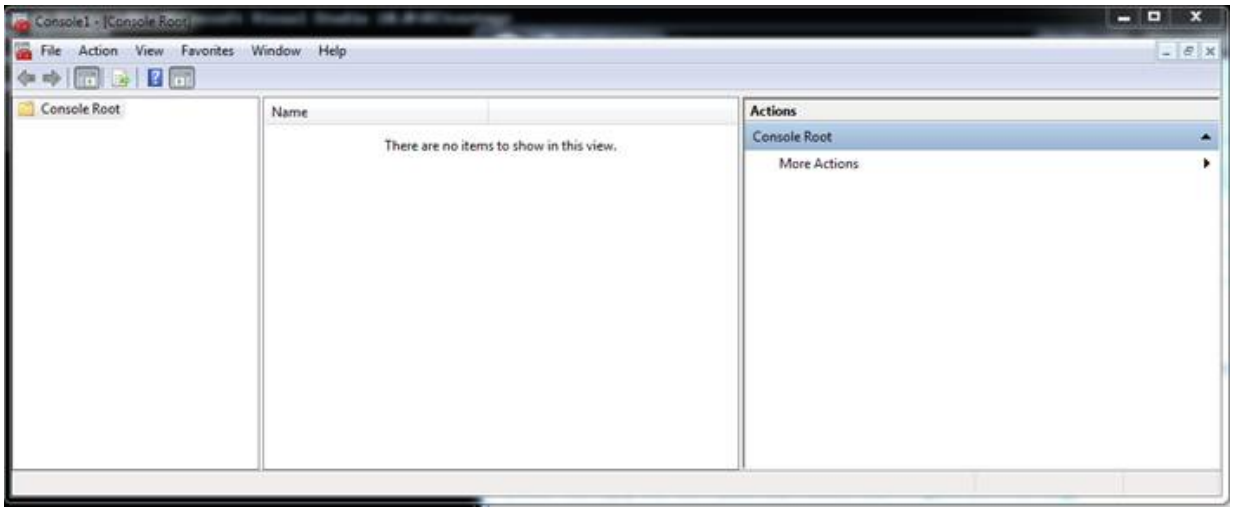
5.2.2.2 Delete test certificate

This article is about how to delete a test certificate.

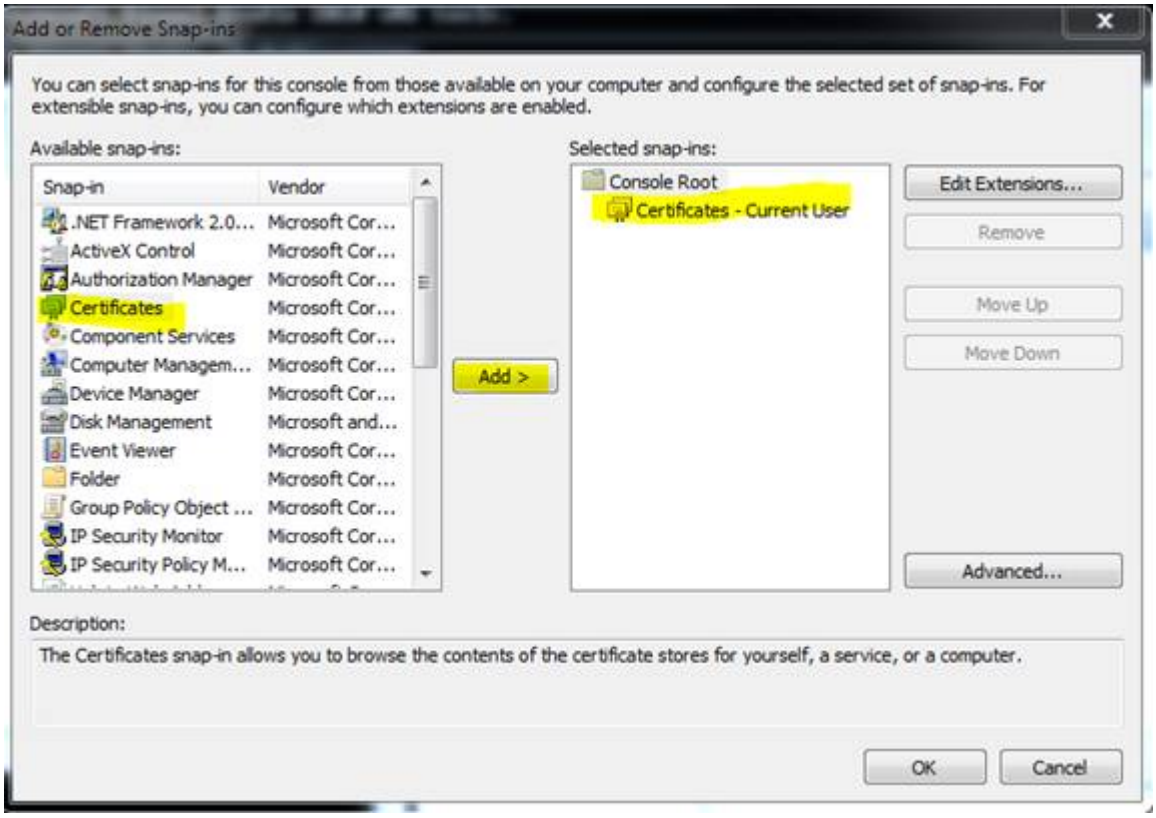
Overview

A certificate can be deleted with the Microsoft Management Console:

1. Start the management console MMC.exe via the Start menu or the user interface.

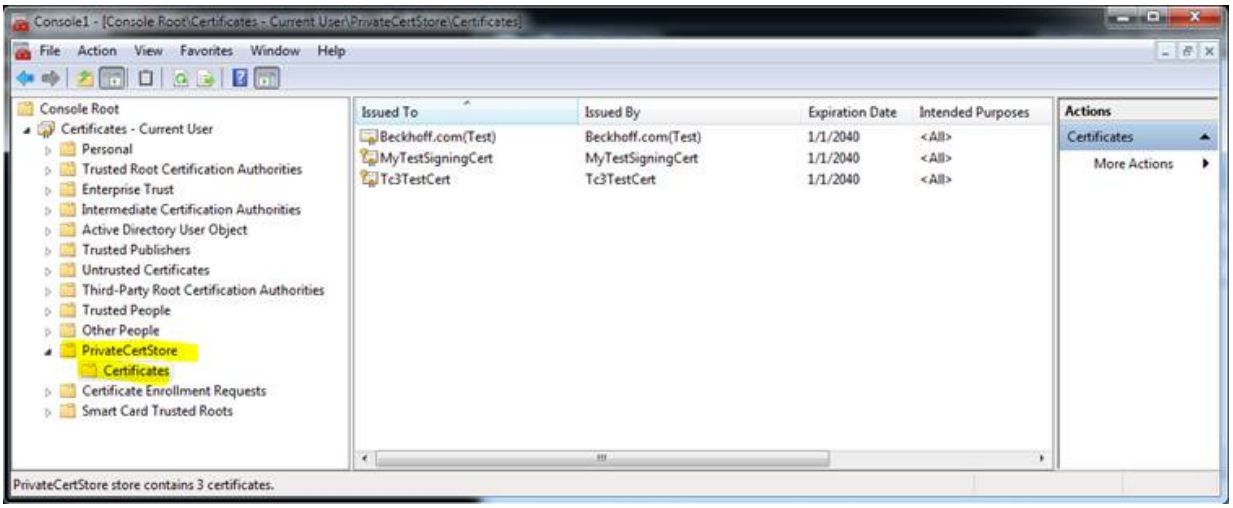


2. Click in the menu on **File** -> **Add/Remove Snap-in..** and select the certificate snap-in for the current user; conclude with **OK**.



⇨ The certificates are listed in the node under **PrivateCertStore/Certificates**.

3. Select the certificate to be deleted.



5.2.2.3 Windows driver without test mode

For Windows operating systems the driver has to be signed via "Attestation Signing". This requires an EV certificate.
Microsoft provides relevant instructions: https://docs.microsoft.com/en-us/windows-hardware/drivers/dashboard/attestation-signing-a-kernel-driver-for-public-release
he drivers created in this way are also suitable for devices that have secure boot enabled.

Microsoft announced that the previous procedure using CrossSigning certificates (signing tool with parameter /ac) will be discontinued from July 2021. It can no longer be used (depending on the expiry date of the individual CrossSigning certificate).
Versioned C++ projects, which are loaded via the TwinCAT Loader [▶ 50], have been available for TwinCAT C++ for some time, so that they are not drivers in the sense of the operating system. Beckhoff therefore recommends using versioned C++ projects.

A guide is available in the How-to section for the migration of TwinCAT C++ drivers to versioned C++ projects.

6 Modules

The TwinCAT module concept is one of the core elements for the modularization of modern machines. This chapter describes the modular concept and working with modules.

The modular concept applies to all TwinCAT modules, not just C++ modules, although most details only relate to the engineering of C++ modules.

6.1 The TwinCAT Component Object Model (TcCOM) concept

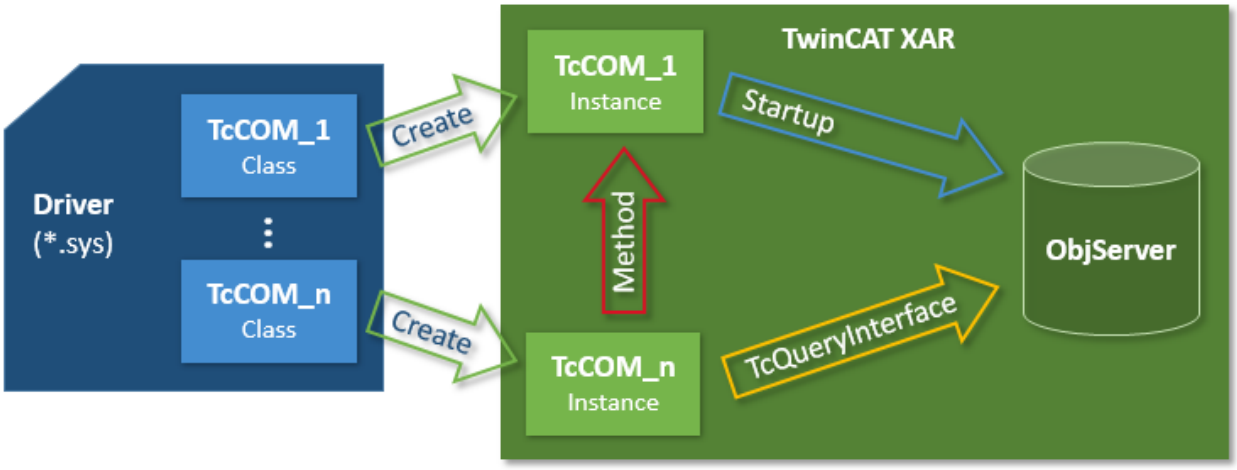
The TwinCAT Component Object Model defines the characteristics and the behavior of the modules. The model derived from the "Component Object Model" COM from Microsoft Windows describes the way in which various independently developed and compiled software components can co-operate with one another. To make that possible, a precisely defined mode of behavior and the observation of interfaces of the module must be defined, so that they can interact. Such an interface is also ideal for facilitating interaction between modules from different manufacturers, for example.

To some degree TcCOM is based on COM (Component Object Model of the Microsoft Windows world), although only a subset of COM is used. In comparison with COM, however, TcCOM contains additional definitions that go beyond COM, for example the state machine module.

Overview and application of TcCOM modules

This introductory overview is intended to make the individual topics easier to understand.

One or several TcCOM modules are consolidated in a driver. This driver is created by TwinCAT Engineering using the MSVC compiler. The modules and interfaces are described in a TMC (TwinCAT Module Class) file. The drivers and their TMC file can now be exchanged and combined between the engineering systems.



Instances of these modules are now created using the engineering facility. They are associated with a TMI file. The instances can be parameterized and linked with each other and with other modules to form the I/O. A corresponding configuration is transferred to the target system, where it is executed.

Corresponding modules are started, which register with the TwinCAT ObjectServer. The TwinCAT XAR also provides the process images. Modules can query the TwinCAT ObjectServer for a reference to another object with regard to a particular interface. If such a reference is available, the interface methods can be called on the module instance.

The following sections substantiate the individual topics.

ID Management

Different types of ID are used for the interaction of the modules with each other and also within the modules. TcCOM uses GUIDs (128-bit) and 32-bit long integers.

TcCOM uses

- GUIDs for: ModulIDs, ClassIDs and InterfaceIDs.
- 32-bit long integers are used for: ParameterIDs, ObjectIDs, ContextIDs, CategoryID.

Interfaces

An important component of COM, and therefore of TcCOM too, are interfaces.

Interfaces define a set of methods that are combined to perform a certain task. An interface is referenced with a unique ID (InterfaceID), which must never be modified as long as the interface does not change. This ID enables modules to determine whether they can cooperate with other modules. At the same time the development process can take place independently, if the interfaces are clearly defined. Modifications of interfaces therefore lead to different IDs. The TcCOM concept is designed such that InterfaceIDs can superpose other (older) InterfaceIDs ("Hides" in the TMC description / TMC editor). In this way, both versions of the interface are available, while on the other hand it is always clear which is the latest InterfaceID. The same concept also exists for the data types.

TcCOM itself already defines a whole series of interfaces that are prescribed in some cases (e.g. ITComObject), but are optional in most. Many interfaces only make sense in certain application areas. Other interfaces are so general that they can often be re-used. Provision is made for customer-defined interfaces, so that two third-party modules can interact with each other, for example.

- All interfaces are derived from the basic interface ItcUnknown which, like the corresponding interface of COM, provides the basic services for querying other interfaces of the module (TcQueryInterface) and for controlling the service life of the module (TcAddRef and TcRelease).
- The ITComObject interface, which must be implemented by each module, contains methods for accessing the name, ObjectID, ObjectID of the parent, parameters and state machine of the module.

Several general interfaces are used by many modules:

- ITcCyclic is implemented by modules, which are called cyclically ("CycleUpdate"). The module can register via the ITcCyclicCaller interface of a TwinCAT task to obtain cyclic calls.
- The ITcADI interface can be used to access data areas of a module.
- ITcWatchSource is implemented by default; it facilitates ADS DeviceNotifications and other features.
- The ITcTask interface, which is implemented by the tasks of the real-time system, provides information about the cycle time, the priority and other task information.
- The ITComObjectServer interface is implemented by the ObjectServer and referenced by all modules.

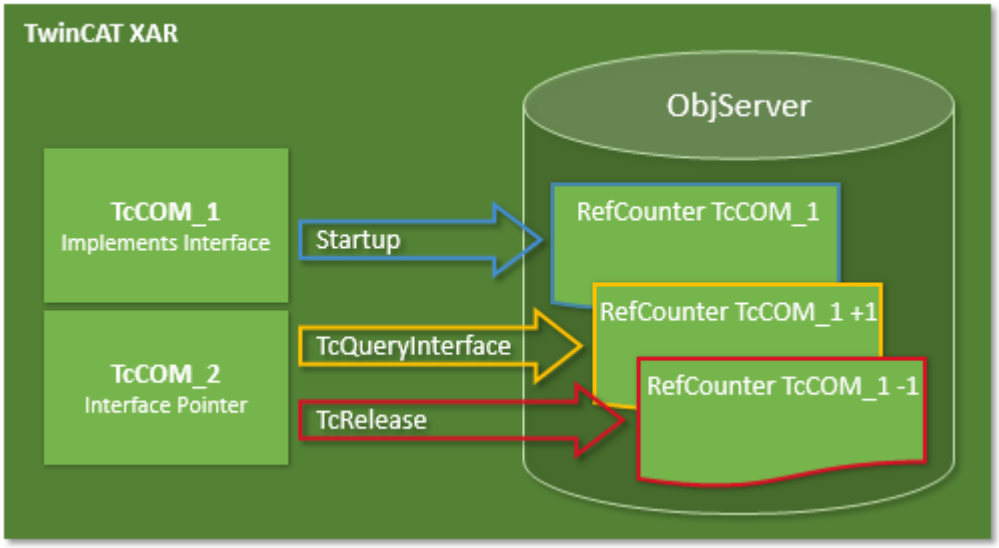
A whole series of general interfaces has already been defined. General interfaces have the advantage that their use supports the exchange and recycling of modules. Only if no suitable general interfaces exist should you define your own interfaces.

Class Factories

"Class Factories" are used for creating modules in C++. All modules contained in a driver have a common Class Factory. The Class Factory registers once with the ObjectServer and offers its services for the creation of certain module classes. The module classes are identified by the unique ClassID of the module. When the ObjectServer requests a new module (based on the initialization data of the configurator or through other modules at runtime), the module selects the right Class Factory based on the ClassID and triggers creation of the module via its ITcClassFactory interface.

Module service life

Similar to COM, the service life of a module is determined via a reference counter (RefCounter). The reference counter is incremented whenever a module interface is queried. The counter is decremented when the interface is released. An interface is also queried when a module logs into the ObjectServer (the ITComObject interface), so that the reference counter is at least 1. The counter is decremented on logout. When the counter reaches 0, the module deletes itself automatically, usually after logout from the ObjectServer. If another module already maintains a reference (has an interface pointer), the module continues to exist, and the interface pointer remains valid, until this pointer is released.



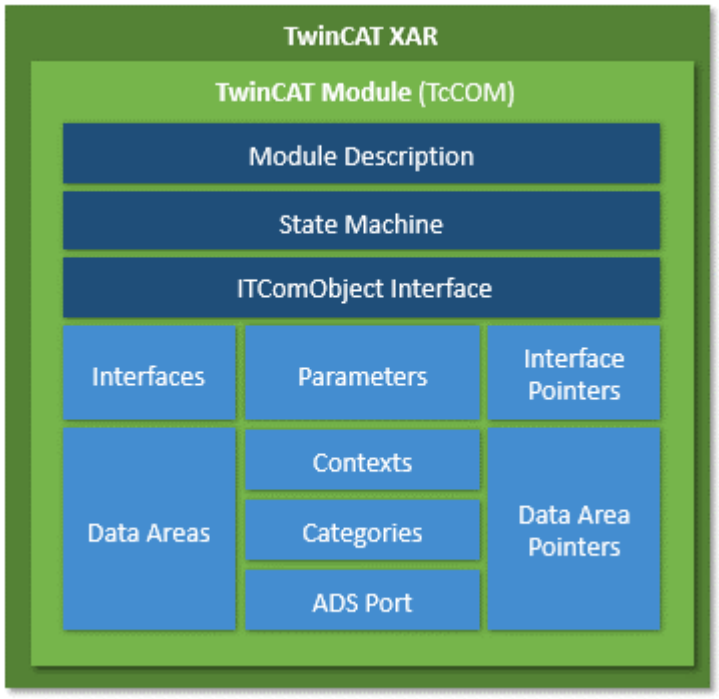
6.1.1 TwinCAT module properties

A TcCOM module has a number of formally defined, prescribed and optional properties. The properties are sufficiently formalized to enable interchangeable application. Each module has a module description, which describes the module properties. They are used for configuring the modules and their relationships with each other.

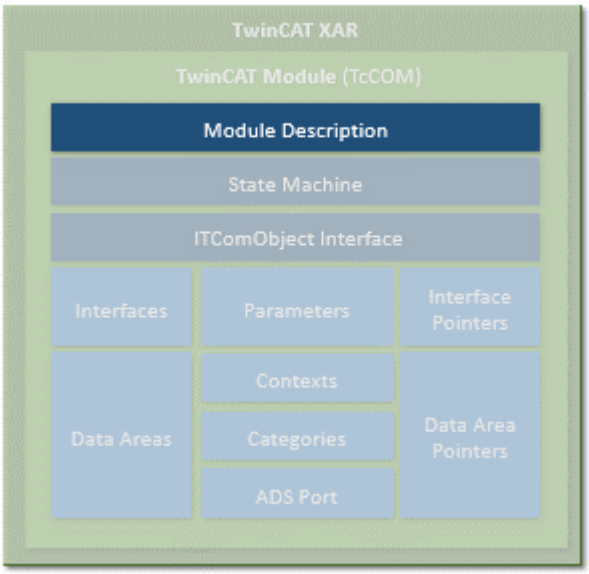
If a module is instantiated in the TwinCAT runtime, it registers itself with a central system instance, the ObjectServer. This makes it reachable and parameterizable for other modules and also for general tools. Modules can be compiled independently and can therefore also be developed, tested and updated independently. Modules can be very simple, e.g. they may only contain a basic function such as low-pass filter. Or they may be very complex internally and contain the whole control system for a machine subassembly.

There are a great many applications for modules; all tasks of an automation system can be specified in modules. Accordingly, no distinction is made between modules, which primarily represent the basic functions of an automation system, such as real-time tasks, fieldbus drivers or a PLC runtime system, and user- or application-specific algorithms for controlling a machine unit.

The diagram below shows a common TwinCAT module with his main properties. The dark blue blocks define prescribed properties, the light blue blocks optional properties.



Module description



Each TcCOM module has some general description parameters. These include a ClassID, which unambiguously references the module class. It is instantiated by the corresponding ClassFactory. Each module instance has an ObjectID, which is unique in the TwinCAT runtime. In addition there is a parent ObjectID, which refers to a possible logical parent.

The description, state machine and parameters of the module described below can be reached via the ITComObject interface (see "Interfaces").

Class description files (*.tmc)

The module classes are described in class description files (TwinCAT Module Class; *.tmc).

These files are used by developers to describe the module properties and interfaces, so that others can use and embed the module. In addition to general information (vendor data, module class ID etc.), optional module properties are described.

- Supported categories
- Implemented interfaces
- Data areas with corresponding symbols
- Parameter
- Interface pointers
- Data pointers, which can be set

The system configurator uses the class description files mainly as a basis for the integration of a module instance in the configuration, for specifying the parameters and for configuring the links with other modules.

They also include the description of all data types in the modules, which are then adopted by the configurator in its general data type system. In this way, all interfaces of the TMC descriptions present in the system can be used by all modules.

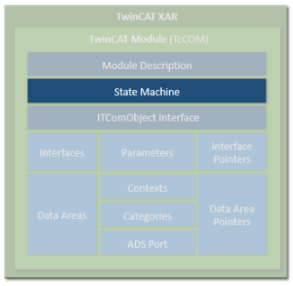
More complex configurations involving several modules can also be described in the class description files, which are preconfigured and linked for a specific application. Accordingly, a module for a complex machine unit, which internally consists of a number of submodules, can be defined and preconfigured as an entity during the development phase.

Instance description files (*.tmi)

An instance of a certain module is described in the instance description file (TwinCAT Module Instance; *.tmi). The instance descriptions are based on a similar format, although in contrast to the class description files they already contain concrete specifications for the parameters, interface pointers etc. for the special module instance within a project.

The instance description files are created by TwinCAT Engineering (XAE), when an instance of a class description is created for a specific project. They are mainly used for the exchange of data between all tools involved in the configuration. However, the instance descriptions can also be used cross-project, for example if a specially parameterized module is to be used again in a new project.

State machine

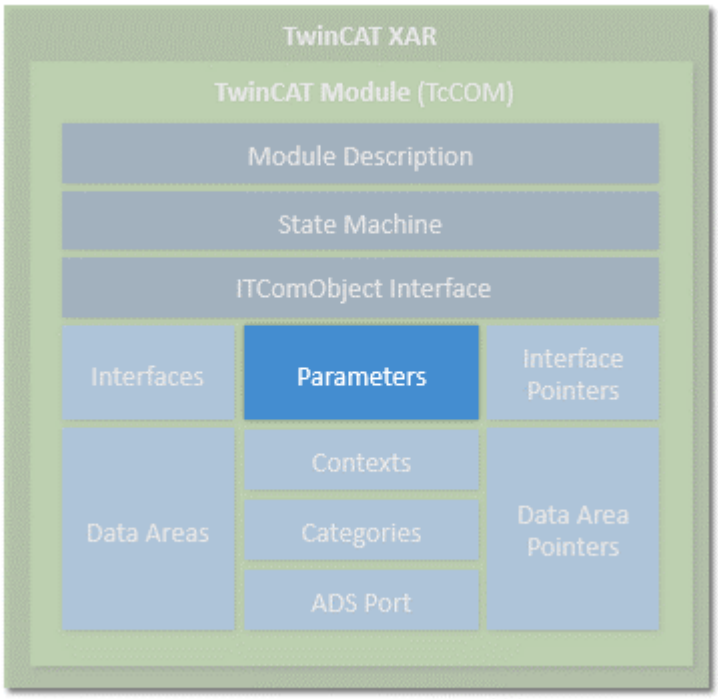


Each module contains a state machine, which describes the initialization state of the module and the means with which this state can be modified from outside. The state machine describes the states, which occur during starting and stopping of the module. This relates to module creation, parameterization and production in conjunction with the other modules.

Application-specific states (e.g. of the fieldbus or driver) can be described in their own state machines. The state machine of the TcCOM modules defines the states INIT, PREOP, SAFEOP and OP. Although the state designations are the same as under EtherCAT fieldbus, the actual states differ. When the TcCOM module implements a fieldbus driver for EtherCAT, it has two state machines (module and fieldbus state machine), which are passed through sequentially. The module state machine must have reached the operating state (OP) before the fieldbus state machine can start.

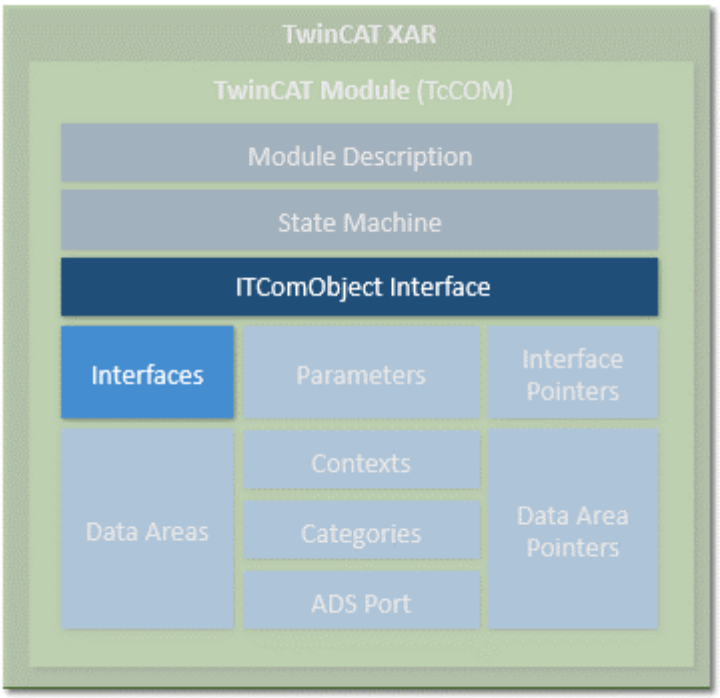
The state machine is described [▶ 44] in detail separately.

Parameter



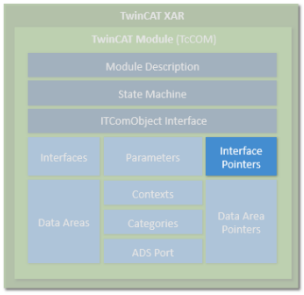
Modules can have parameters, which can be read or written during initialization or later at runtime (OP state). Each parameter is designated by a parameter ID. The uniqueness of the parameter ID can be global, limited global or module-specific. Further details can be found in the "ID Management" section. In addition to the parameter ID, the parameter contains the current data; the data type depends on the parameter and is defined unambiguously for the respective parameter ID.

Interfaces



Interfaces consist of a defined set of methods (functions), which offer modules through which they can be contacted by other modules. Interfaces are characterized by a unique ID, as described above. A module must support at least the ITComObject interface and may in addition contain as many interfaces as required. An interface reference can be queried by calling the method "TcQueryInterface" with specification of the corresponding interface ID.

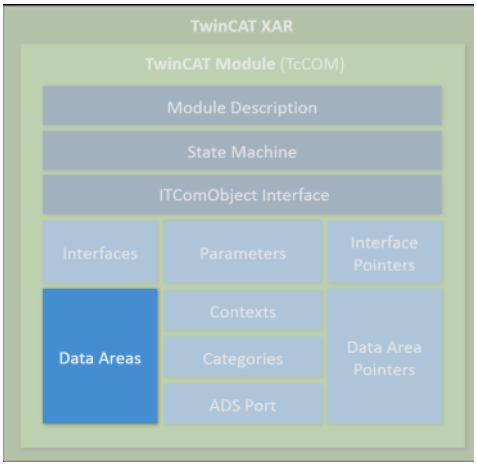
Interface pointers



Interface pointers behave like the counterpart of interfaces. If a module wants to use an interface of another module, it must have an interface pointer of the corresponding interface type and ensure that it points to the other module. The methods of the other module can then be used.

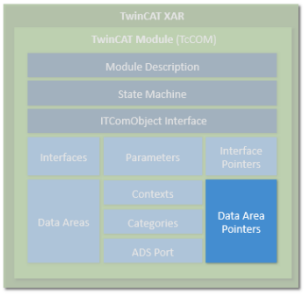
Interface pointers are usually set on startup of the state machine. During the transition from INIT to PREOP (IP), the module receives the object ID of the other modules with the corresponding interface; during the transition from PREOP to SAFEOP (PS) or SAFEOP to OP (SO), the instance of the other modules is searched with the ObjectServer, and the corresponding interface is set with the Method Query interface. During the state transition in the opposite direction, i.e. from SAFEOP to PREOP (SP) or OP to SAFEOP (OS), the interface must be enabled again.

Data areas



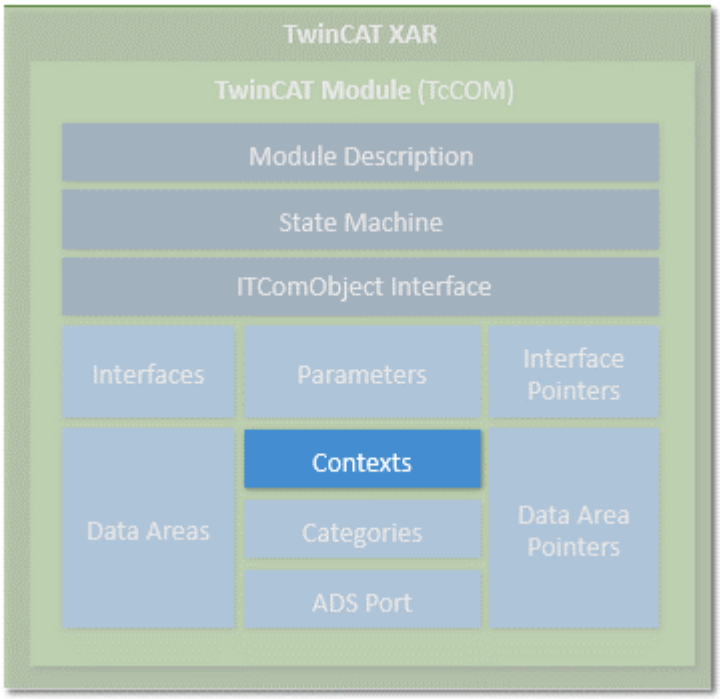
Modules can contain data areas, which can be used by the environment (e.g. by other modules or the IO area of TwinCAT). These data areas can contain any data. They are often used for process image data (inputs and outputs). The structure of the data areas is defined in the device description of the module. If a module has data areas, which it wants to make accessible for other modules, it implements the ITcADI interface to enable access to the data. Data areas can contain symbol information, which describes the structure of the respective data area in more detail.

Data area pointer



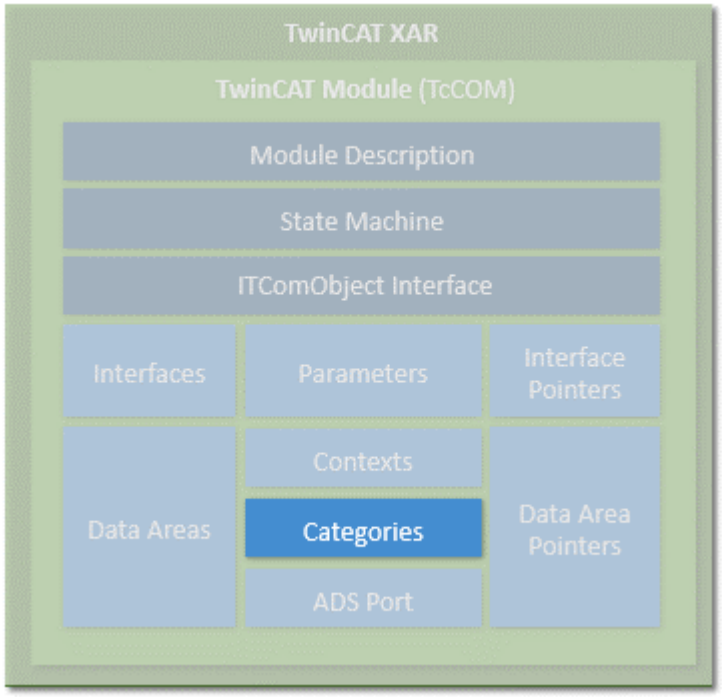
If a module wants to access the data area of other modules, it can contain data area pointers. These are normally set during initialization of the state machine to data areas or data area sections of other modules. The access is directly to the memory area, so that corresponding protection mechanisms for competing access operations have to be implemented, if necessary. In many cases it is preferable to use a corresponding interface.

Context



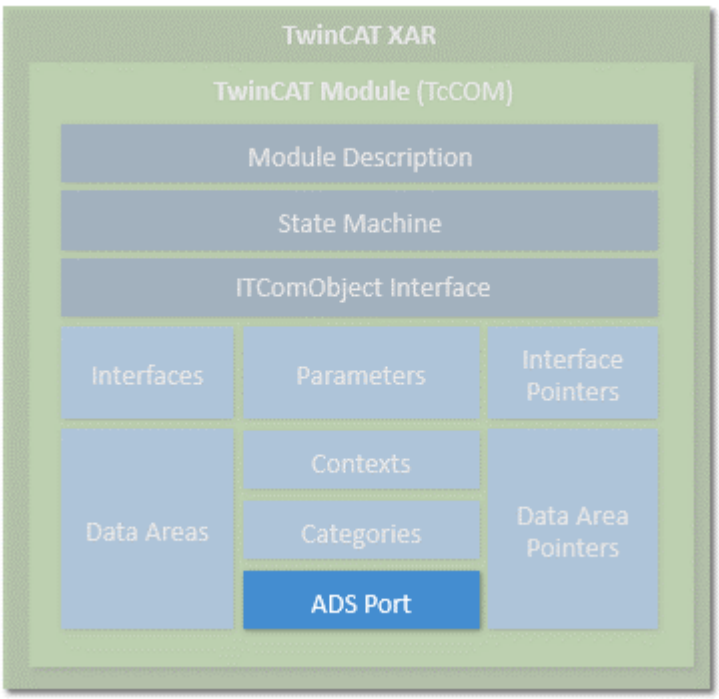
The context should be regarded as real-time task context. Context is required for the configuration of the modules, for example. Simple modules usually operate in a single time context, which therefore requires no detailed specification. Other modules may partly be active in several contexts (e.g. an EtherCAT master can support several independent real-time tasks, or a control loop can process control loops of the layer below in another cycle time). If a module has more than one time-dependent context, this must be specified the in the module description.

Categories



Modules can offer categories by implementing the interface ITComObjectCategory. Categories are enumerated by the ObjectServer, and objects, which use this to associated themselves with categories, can be queried by the ObjectServer (ITComObjectEnumPtr).

ADS



Each module that is entered in the ObjectServer can be reached via ADS. The ObjectServer uses the ITComObject interface of the modules in order to read or write parameters or to access the state machine, for example. In addition, a dedicated ADS port can be implemented, through which dedicated ADS commands can be received.

System module

In addition, the TwinCAT runtime provides a number of system modules, which make the basic runtime services available for other modules. These system modules have a fixed, constant ObjectID, through which the other modules can access it. An example for such a system module is the real-time system, which makes the basic real-time system services, i.e. generation of real-time tasks, available via the ITcRTime interface. The ADS router is also implemented as a system module, so that other modules can register their ADS port here.

Creation of modules

Modules can be created both in C++ and in IEC 61131-3. The object-oriented extensions of the TwinCAT PLC are used for this purpose. Modules from both worlds can interact via interfaces in the same way as pure C++ modules. The object-oriented extension makes the same interfaces available as in C++.

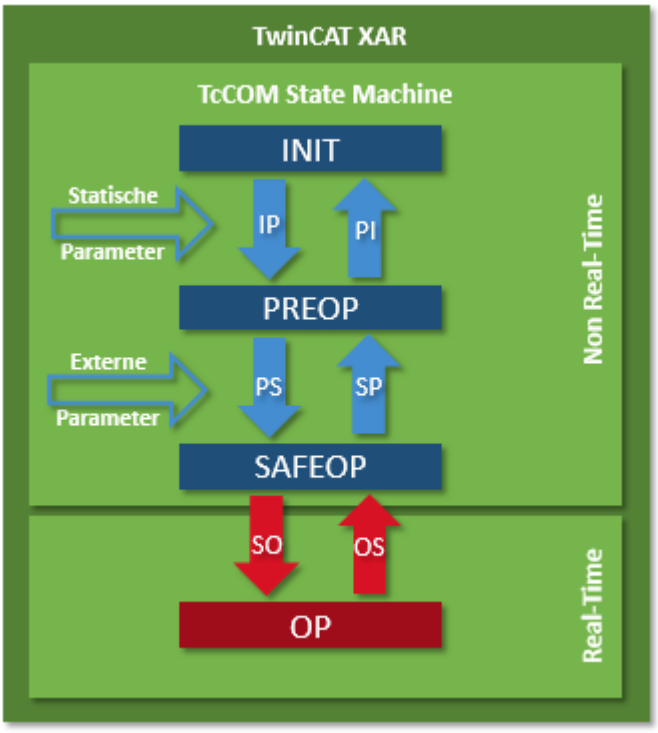
The PLC modules also register via the ObjectServer and can therefore be reached through it. PLC modules vary in terms of complexity. It makes no difference whether only a small filter module is generated or a complete PLC program is packed into a module. Due to the automation, each PLC program is a module within the meaning of TwinCAT modules. Each conventional PLC program is automatically packed into a module and registers itself with the ObjectServer and one or several task modules. Access to the process data of a PLC module (e.g. mapping with regard to a fieldbus driver) is also controlled via the defined data areas and ITcADI.

This behavior remains transparent and invisible for PLC programmers, as long as they decide to explicitly define parts of the PLC program as TwinCAT modules, so that they can be used with suitable flexibility.

6.1.2 TwinCAT module state machine

In addition to the states (INIT, PREOP, SAFEOP and OP), there are corresponding state transitions, within which general or module-specific actions have to be executed or can be executed. The design of the state machine is very simple. In any case, there are only transitions to the next or previous step.

This results in the state transitions: INIT to PREOP (IP), PREOP to SAFEOP (PS) and SAFEOP to OP (SO). In the opposite direction, the following state transitions exist: OP to SAFEOP (OS), SAFEOP to PREOP (SP) and PREOP to INIT (PI). Up to and including the SAFEOP state, all states and state transitions take place within the non-real-time context. Only the transition from SAFEOP to OP, the OP state and the transition from OP to SAFEOP take place in the real-time context. This differentiation is relevant when resources are allocated or enabled, or when modules register or deregister with other modules.



State: INIT

The INIT state is only a virtual state. Immediately after creation of a module, the module changes from INIT to PREOP, i.e. the IP state transition is executed. The instantiation and the IP state transition always take place together, so that the module never remains in INIT state. Only when the module is removed does it remain in INIT state for a short time.

Transition: INIT to PREOP (IP)

During the IP state transition, the module registers with the ObjectServer with its unique ObjectID. The initialization parameters, which are also allocated during object creation, are transferred to the module. During this transition the module cannot establish connections to other modules, because it is not clear whether the other modules already exist and are registered with the ObjectServer. When the module requires system resources (e.g. memory), these can be allocated during the state transition. All allocated resources have to be released again during the transition from PREOP to INIT (PI).

State: PREOP

In PREOP state, module creation is complete and the module is usually fully parameterized, even if further parameters may be added during the transition from PREOP to SAFEOP. The module is registered in the ObjectServer, although no connections with other modules have been created yet.

Transition: PREOP to SAFEOP (PS)

In this state transition the module can establish connections with other modules. To this end it has usually received, among other things, ObjectIDs of other modules with the initialization data, which are now converted to actual connections with these modules via the ObjectServer.

The transition can generally be triggered by the system according to the configurator, or by another module (e.g. the parent module). During this state transition further parameters can be transferred. For example, the parent module can transfer its own parameters to the child module.

State: SAFEOP

The module is still in the non-real-time context and is waiting to be switched to OP state by the system or by other modules.

Transition: SAFEOP to OP (SO)

The state transition from SAFEOP to OP, the state OP, and the transition from OP to SAFEOP take place in the real-time context. System resources may no longer be allocated. On the other hand, resources can now be requested by other modules, and modules can register with other modules, e.g. in order to obtain a cyclic call during tasks.

This transition should not be used for long-running tasks. For example, file operations should be executed during PS.

State: OP

In OP state the module starts working and is fully active in the meaning of the TwinCAT system.

Transition: OP to SAFEOP (OS)

This state transition takes place in the real-time context. All actions from the SO transition are reversed, and all resources requested during the SO transition are released again.

Transition: SAFEOP to PREOP (SP)

All actions from the PS transition are reversed, and all resources requested during the PS transition are released again.

Transition: PREOP to INIT (PI)

All actions from the IP transition are reversed, and all resources requested during the IP transition are released again. The module signs off from the ObjectServer and usually deletes itself (see "Service life").

6.2 Module-to-module communication

TcCOM modules can communicate with one another. This article is intended to provide an overview of the various options. There are four methods of module-to-module communication:

- IO Mapping (linking of input/output symbols)
- IO Data Pointer
- Method calls via interface
- ADS

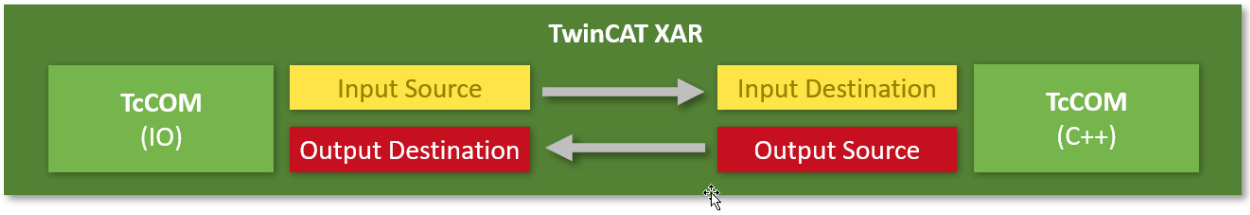
These four methods will now be described.

IO Mapping (linking of input/output symbols)

The inputs and outputs of TcCOM modules can be linked by IO Mapping in the same way as the links to physical symbols in the fieldbus level. To do this, data areas are created in the TMC editor [▶ 119] that describe the corresponding inputs/outputs. These are then linked in the TwinCAT solution.

Mapping is used to provide or transfer data from one module to another. Data consistency is ensured by synchronous or asynchronous mapping, which is carried out after or before the actual cyclic program sequence.

The implementing language (PLC, C++, Matlab®) is irrelevant.



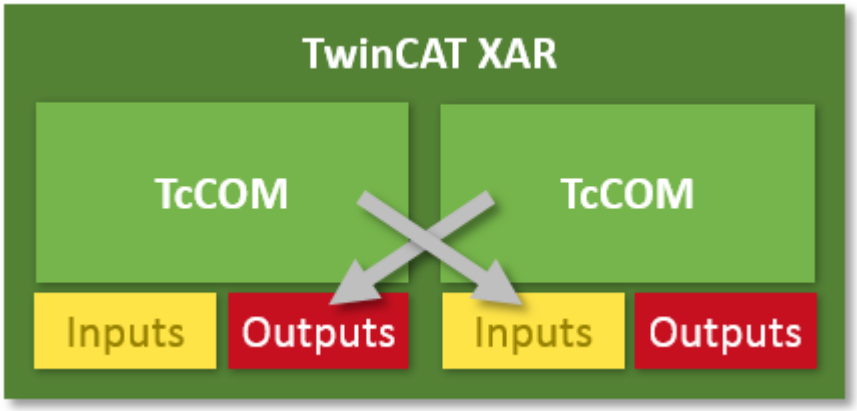
The following sample shows the implementation:

Sample12: Module communication: Using IO mapping [▶ 285]

IO Data Pointer

Direct memory access is also possible within a task via the Data Area Pointers, which are created in the TMC Editor.

If several callers of a task or callers from other tasks occur, the user must ensure the data consistency through appropriate mechanisms. Data pointers are available for C++ and MATLAB®.



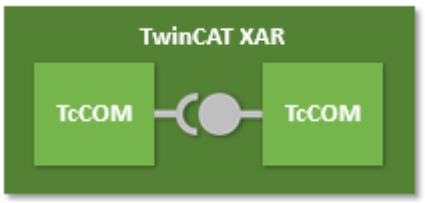
The following sample shows the implementation:

Sample10: Module communication: Use of data pointers [▶ 256]

Method calls via interfaces

As already described, TcCOM modules can offer interfaces that are also defined in the TMC editor. If a module implements them ("Implemented Interfaces" in the TMC editor [▶ 111]), it offers appropriate methods. A calling module will then have an "Interface Pointer" to this module in order to call the methods.

These are blocking calls, meaning that the caller blocks until the called methods come back and the return values of the methods can thus be directly used. If several callers of a task or callers from other tasks occur, the user must ensure the data consistency through appropriate mechanisms.



The following samples show the implementation:

Sample11: Module communication: PLC module invokes method of C module [▶ 257]

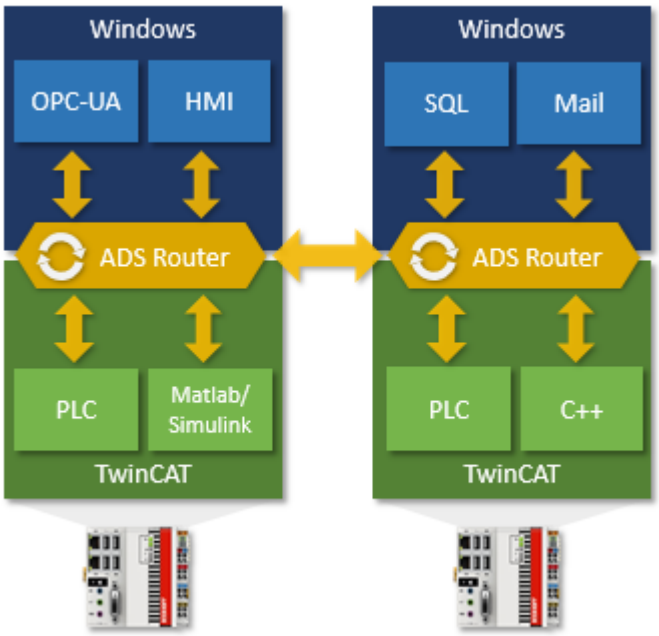
Sample11a: Module communication: C module calls a method of another C module [▶ 284]

Further samples exist for the communication with the PLC [▶ 304].

ADS

As the internal communication of the TwinCAT system in general, ADS can also be used to communicate between modules. Communication in this case is acyclic, event-controlled communication.

At the same time ADS can also be used to collect or provide data from the UserMode and communicate with other controllers (i.e. via the network). ADS can also be used to ensure data-consistent communication, e.g. between tasks/cores/CPUs. In this case TcCOM modules can be both clients (requesters) and servers (providers). The implementing language (PLC, C++, Matlab®) is irrelevant.



The following samples show the implementation:

Sample03: C++ as ADS server [▶ 237]

Sample06: UI-C#-ADS client uploading the symbolic from module [▶ 247]

Sample07: Receiving ADS Notifications [▶ 252]

Sample08: Provision of ADS-RPC [▶ 253]

7 Modules - Handling

TcCOM modules are implemented and loaded after a build.

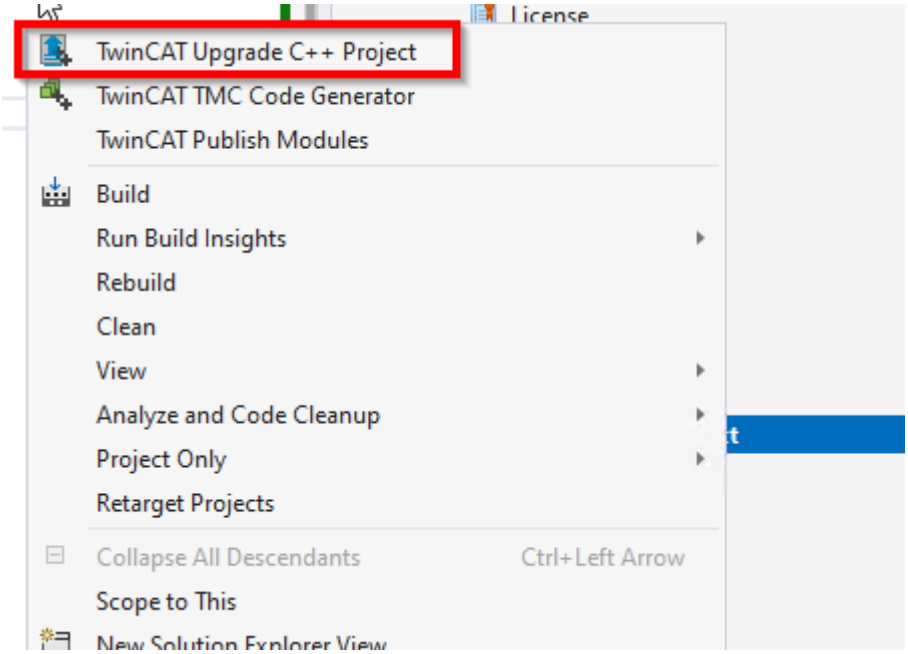
This section describes the handling of modules when they are exchanged between systems. These modules are programmed in Versioned C++ projects [▶ 49] that generate a tmx file in order to load them with the TwinCAT Loader [▶ 50] (since TwinCAT 3.1 Build 4024).

Driver projects

For older projects, Beckhoff recommends switching to the versioned C++ projects [▶ 49], as these offer the following advantages, for example:

- Driver signature [▶ 21] via OEM certificates that can be obtained from Beckhoff.
- Versioned storage [▶ 50] of the binaries.
- Online Change capability [▶ 148], if required.

After installation under TwinCAT 3.1 Build 4026, a conversion is available on the C++ node for migration from the older projects ("driver projects"). This can be accessed via the context menu:



The migration generates a report as an HTML file that should be taken into account.

7.1 Versioned C++ Projects

Versioned TwinCAT C++ projects result in an architecture-dependent file when building.

- TMX files for Windows (x86 / x64) and TwinCAT/BSD® - these are loaded via the TwinCAT Loader [▶ 50]. They must be signed by a TwinCAT user certificate.
- TME files for Beckhoff RT Linux® are loaded by the operating system and do not need to be signed.

If a C++ project was created using the “Versioned C++ Project” template, the binaries are stored by Publish in the TwinCAT Engineering Repository (under TwinCAT 3.1 Build 4024: *C:\TwinCAT\3.1\Repository* and TwinCAT 3.1 Build 4026 C:\ProgramData\Beckhoff\TwinCAT\3.1\Repository) at a vendor- and version-specific location.

From here, modules are transferred to the target system, if they are needed:

- Windows with TwinCAT 3.1 Build 4024: C:\TwinCAT\3.1\Boot\Repository
- Windows with TwinCAT 3.1 Build 4026: C:\ProgramData\Beckhoff\TwinCAT\3.1\Boot\Repository
- TwinCAT/BSD®: /usr/local/etc/TwinCAT/3.1/Boot/Repository

- Beckhoff RT Linux®: /etc/TwinCAT/3.1/Boot/Repository

This can be either at the time of activation (**Activate Configuration**) or at the time of the Online Change [▶ 148].

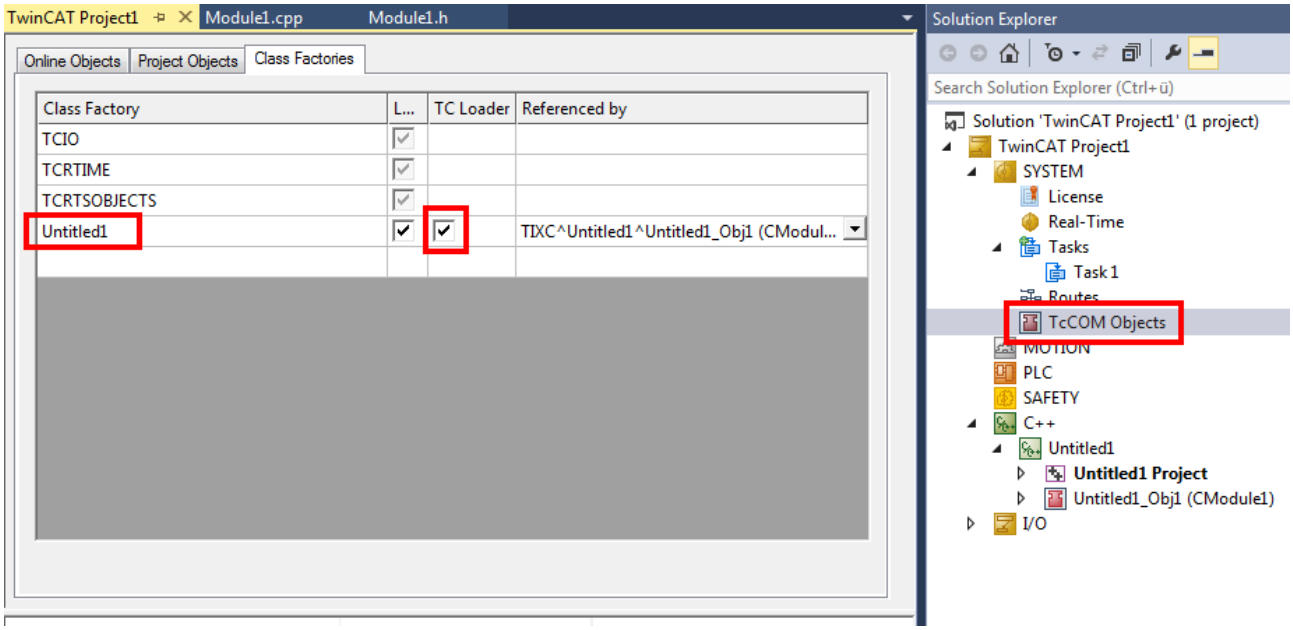
Additionally, it is possible to create an archive for the transfer between engineering systems of the binary version of this project, which is configured by the project properties [▶ 143].

7.2 Starting Modules

TwinCAT C++ modules can be started in two ways:

- TMX files are loaded by the TwinCAT Loader: [▶ 50] The TwinCAT Loader starts the TwinCAT module.
  - The TwinCAT Loader requires a signature [▶ 20] with TwinCAT user certificate.
  - This option is mandatory for encrypted modules [▶ 53].
  - The TwinCAT Loader is required for the versioned C++ projects [▶ 85].
- TME files are loaded by the operating system

Via **System** -> **TcCOM Modules** -> **Class Factories** tab you can see whether the TwinCAT Loader or the operating system is used:



7.3 TwinCAT Loader

From TwinCAT 3.1 Build 4024.0

The functionality described here is available from TwinCAT 3.1. 4024.0.

TwinCAT 3 has an integrated function for loading modules created as TMX files. This is offered for Windows and TwinCAT/BSD® systems.
Modules loaded with the TwinCAT Loader:

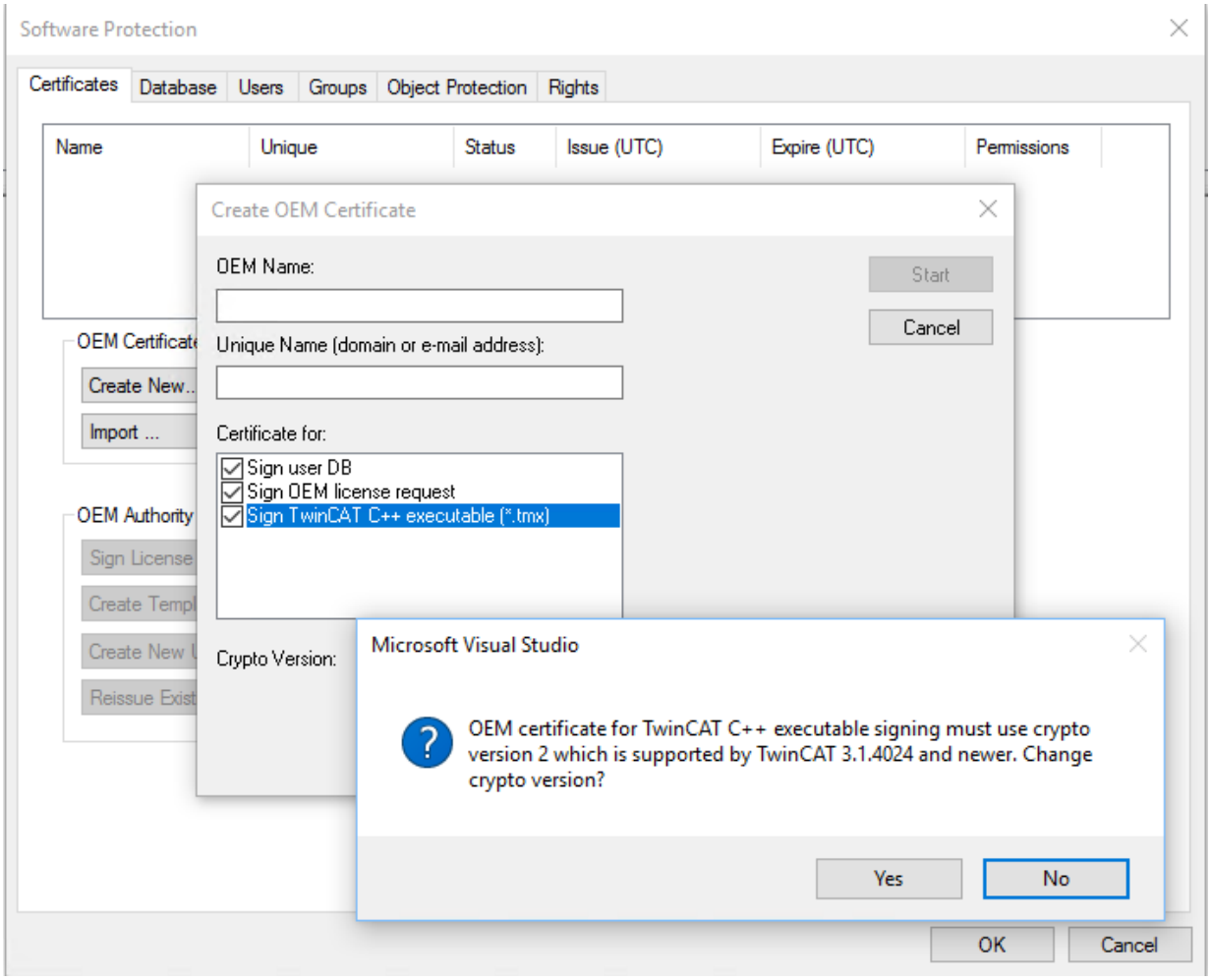
- **Must** be signed: Test signing [▶ 51].
- **Can** be encrypted: Encrypting Modules [▶ 53], for which the TwinCAT Software Protection must be configured with a User DB.

For Beckhoff RT Linux®, the modules are loaded via the operating system and do not have to be signed, but cannot be encrypted with TwinCAT Software Protection.

7.3.1 Test signing

The test signing for TwinCAT can be carried out with the same TwinCAT user certificate as for the actual delivery (see Request TwinCAT 3 user certificate [▶ 24]).

1. For test operation, e.g., during software development, it is sufficient to create a TwinCAT user certificate [▶ 25]. Make sure that you select the purpose "Sign TwinCAT C++ executable (*.tmx)". For this the Crypto version 2 is required, a message appears.

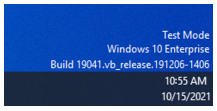


On XAR (and XAE, if it is a local test), activate the test mode so that the operating system can accept the self-signed certificates. This can be done on both engineering systems (XAE) and runtime systems (XAR).

For Windows

Use the administrator prompt to execute the following:
`bcdedit /set testsigning yes`
and reboot the target system.
You may have to switch off "SecureBoot" for this, which can be done in the bios.

If test signing mode is enabled, this is displayed at the bottom right of the desktop. The system now accepts all signed drivers for execution.



For TwinCAT/BSD®

In the file `/usr/local/etc/TwinCAT/3.1/TcRegistry.xml` enter "`<Value Name="EnableTestSigning" Type="DW"1</Value>`" under Key "`System`".

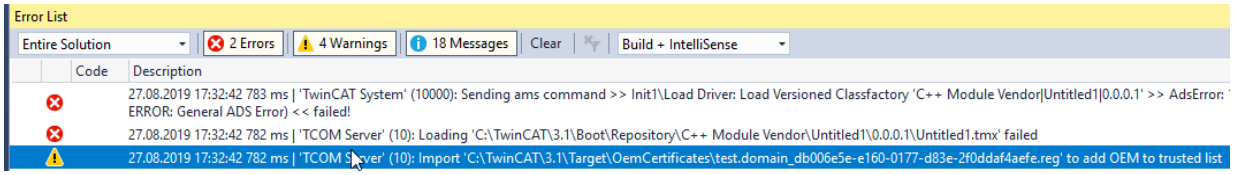
```
<Key Name="System">
  <Value Name="RunAsDevice" Type="DW">1</Value>
  <Value Name="RTimeMode" Type="DW">0</Value>
  <Value Name="AmsNetId" Type="BIN">052445B00101</Value>
  <Value Name="LockedMemSize" Type="DW">33554432</Value>
  <Value Name="EnableTestSigning" Type="DW">1</Value>
</Key>
```

Then restart the TwinCAT system service:

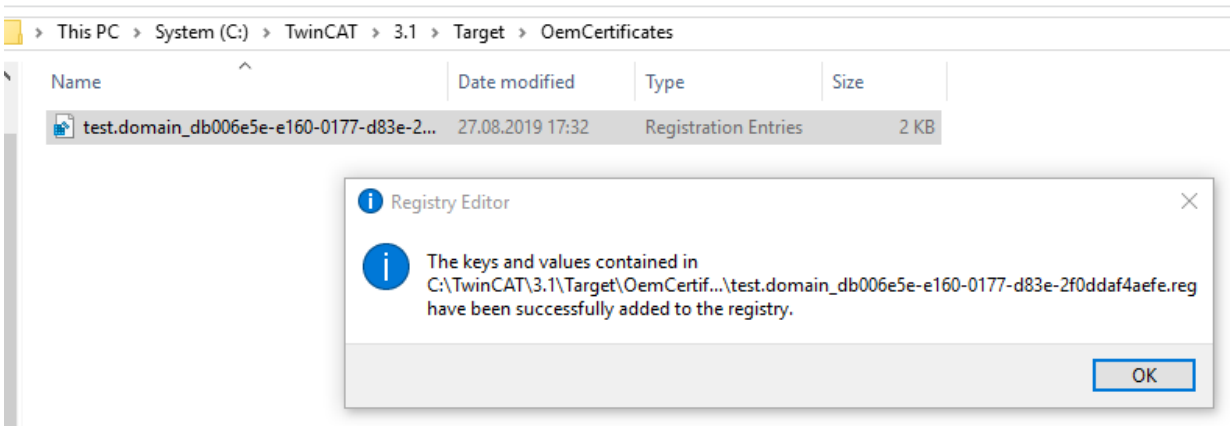
```
doas service TcSystemService restart
```

After the respective procedure, the system accepts all signed drivers for execution.

1. During the first activation (Activate Configuration) with a TwinCAT user certificate, the target system detects that the certificate is not trusted and the activation process is aborted:



For Windows:
A local user with administration rights can trust the certificate via the created REG file by simply executing it:



For TwinCAT/BSD®:
If the "Tcimportcert" package is not installed, install it: `pkg install TcImportCert`
Trust the certificate via `doas tcimportcert /usr/local/etc/TwinCAT/3.1/Target/OemCertificates/<CreatedFile>.reg`.
Then restart the TwinCAT System Service or reboot the system:
`doas service TcSystemService restart`

⇨ This process only enables C++ modules with a signature from the trusted TwinCAT user certificates to run.

2. Following this process you can use the TwinCAT user certificate for signing with the test mode of the operating system.
   This is configured in the project properties [▶ 144].
   Use the TcSignTool [▶ 55] to avoid storing the password of the TwinCAT user certificate in the project, where it would also end up in version management, for example.

If you want to use the TwinCAT user certificate without TestMode for delivery, you must have the certificate countersigned by Beckhoff [▶ 23].

7.3.2 Encrypting Modules

TwinCAT C++ modules loaded via the TwinCAT Loader (TMX files) can be encrypted, i.e. a key protects the content of the driver against manipulation and reverse engineering at file level.

Since the TwinCAT Loader is not used under Beckhoff RT Linux®, the functionality described here cannot be used.

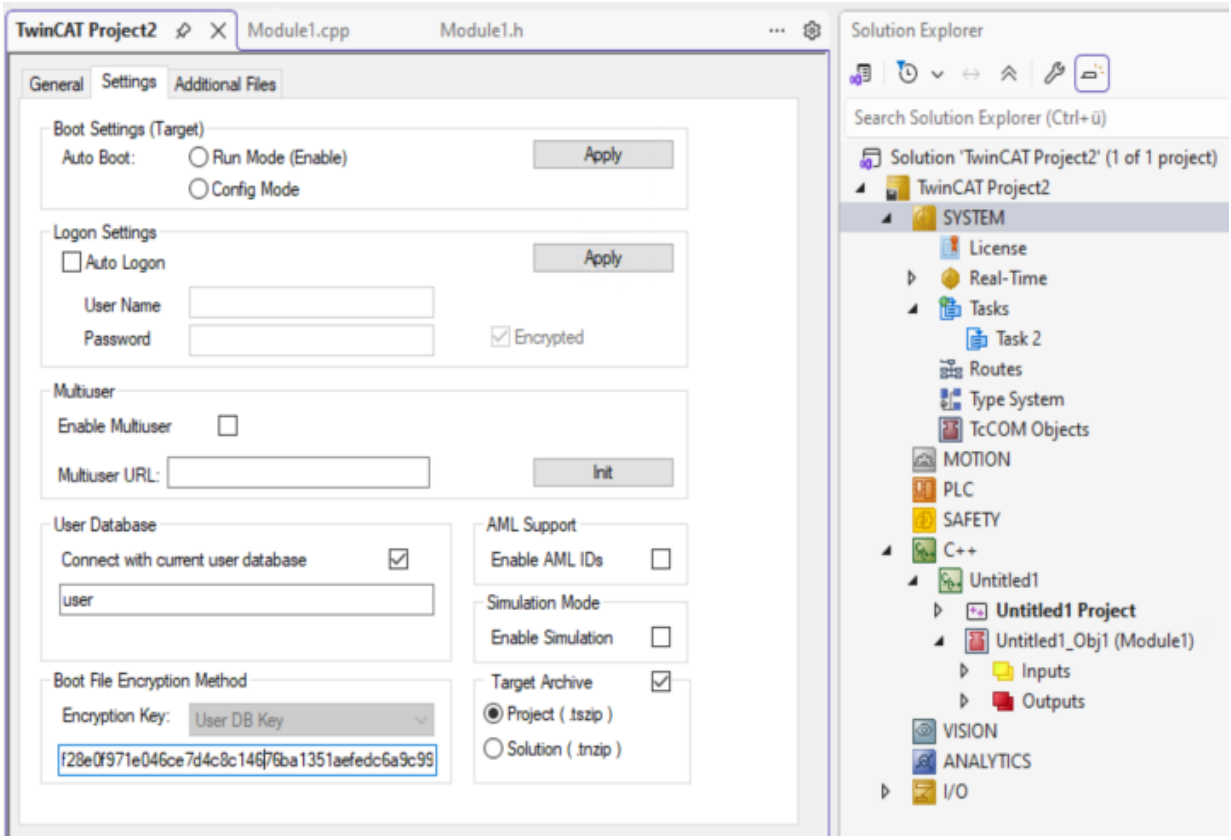
No debugging

Encrypted modules cannot be searched for errors. Encrypted modules are not displayed in the debugger.

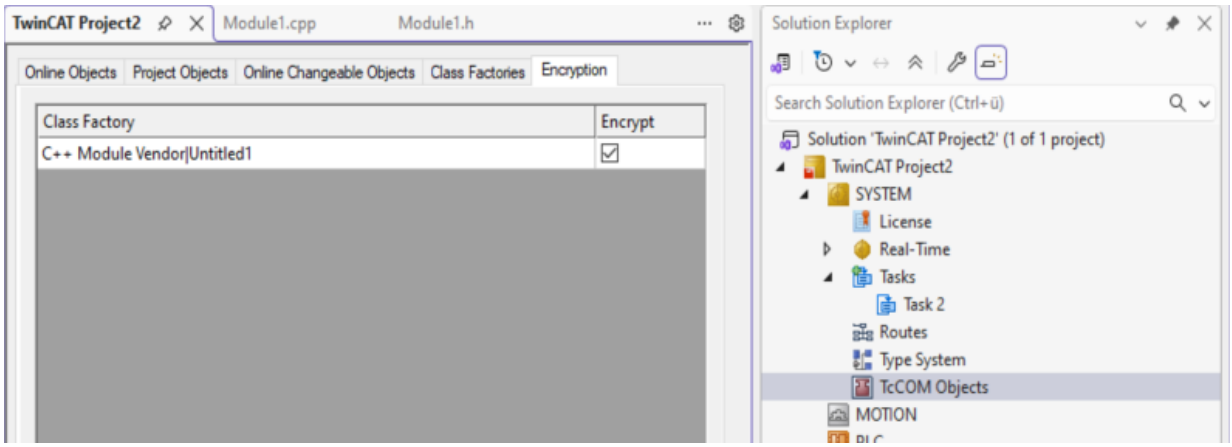
Module encryption is enabled as follows:

- ✓ A TwinCAT user certificate with "Sign TwinCAT C++ executable (*.tmx)" and thus "Crypto Version" 2 is required.
- ✓ The Twincat Software Protection must be configured.

1. In the system tree, select the Solution **User DB Key** as the Boot File Encryption Key.



2. Select the appropriate entry under **System->TcCOM Objects -> Encryption**.



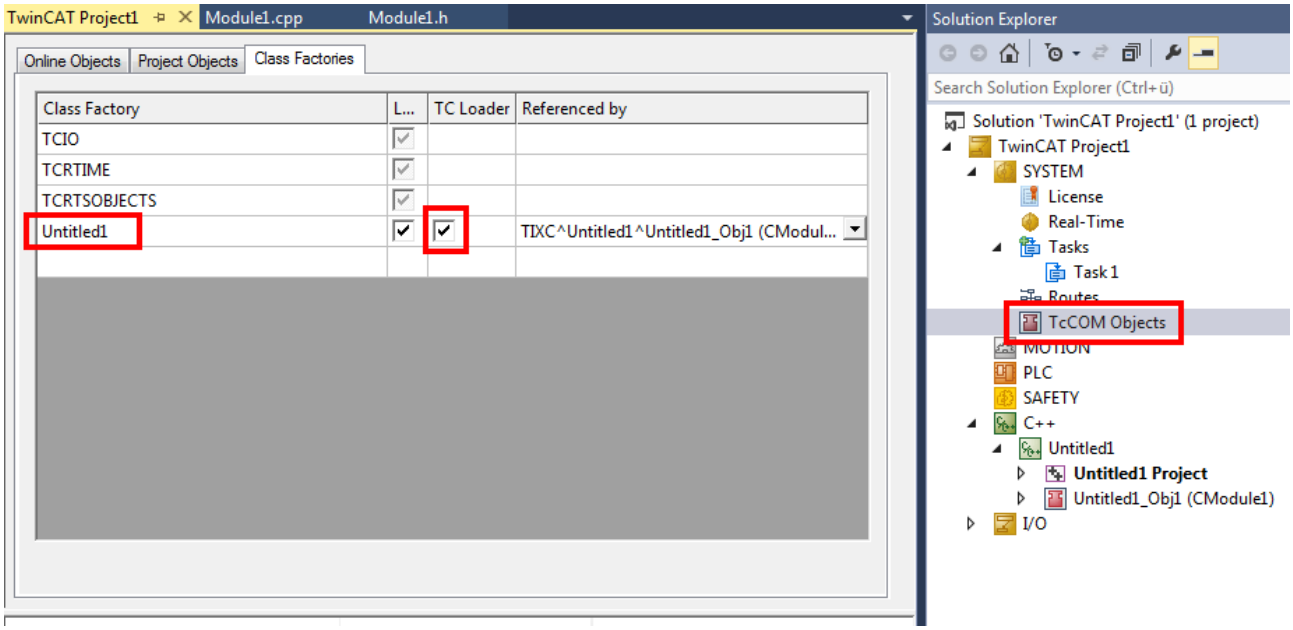
3. To start, an encrypted module must be loaded with the TwinCAT Loader.

⇨ For versioned TMX: The drivers are stored unencrypted in XAE and encrypted when they are activated on the target system. The encryption can also be checked in the project_boot directory (open the TMX file in the editor. File starts with "TcEncryptedFile".

TwinCAT C++ modules can be started in two ways:

- TMX files are loaded by the TwinCAT Loader: [▶ 50] The TwinCAT Loader starts the TwinCAT module.
  - The TwinCAT Loader requires a signature [▶ 20] with TwinCAT user certificate.
  - This option is mandatory for encrypted modules [▶ 53].
  - The TwinCAT Loader is required for the versioned C++ projects [▶ 85].
- TME files are loaded by the operating system

Via **System** -> **TcCOM Modules** -> **Class Factories** tab you can see whether the TwinCAT Loader or the operating system is used:



Also see about this

- Versioned C++ Projects [▶ 49]

7.3.3 Return Codes

Loading a module with the TwinCAT Loader can fail for various reasons.

Return codes under Windows

Hex	Description
0xC1	File is corrupted; PE file checksum error.
0x241	Signing error: TwinCAT user certificate does not match the file hash.
0x4FB	Signing error: File not signed.
0x1772	File is encrypted, but cannot be decoded with known keys.

Return codes under TwinCAT/BSD®

Hex	Description
0xC0000001 0xC0000004 0xC000007B 0xC0000173 0xC0000221	File is corrupted; PE file checksum error.
0xC0000102 0xC0000424	Signing error: TwinCAT user certificate does not match the file hash.
0x4FB	Signing error: File not signed.
0xC0000428	Signing error: Used certificate not countersigned. Test mode necessary.
0xC0000603	Signing error: Certificate is not trusted. Add necessary (see "tcinsertcert")
0xC0000385	Signing error: File not signed. Check the settings in Engineering.
0xC0000293	File is encrypted, but cannot be decoded with known keys.
0xC0000225	File not found.

7.3.4 TcSignTool - Storage of the certificate password outside the project

The TcSignTool can be used to store a password for a TwinCAT user certificate in the registry. Thus, the password is not needed in the projects, where the passwords would end up unintentionally in version control systems.

The TcSignTool is a command line program located in the path *C:\TwinCAT\3.1\sdk\Bin* or *C:\Program Files (x86)\Beckhoff\TwinCAT\3.1\SDK\Bin*.

The storage of the password is carried out with the following parameters:

```
tcsigntool grant /f "…CustomConfig\Certificates\MyCertificate.tccert" /p MyPassword
```

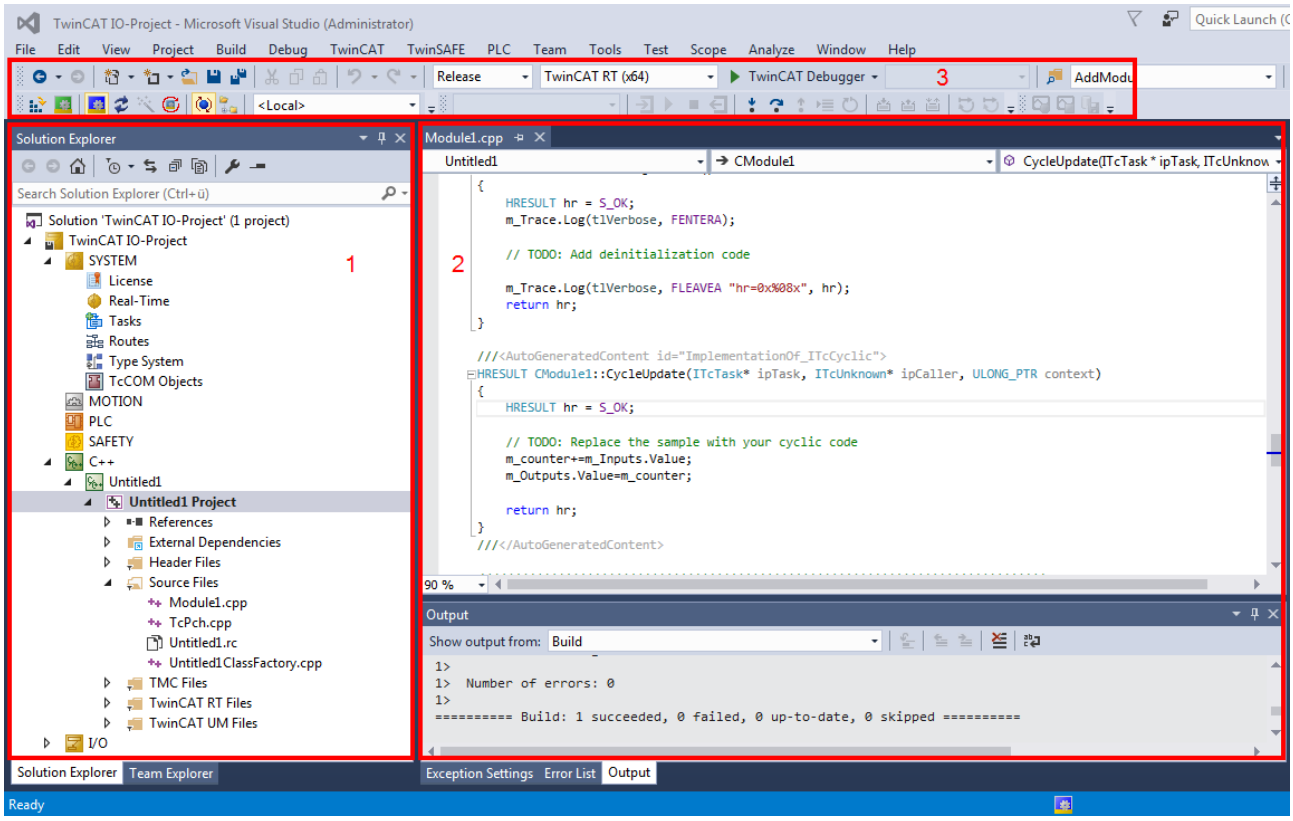
The password is deleted with the following parameters:

```
tcsigntool grant /f "…CustomConfig\Certificates\MyCertificate.tccert" /r
```

The unencrypted password is stored under: `HKEY_CURRENT_USER\SOFTWARE\Beckhoff\TcSignTool\`

8 TwinCAT C++ development

Overview of the development environment



The layout of Visual Studio is flexible and adaptable, so that only a brief overview of a common configuration can be provided here. The user is free to configure windows and arrangements as required.

1. In the TwinCAT Solution, a TwinCAT C++ project can be created by right-clicking on the C++ icon. This project contains the sources (“Untitled Project”) of several modules [▶ 35], if applicable, and module instances ("Untitled1_Obj1 (CModule1)") can be created. The module instances have inputs/ outputs, which can be linked in the usual way ("Link"). There are further options [▶ 46] for module interaction.
2. The Visual Studio editor for Visual C++ is used for programming. Here, particular attention should be paid to the drop-down boxes for fast navigation within a file. In the lower area the result of the compile process is output. It is also possible to switch to the TwinCAT messages (see Module messages for the Engineering (logging / tracing) [▶ 209]).
   The usual features such as breakpoints (cf. Debugging [▶ 75]) can be used in the editors.
3. The freely configurable toolbar usually contains the toolbar for TwinCAT XAE Base. **Activate Configuration**, **RUN**, **CONFIG**, Choose Target System (in this case **<Local>**) and several other buttons provide fast access to frequently used functions. The **TwinCAT Debugger** is the button for establishing a connection to the target system with regard to C++ modules (the PLC uses an independent debugger). Like in other C++ programs, and in contrast to PLC, in TwinCAT C++ a distinction has to be made between "Release" and "Debug". In a build process for "Release", the code is optimized to such an extent that a debugger may no longer reliably reach the breakpoints, and incorrect data may be displayed.

Procedure

This section describes the processes for programming, compiling and starting a TwinCAT C++ project.

It provides a general overview of the engineering process for TwinCAT C++ projects with reference to the corresponding detailed documentation. The quick start [▶ 58] guide describes the individual common steps.

1. Type declaration and module type:
   The TwinCAT Module Class Editor (TMC) [▶ 88] and TMC Code Generator are used for the definition of data types and interfaces, and also for the modules that use these.
   The TMC Code Generator generates source code based on the processed TMC file and prepares data types / interfaces for use in other projects (like PLC).
   Editing and starting the Code Generator can take place as often as you like – the code generation pays attention to programmed user code and preserves it.
2. Programming
   The familiar Visual Studio C++ programming environment is used for the development and debugging [▶ 75] of the user-defined code within the code template.
3. Instantiating modules [▶ 35]
   The program describes a class, which is instantiated as objects. The TwinCAT Module Instance Configurator [▶ 129] is used to configure the instance. General configuration elements are: Assign task, download symbol information for runtime (TwinCAT Module Instance (TMI) file), or specify parameter/ interface pointers.
4. Mapping of variables
   The input and output variables of an object can be linked with variables of other objects or PLC projects, using the standard TwinCAT System Manager.
5. Building
   During the building (compilation and linking) of the TwinCAT C++ project, all components are compiled for the selected platform. The platform is determined automatically when the target system is selected.
6. Publishing
   During publishing of a module, the drivers for all platforms are created, and the module is prepared for distribution. The directory created in the repository can be distributed without having to transfer the source code. Only binary code with the interface description is transferred.
7. Signature for Windows and TwinCAT/BSD® (see Module signing [▶ 20])
   The TwinCAT C++ projects in TMX files must be signed. The signature process can be user-defined [▶ 34]. This does not apply to TME files under Beckhoff RT Linux®.
8. Activation
   The TwinCAT C++ projects can be activated like any other TwinCAT project via **Activate Configuration**. The dialog then requests to switch TwinCAT to RUN mode.
   Debugging [▶ 75] in real-time (which is familiar from IEC61131-based systems) and the setting of (conditional) breakpoints is possible for TwinCAT C++ modules.
   Debugging is currently not supported for Beckhoff RT Linux® systems.

⇨ The module runs under real-time conditions.

9 Quick Start

This quick start shows how you can familiarize yourself with the TwinCAT C++ module engineering in a short time. Each step in the creation of a module that runs in a real-time context is described in detail.

Two different scenarios are discussed:

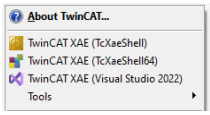
- TwinCAT Versioned C++ projects.
  Modules based on such project are loaded by TwinCAT and stored versioned in binary form.
- We illustrate how to switch between the different versions via Online Change, using versioned C++ projects as a basis.

Before the quick start, please pay attention to the preparation - just once! [▶ 20] In particular, prepare the respective driver signing if you want to run your project on Windows or TwinCAT/BSD®. This is not required for Beckhoff RT Linux®.

9.1 Create TwinCAT 3 project

Start the TwinCAT Engineering Environment (XAE)

Microsoft Visual Studio® can be started via the TwinCAT SysTray icon.

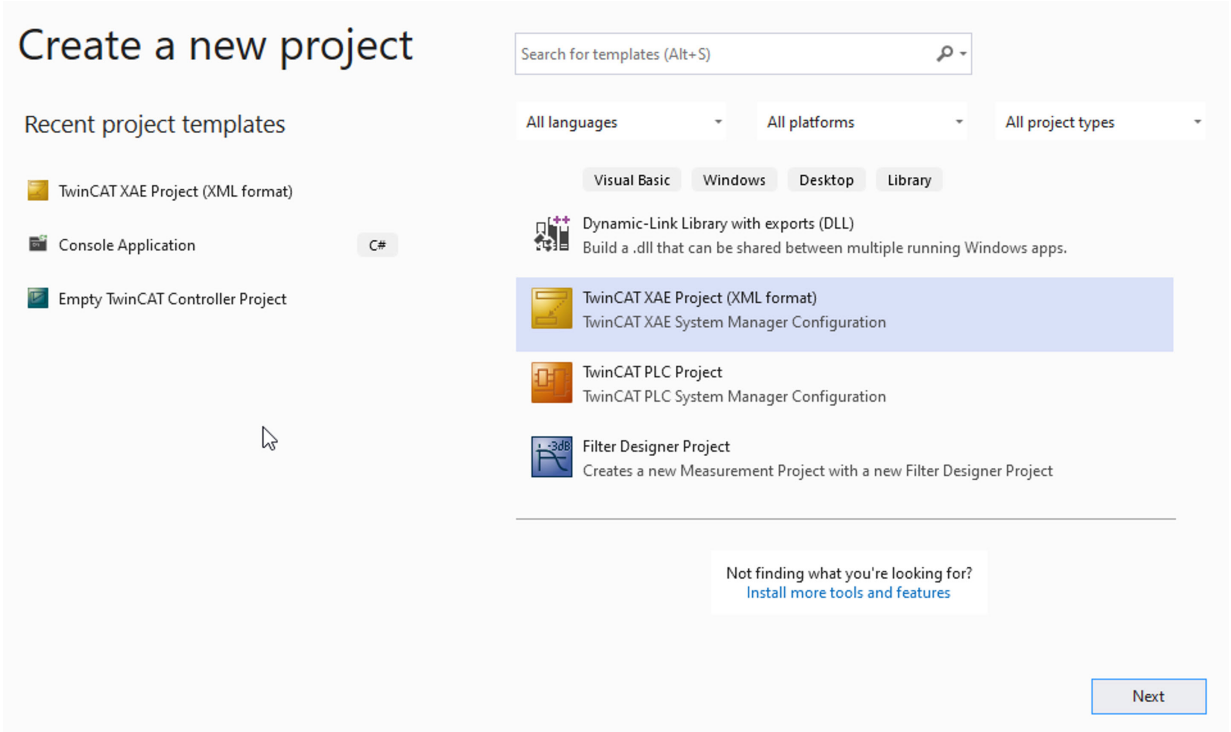


The Microsoft Visual Studio® versions offered are those in which TwinCAT is activated.
Alternatively, Microsoft Visual Studio® can also be started via the Start menu.

Creating a TwinCAT 3 C++ project

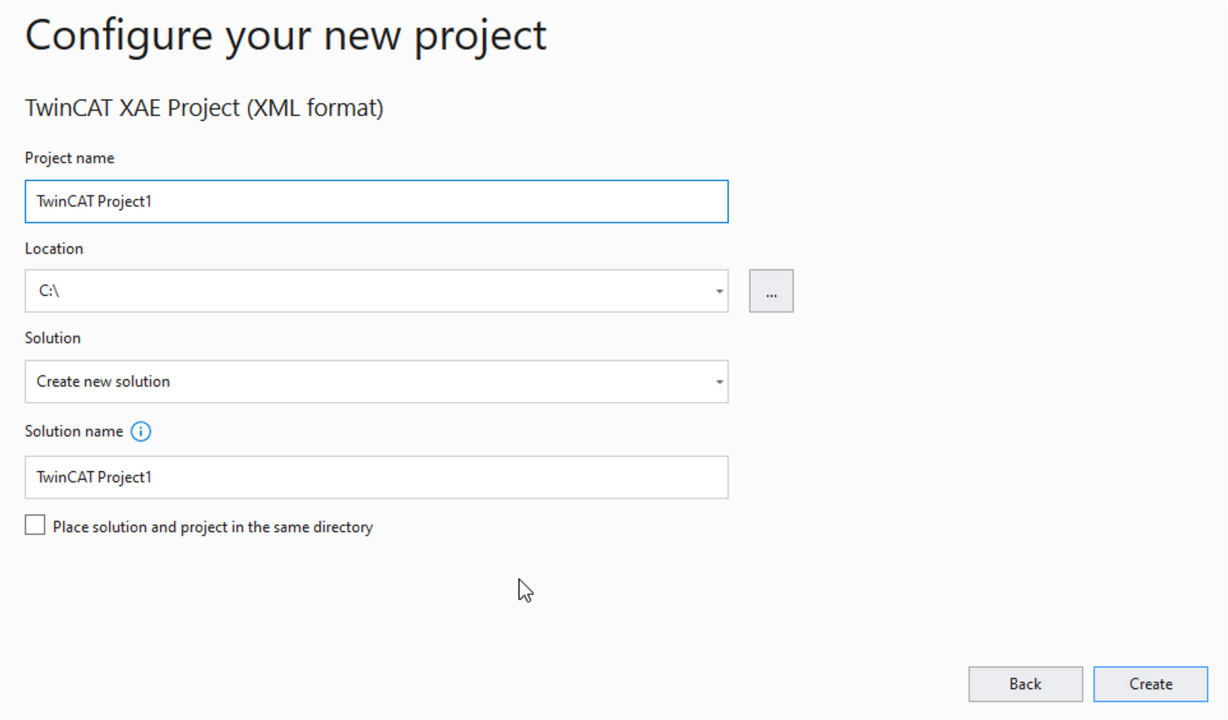
Carry out the following steps to create a TwinCAT C++ project:

1. Select **New TwinCAT Project …** via the Start page.

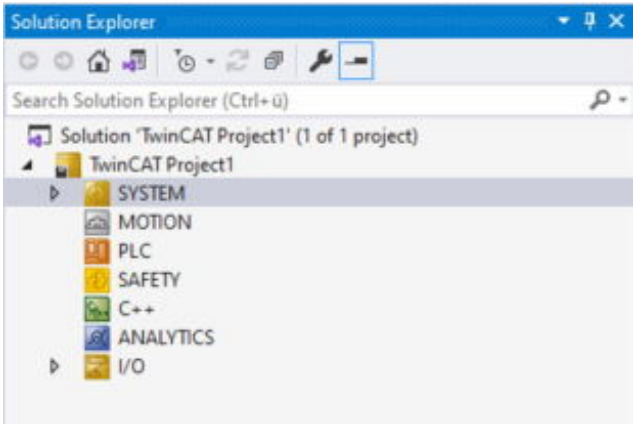


2. Alternatively, you can create a project by clicking on: **File** -> **New** -> **Project**.
   ⇨ All existing project templates are displayed.
3. Select **TwinCAT XAE Project** and optionally enter a suitable project name.

4. Click **OK**.



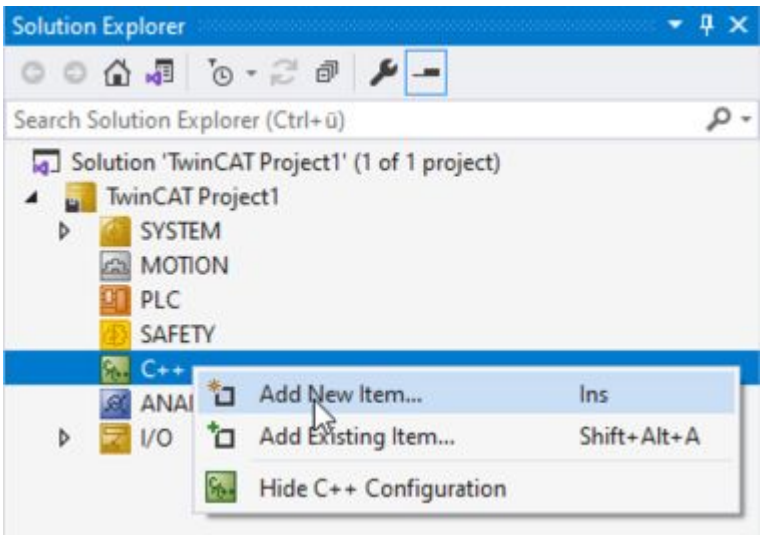
⇨ The Visual Studio Solution Explorer then displays the TwinCAT 3 project.



9.2 Create TwinCAT 3 C++ project

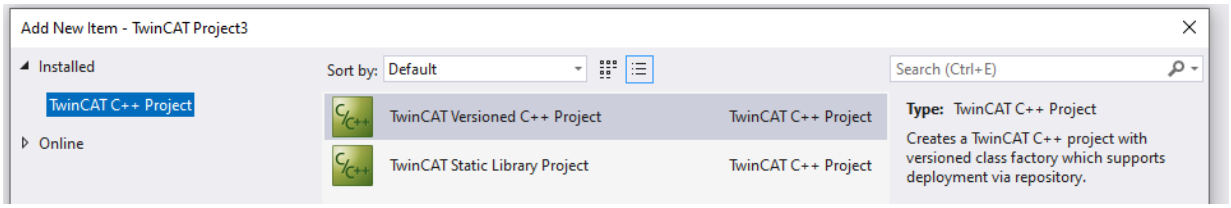
After creating a TwinCAT 3 project, open the C++ node and proceed as follows:

1. Right-click **C++** and select **Add New Item...**.
   If the green C++ symbol is not listed, this means that either a target device is selected that doesn't support TwinCAT C++ or the TwinCAT Solution is currently open in a version of Visual Studio that is not C++-capable (cf. Installation [▶ 17]).



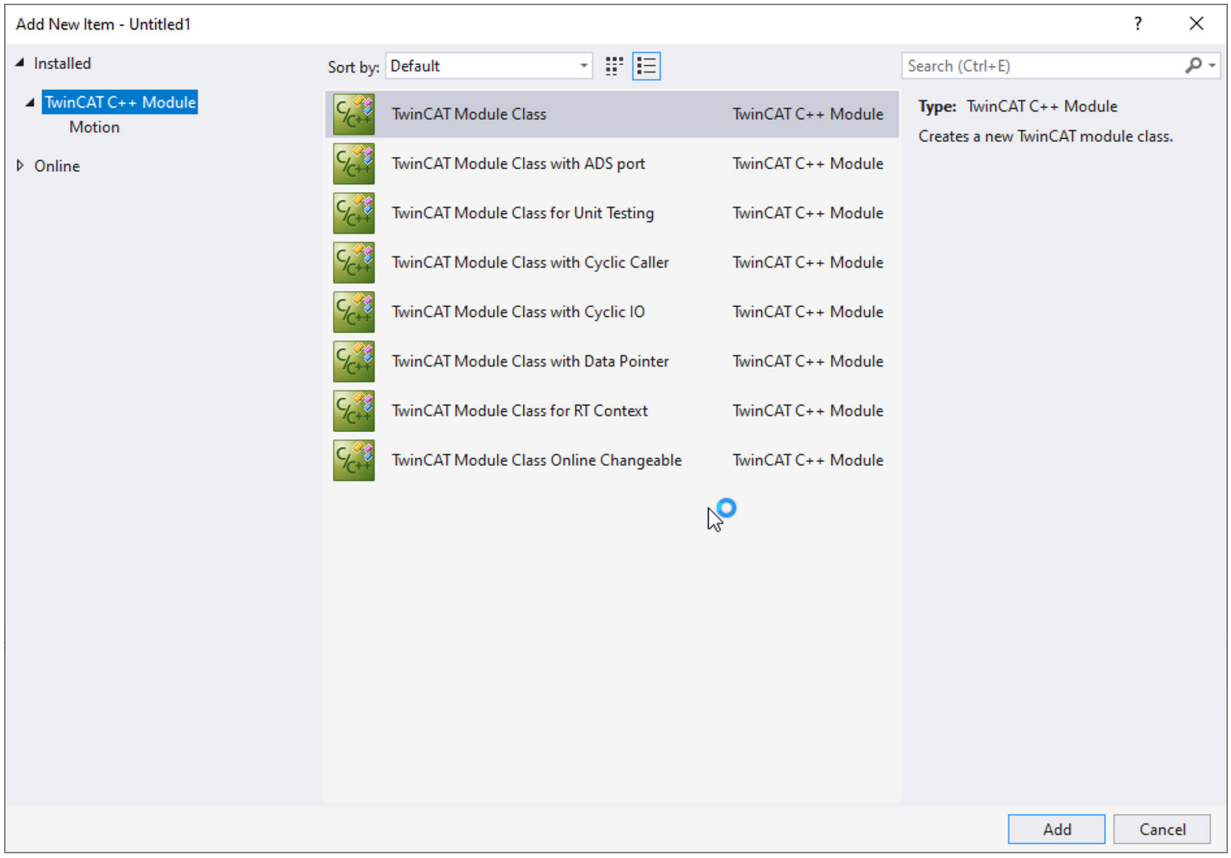
   ⇨ The TwinCAT C++ Project Wizard [▶ 85] is shown and all existing project templates are listed.

2. Select **TwinCAT Versioned C++ Project**, optionally enter a related project name and click on **OK**. Alternatively, use the **TwinCAT Static Library Project**, which provides an environment for programming static TwinCAT C++ libraries (see Sample 25 [▶ 296]).



   ⇨ The TwinCAT Module wizard [▶ 86] is displayed.

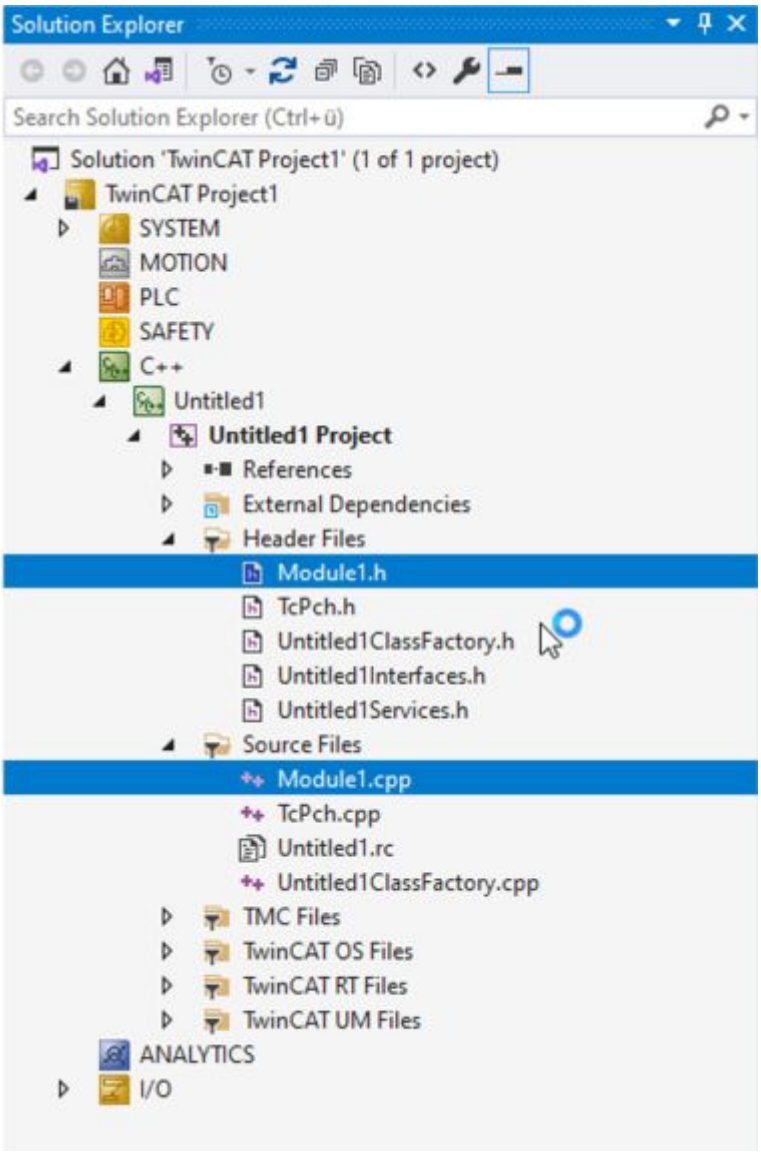
3. In this case, select **TwinCAT Module Class with Cyclic IO** and click on **OK**. If you want to use the Online Change capability, select the **TwinCAT Module Class Online Changeable**.



4. Enter a unique name in the **TwinCAT Class Wizard** dialog box or continue with the "Module1" suggestion.



⇨ A TwinCAT 3 C++ project will then be created on the basis of the selected template:

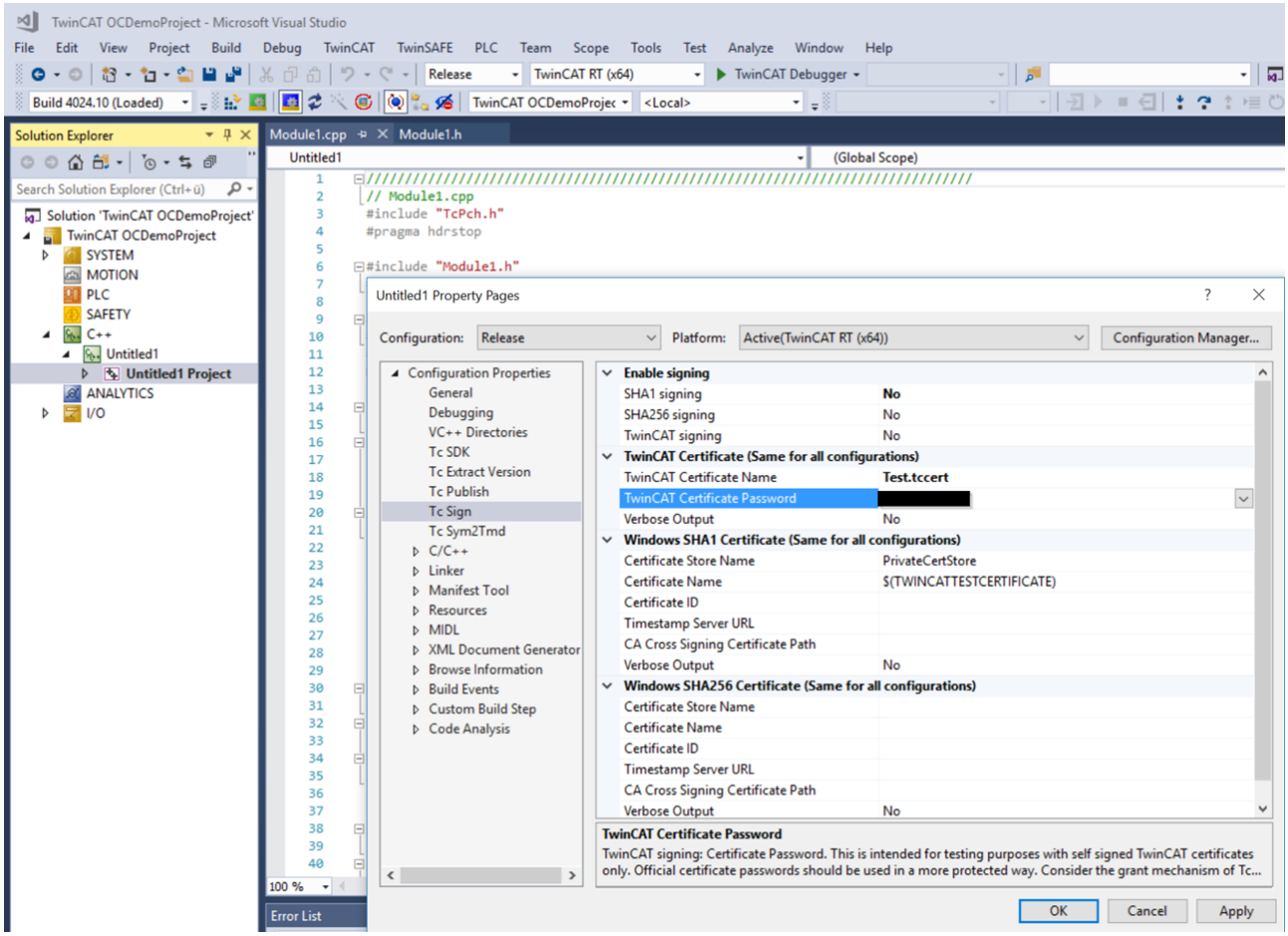


9.3 TwinCAT 3 C++ Configure project

✓ You have created a TwinCAT Versioned C++ project and also had a module created by the wizard.

✓ You want to use the project with Windows or TwinCAT/BSD® as the runtime system. Signing can be omitted for Beckhoff RT Linux® as the operating system does not have a file-based signature.

1. Go to the properties of the project by right-clicking.
2. Activate the TwinCAT Signing in the **TC Sign** tab.
3. If you have not yet created a TwinCAT user certificate, follow the instructions and observe to select the **Sign TwinCAT C++ executeables**.
4. Enter the file name of the TwinCAT user certificate and the password.
   (Note that this is stored unencrypted in the solution and is therefore also loaded on servers via version control, for example. If necessary, use the TcSignTool [▶ 55].)

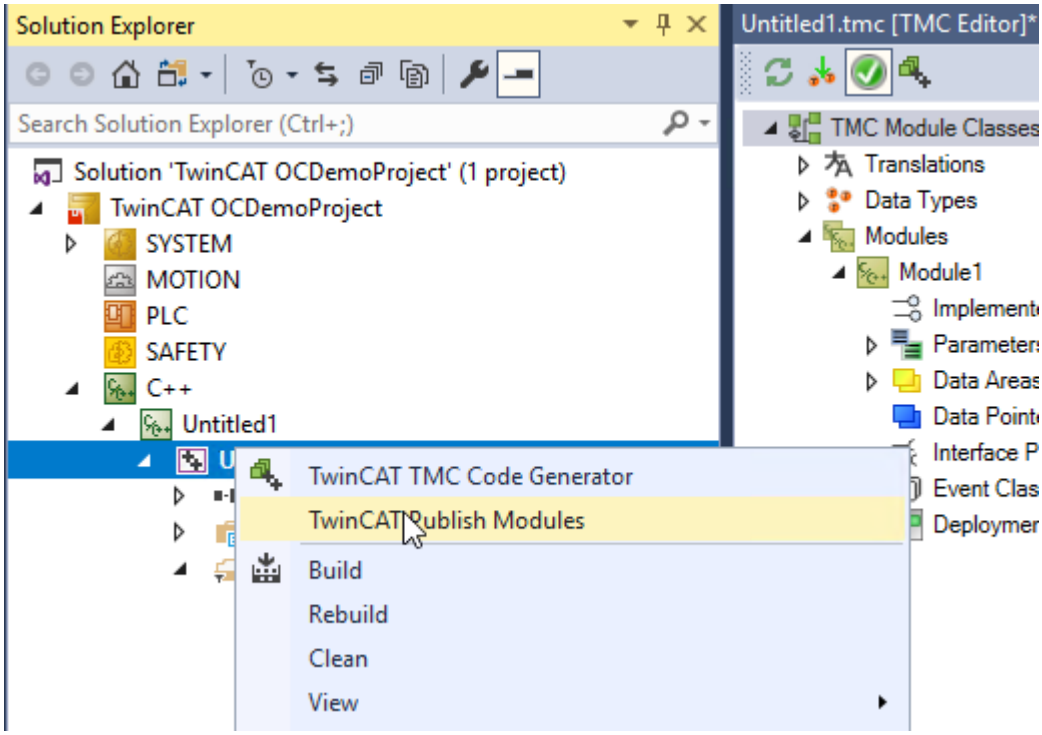


9.4 Implement TwinCAT 3 C++ project

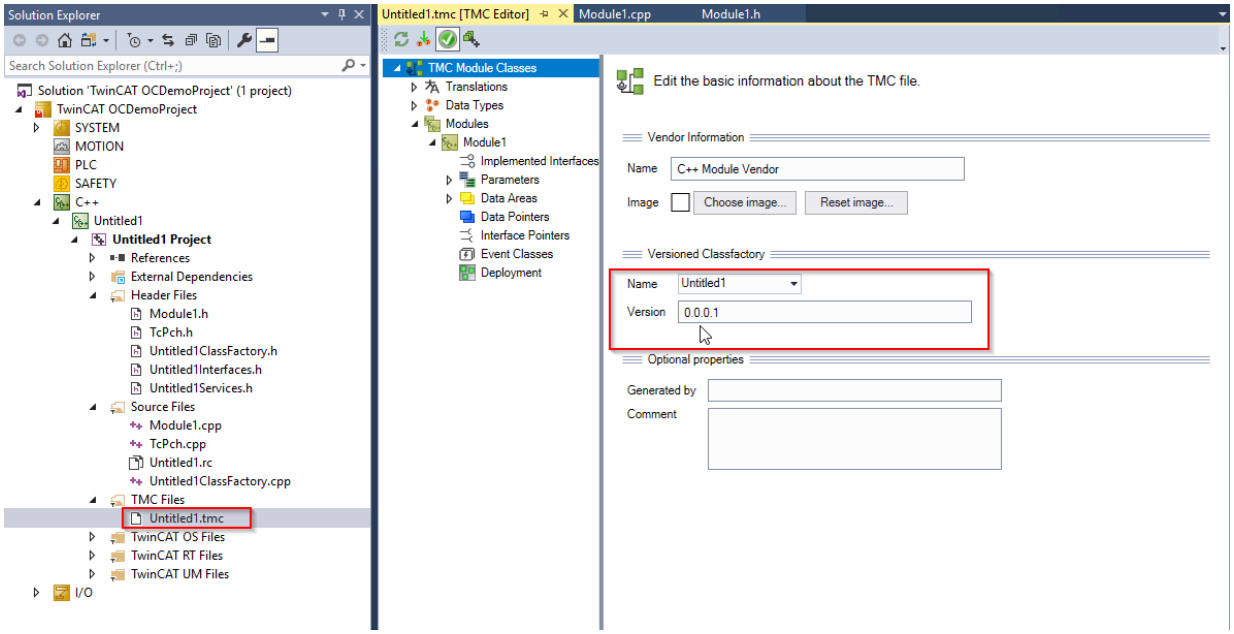
This article describes how the sample project can be changed.

The implementation begins after creating a TwinCAT C++ project and opening <MyClass>.cpp (Module1.cpp in this sample).

1. The <MyClass>::CycleUpdate() method is cyclically called – this is the point where the cyclic logic is to be positioned. At this point, add the entire cyclic code. Use the drop-down menu at the top of the editor for navigation.



2. In this case a counter is incremented by the value of the Value variable in the input image (`m_Inputs`). Replace a line in order to increment the counter without dependence on the value of the input image. Replace this line:
   `m_counter+=m_Inputs.Value;`
   with this one:
   `m_counter++;`
3. Save the modifications.
4. If you have prepared the module for Online Change, please note that version 0.0.0.1 has been implemented here, as you can see in the TMC editor.

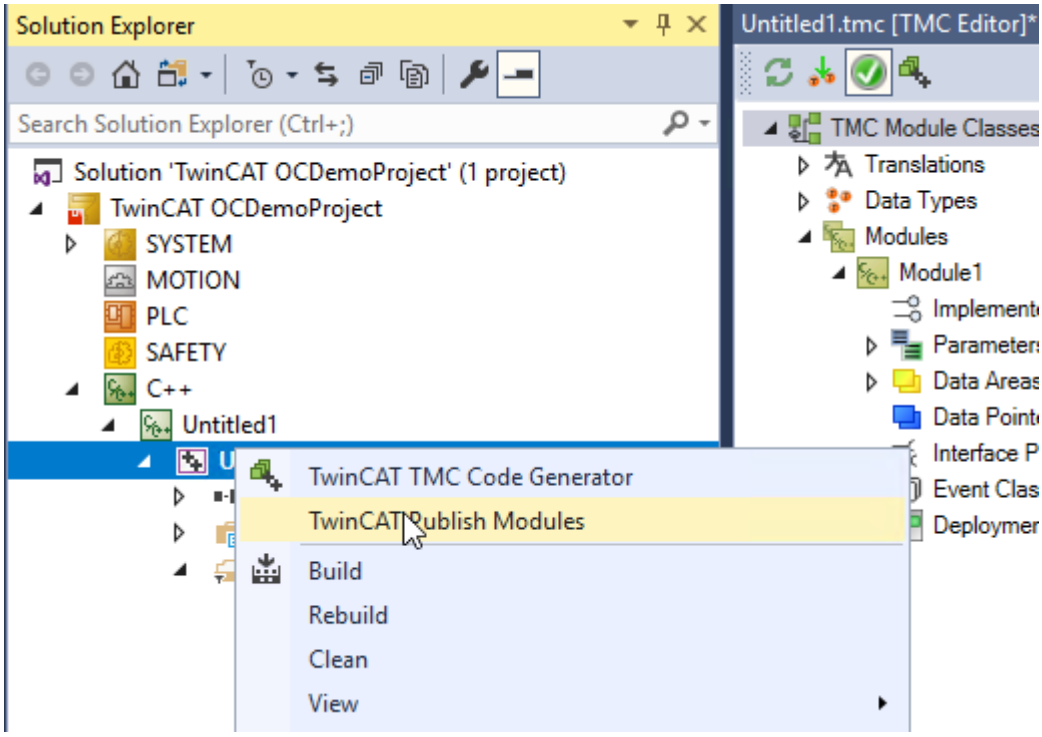


5. The project can now be built and the actual implementation is done until local tests are successful.

9.5 Publish TwinCAT 3 C++ project in version 0.0.0.1

After a TwinCAT C++ project has been created, compiled and tested, the project can be published, i.e. it is prepared for distribution.

1. Click in the context menu on **TwinCAT Publish Modules** to publish the module and thus make it available in the local repository as version 0.0.0.1



⇨ The module is published in version 0.0.0.1 with the incremental counter on the engineering device.

9.6 Implement and publish TwinCAT 3 C++ project version 0.0.0.2

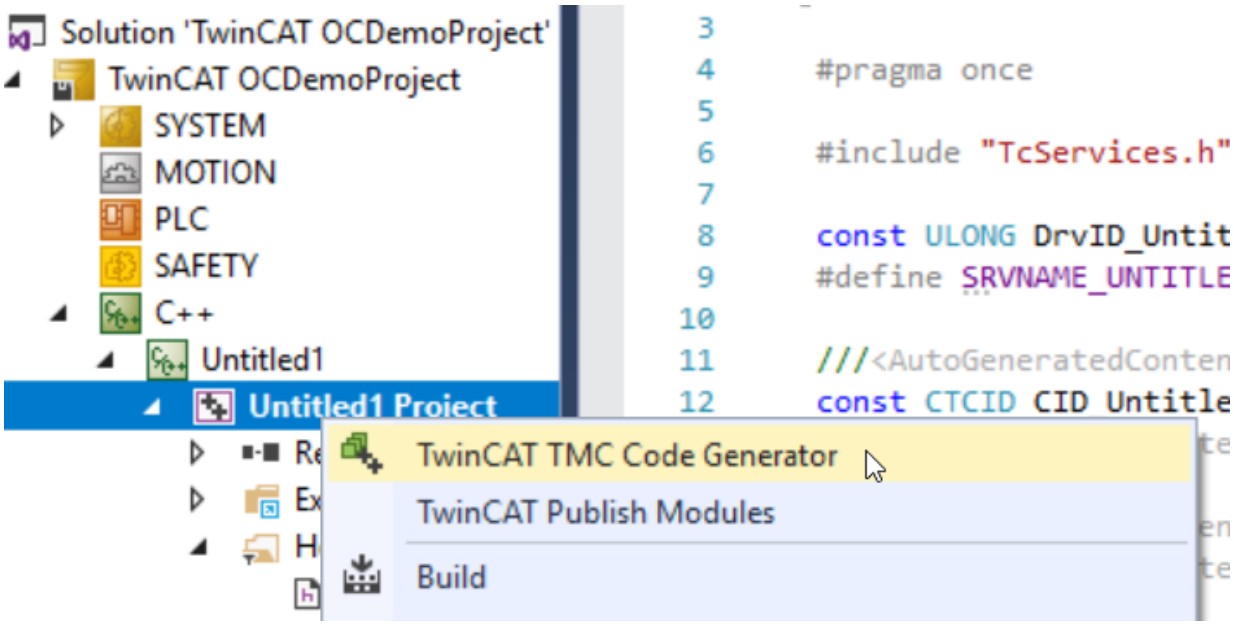
These steps are only necessary if you have previously prepared the module for Online Change.

This article describes how to change the sample project to create a version 0.0.0.2. This can later be exchanged on the target system via Online Change

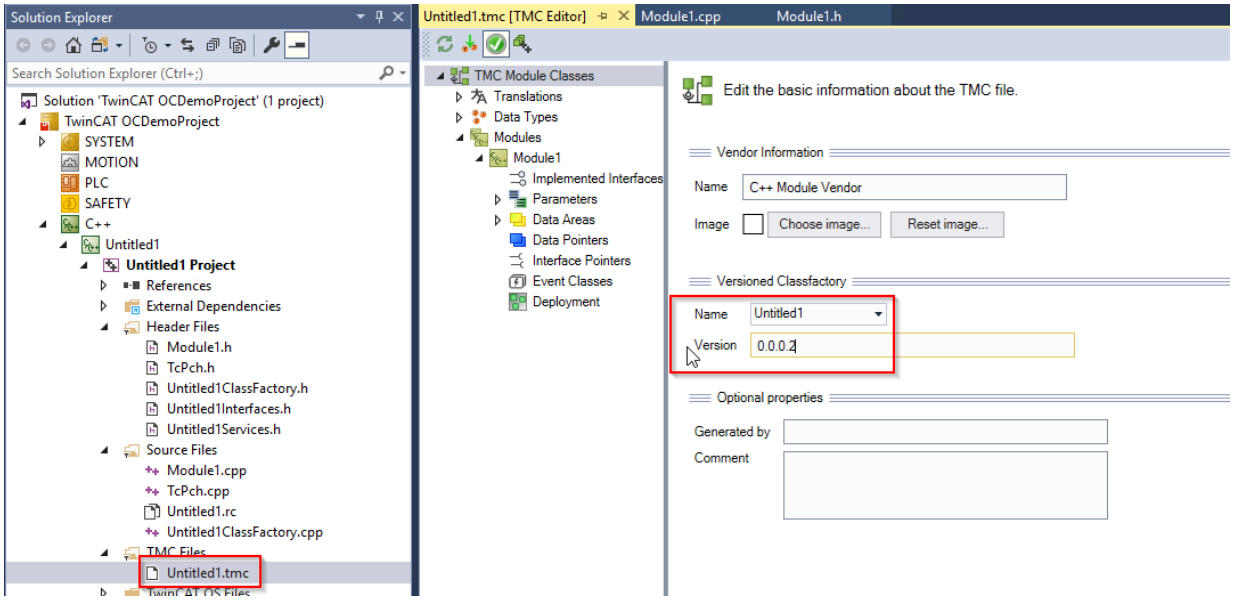
In <MyClass>.cpp (in this sample Module1.cpp) the implementation can be changed.

1. Replace a line to decrement the counter instead of incrementing it.
   Replace this line
   `m_counter++;`
2. with this
   `m_counter--;`
3. Save the modifications.

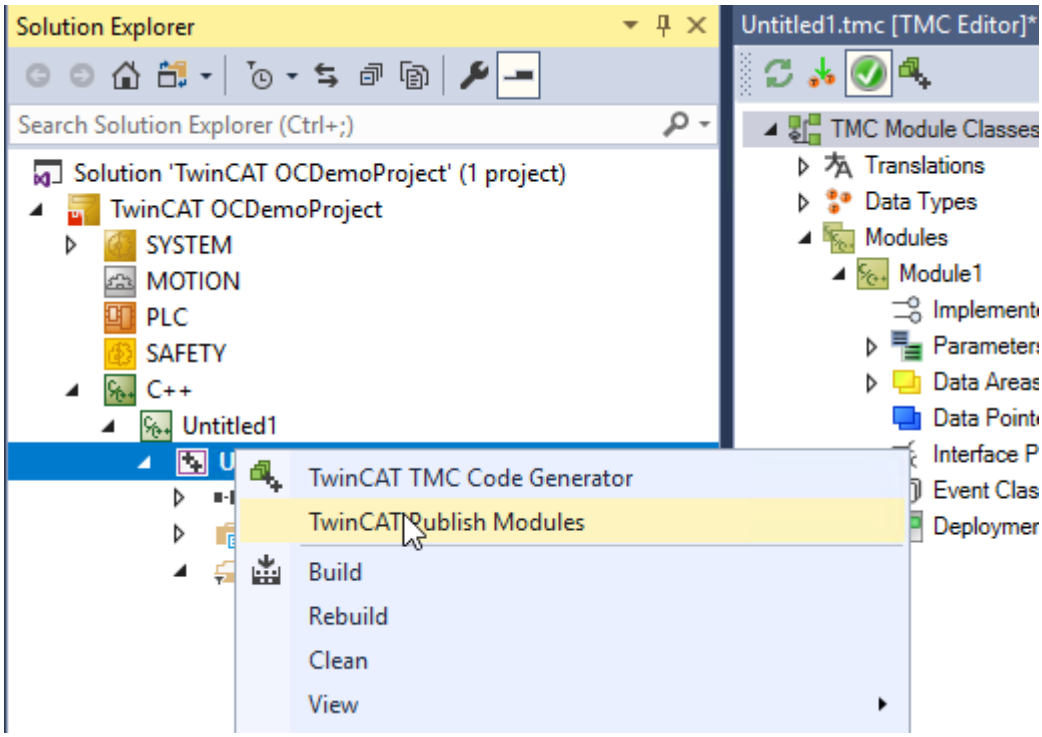
4. Start the Code Generator to take over any possible changes.



5. Parameterize version 0.0.0.2 in the TMC Editor.



6. Publish this version as well:



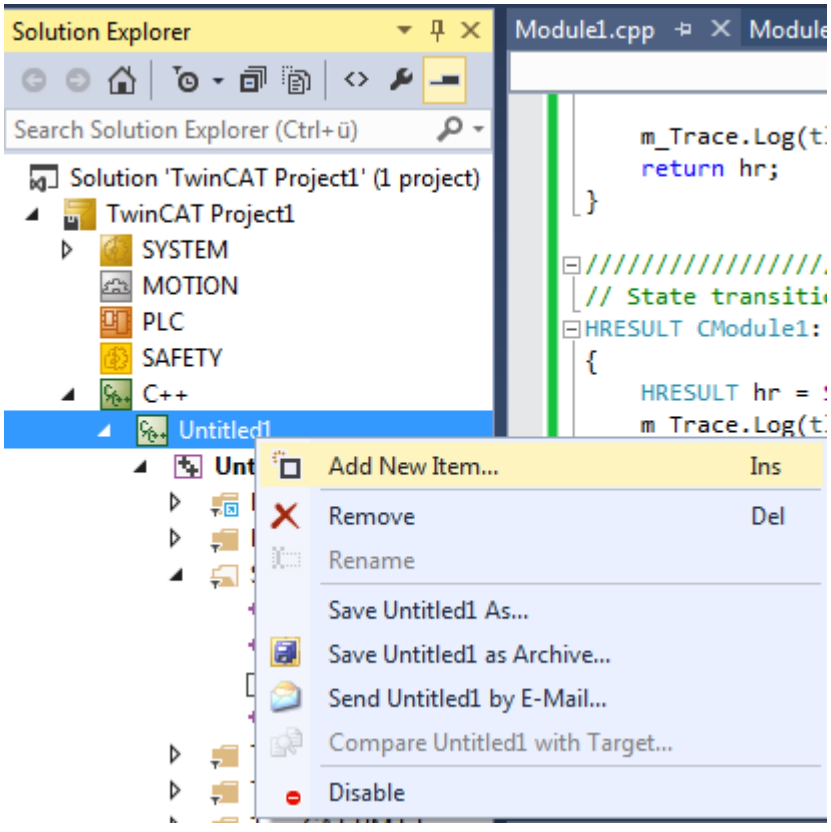
⇨ There are two versions of a module, which can be exchanged during runtime.

9.7 Create TwinCAT 3 C++ Module instance

An instance of the module must be created in order to execute it. Several instances of a module can exist.

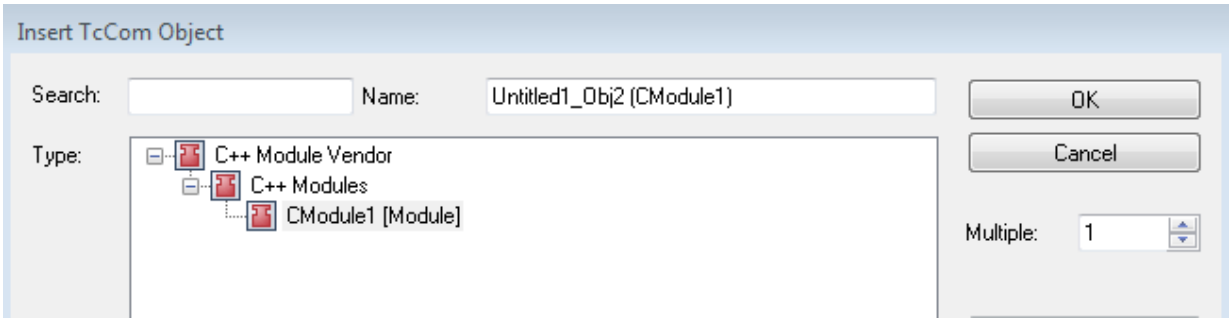
After creating a TwinCAT C++ module, open the **C++** node and follow these steps to create an instance.

1. Right-click on the C++ module (in this case "Untitled1") and select **Add New Item....**



⇨ All existing C++ modules are listed.

2. Select a C++ module. You can use the default name or alternatively enter a new instance name and confirm with **OK** (in this sample the default name was selected).



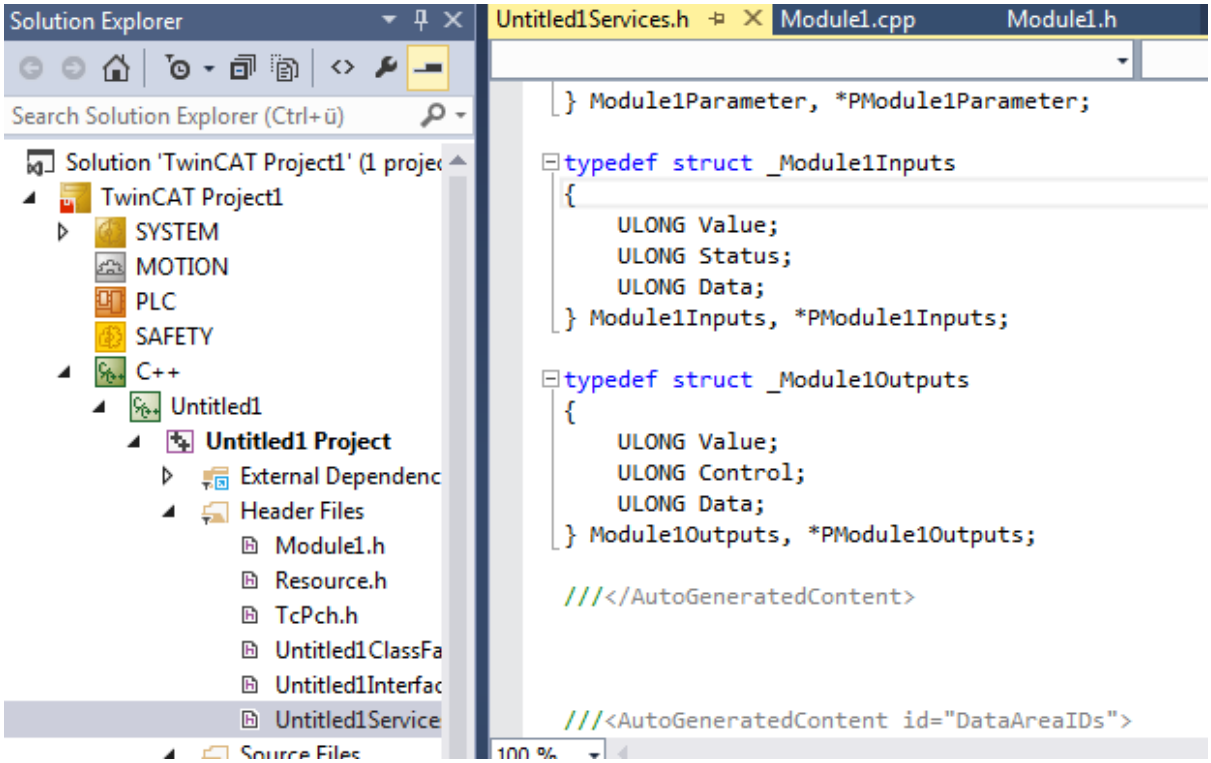
⇨ The new instance "Untitled1_Obj2 (CModule1)" becomes part of the TwinCAT 3 Solution: the new node can be found exactly under the TwinCAT 3 C++ source "Untitled1 Project".

The module already provides a simple I/O interface with 3 variables in each case:

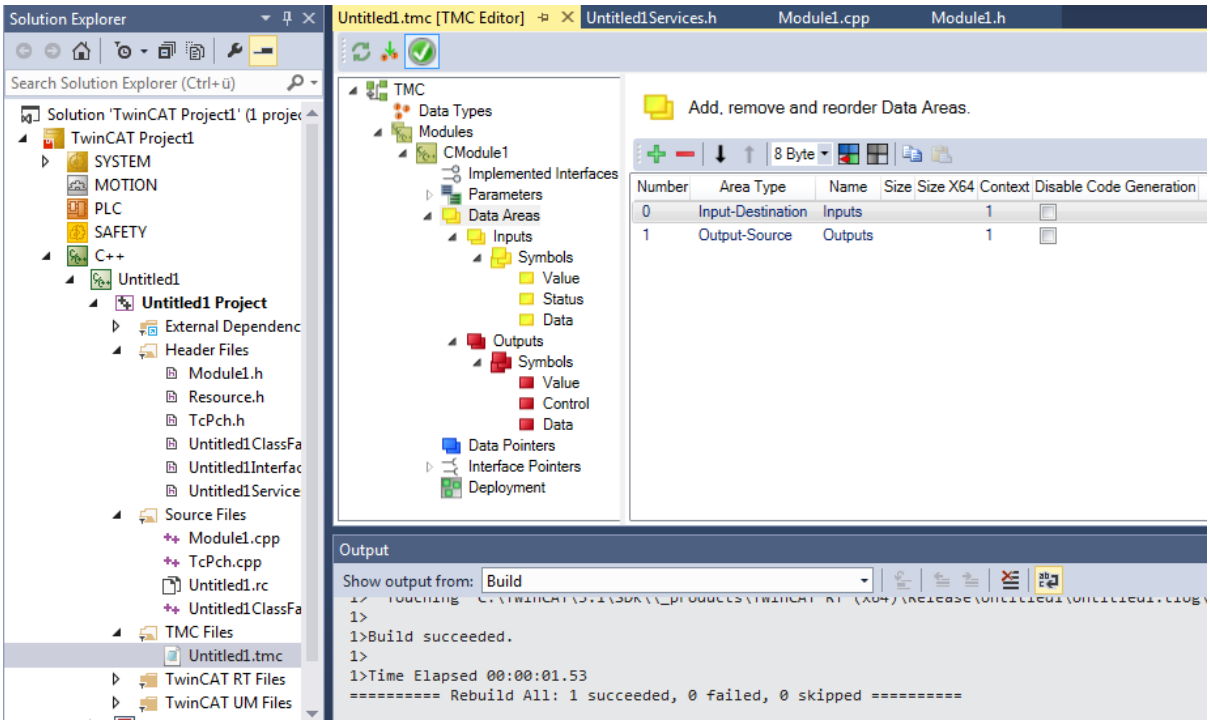
- Input Area: Value, Status, Data
- Output Area: Value, Control, Data

The description of these interfaces corresponds in two places:

- "<Classname>Services.h" (in this sample "Untitled1Services.h")



- "TwinCAT Module Configuration".tmc file (in this sample "Untitled1.tmc")



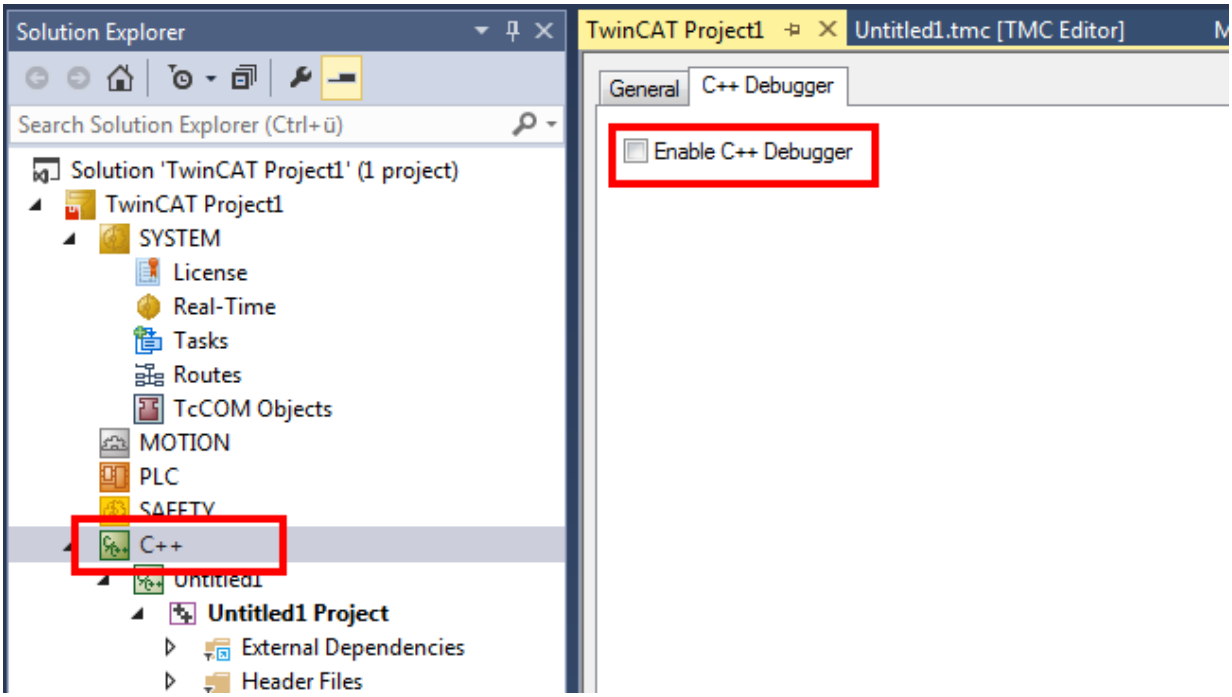
9.8 TwinCAT 3 enable C++ debugger

Beckhoff RT Linux® without debugging

This functionality is currently not available under Beckhoff RT Linux®.

To prevent all dependencies from being loaded for debugging [▶ 75], this function is switched off by default and must be activated once before the activation of the configuration.

1. Select **C++ Debugger** on the C++ node of the Solution tab.
2. Select **Enable C++ Debugger**.
3. Switch **Enable C++ Debugger** on.



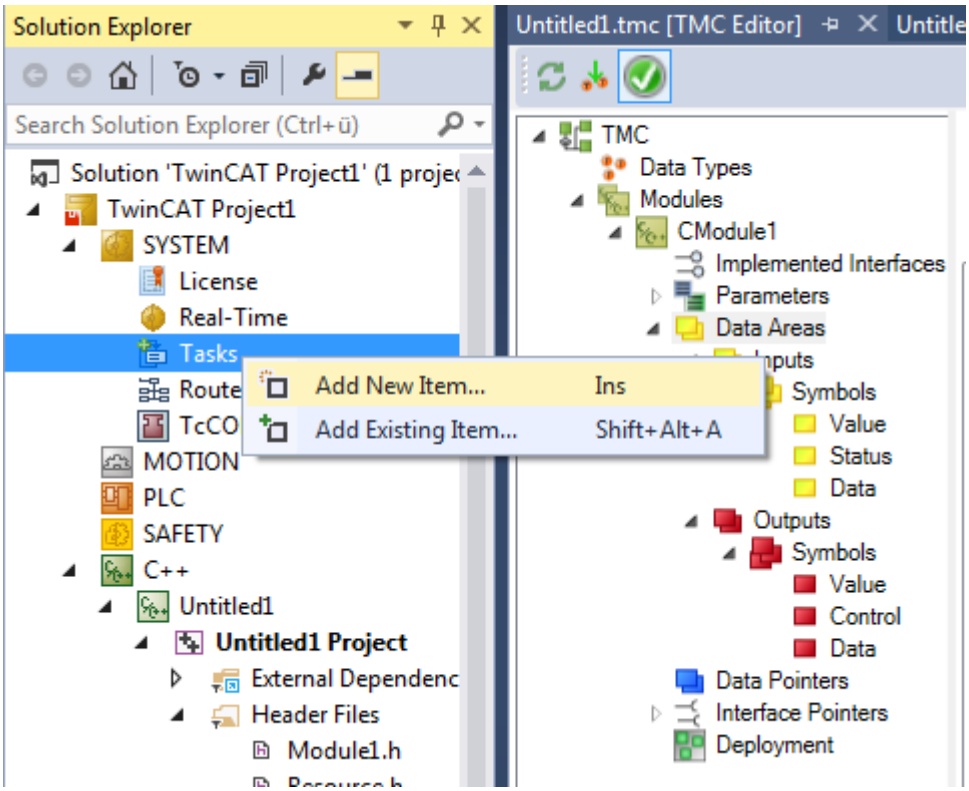
9.9 Create a TwinCAT task and apply it to the module instance

This page describes the linking of a module instance to a task, so that the cyclic interface of the module is called by the TwinCAT real-time system.

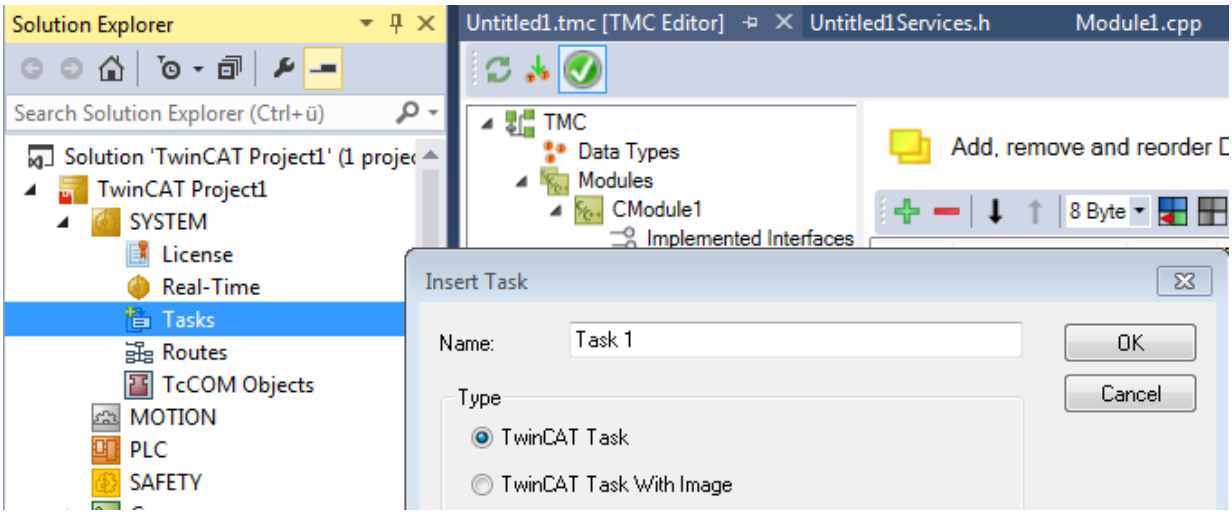
This configuration step only has to be carried out once. No new task needs to be configured for subsequent creations/new compilations of the C++ module in the same project.

Creating a TwinCAT 3 task

1. Open **System**, right-click on **Tasks** and select **Add New Item…**.

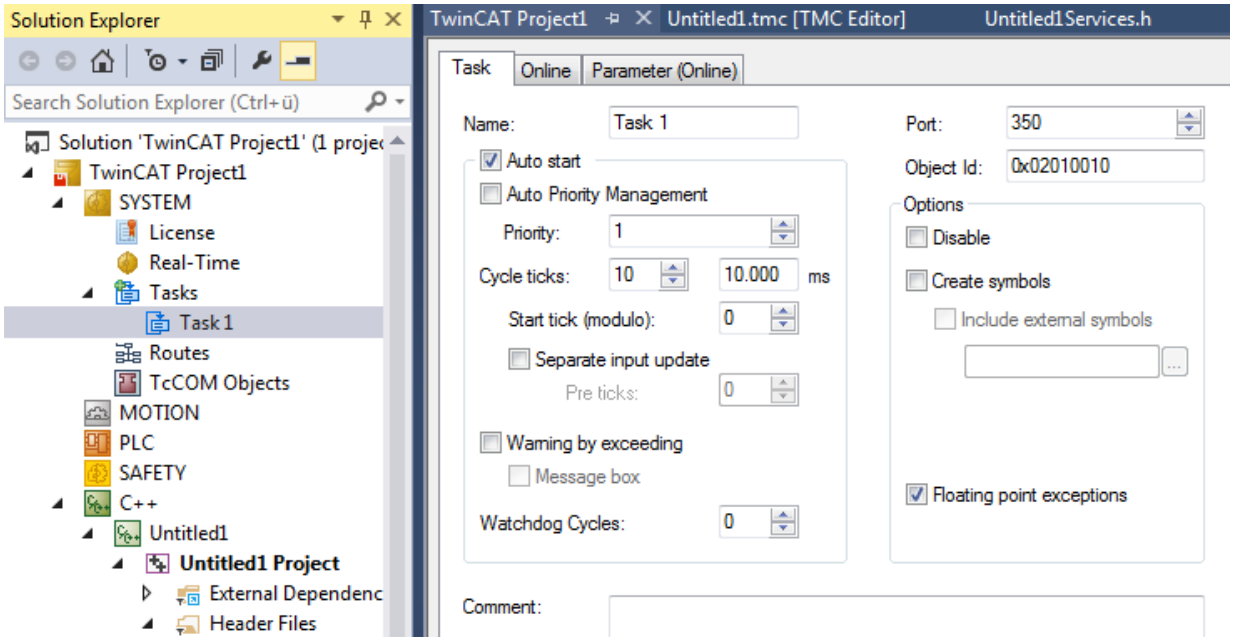


2. Enter a unique name for the task (or retain the default name).
   In this sample the I/O image interface is provided by a C++ module instance, so that no image is necessary at the task for triggering the execution of the C++ module instance.



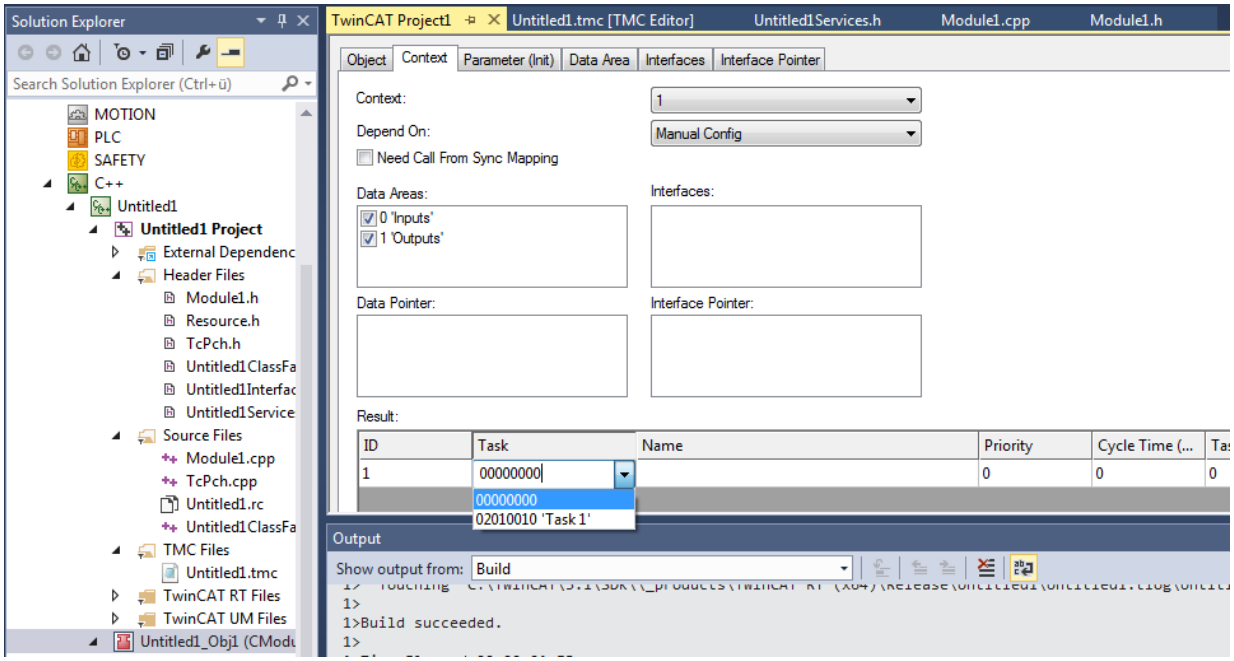
⇨ The new task with the name "Task 1" has been created.

3. The task can now be configured; double-click on the task to do this.
   The most important parameters are **Auto start** and **Priority**:
   Auto start must be activated in order to automatically start a task that is to be cyclically executed. The **Cycle ticks** define the timing of the clock in relation to the basic clock (see real-time settings).



Configuring a TwinCAT 3 C++ module instance that is called from the task

1. Select the C++ module instance in the solution tree.
2. Select the **Context** tab in the right-hand working area.
3. Select the task for the previously created context in the drop-down task menu.
   Select the default **Task 1** in the sample.

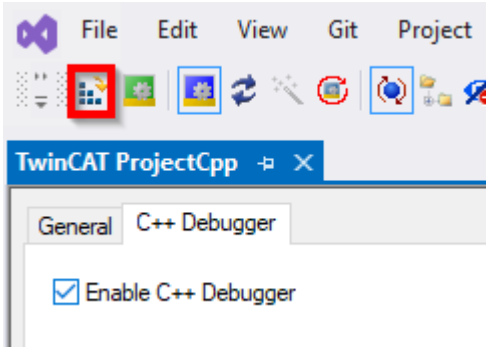


⇨ On completion of this step the **Interface Pointer** is configured as a **CyclicCaller**. The configuration is now complete!

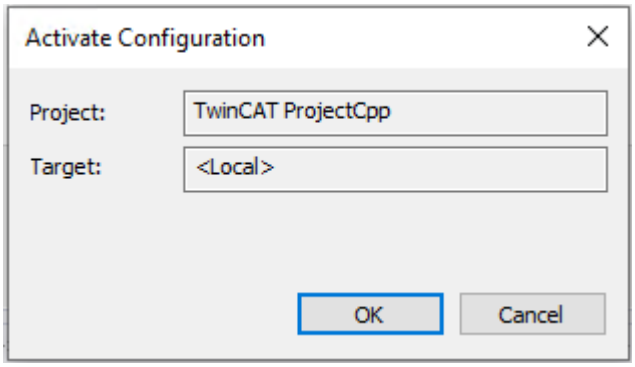
9.10 Activating a TwinCAT 3 project

Once a TwinCAT C++ project has been created, compiled and made available, the configuration must be activated:

1. Click on the symbol **Activate Configuration** – all required files for the TwinCAT project are transferred to the target system:



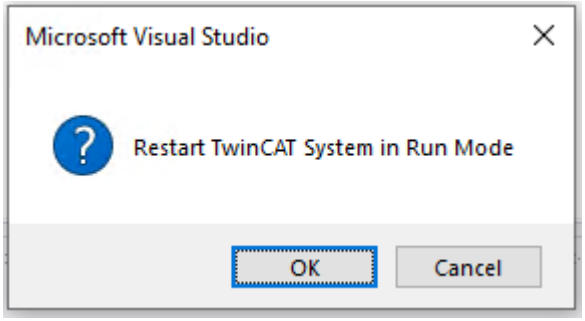
2. In the next step, confirm the activation of the new configuration. The previous old configuration will be overwritten.



⇨ The certificates required for execution are displayed and can be approved automatically:



3. If you have no license on the target system, you will be offered the option to create a 7-day trial license. This can be repeated any number of times.
4. TwinCAT 3 automatically asks whether the mode should be switched to Run mode.



⇨ With **OK** the TwinCAT 3 project changes to Run mode.
With **Cancel** TwinCAT 3 remains in **Config mode**.

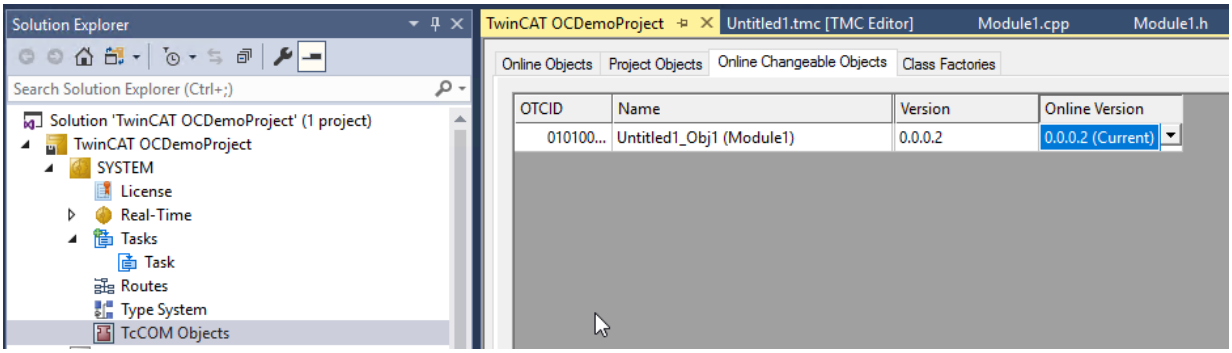
⇨ After switching to Run mode, the TwinCAT System Service symbol at the bottom in Visual Studio lights up green.



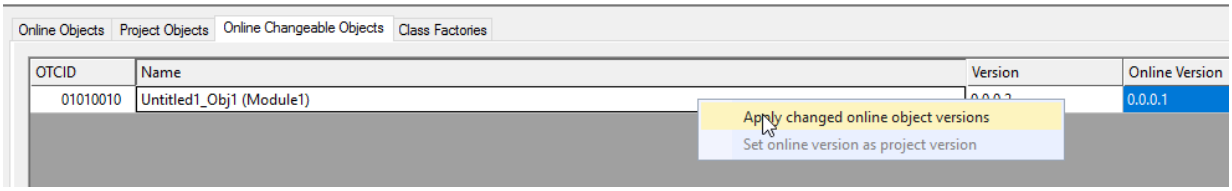
9.11 TwinCAT 3 C++ Implement project Online Change

These steps are only necessary if you have previously prepared the module for Online Change.

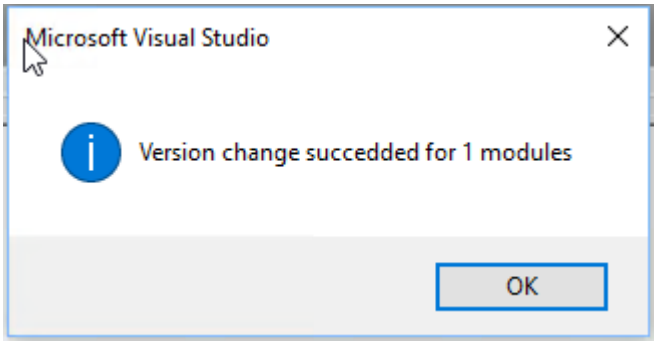
✓ Running TwinCAT C++ project as described above.

1. Switch to the **TcCOM Objects** overview of the **SYSTEM** area and there to the **Online Changeable Objects** tab.
2. 

3. In the column **Online Version** the currently running version is preselected and marked with the extension (Current).
4. Set a different version.
5. Activate this change by right-clicking and **Apply changed online object versions** on the target.



⇨ The version change was made on the target



Also see about this

Activating a TwinCAT 3 project [▶ 73]

10 Debugging

No support for Beckhoff RT Linux®

For Beckhoff RT Linux®, debugging through connections with engineering or loading dump files is currently not supported.

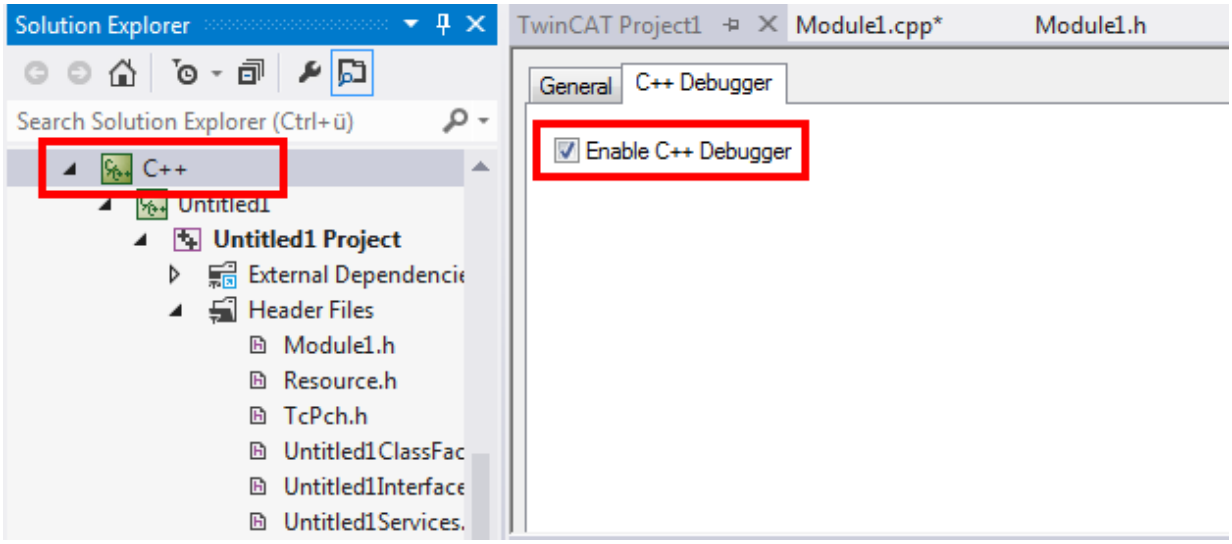
TwinCAT C++ offers various mechanisms for debugging TwinCAT C++ modules running under real-time conditions.
Most of them correspond to the mechanisms that are familiar from the normal C++ development environment. The automation environment requires additional, slightly different debugging mechanisms, which are documented here.

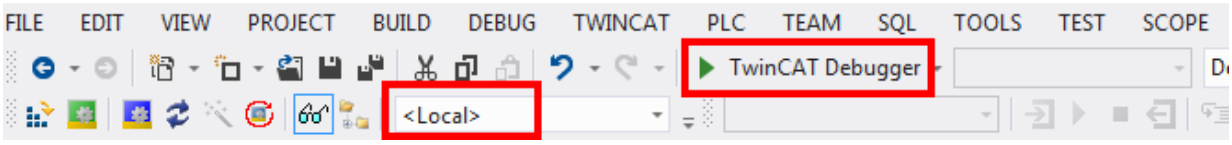
In addition we provide an overview of Visual Studio toolbox that can be used in TwinCAT 3. These were extended, so that data from the target system are displayed.

Debugging must be enabled.

This is configured via the C++ node of the solution:

1. Double-click on the C++ node and switch to the **C++ Debugger** tab to access the checkbox.



2. Make sure that you debug in the "Debug" build. Only here does the assignment of information work correctly. "Release" builds can generate incorrect displays.
3. For all debugging in TwinCAT C++, connect the TwinCAT Engineering with the runtime system (XAR) via the **TwinCAT Debugger** button.
4. 

Breakpoints and step-by-step execution

In most cases when debugging a C++ program, breakpoints are set and the code is then executed step by step while observing the variables, pointers, etc.

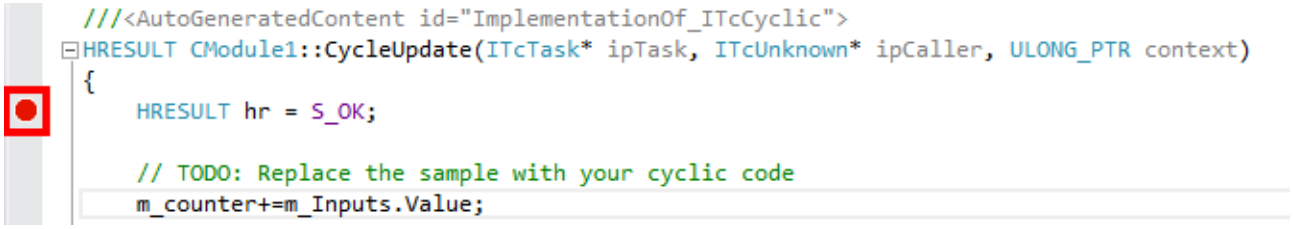
In the context of the Visual Studio debugging environment, TwinCAT offers options to run real-time-executed code step by step. To set a breakpoint, you can navigate through the code and click on the gray column on the left adjacent to the code or use the hotkey (normally F9).

⚠ WARNING

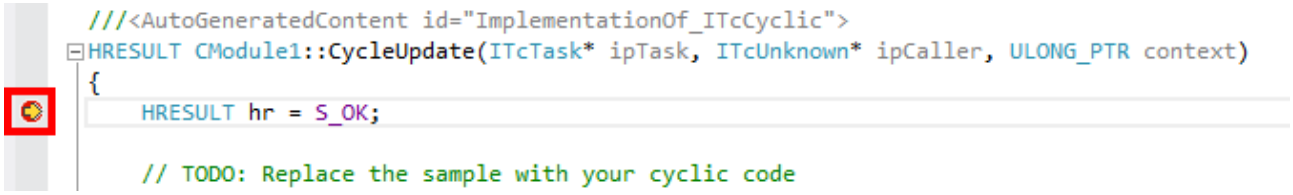
Damage to plants and personal injuries due to unexpected behavior of the machine / plant

Breakpoints change the behavior of the machine or plant. Depending on the machine being controlled, the machine or workpieces may be damaged or the health and life of people may be endangered.

Make sure that the changed behavior of the controlled system does not cause any damage and be sure to note the plant documentation.



On reaching the breakpoint (indicated by an arrow), the execution of the code is stopped.



The code is executed step by step by pressing **Step Over** (Debug menu, toolbar or hotkey F10). The familiar Visual Studio functions **Step in** (F11) and **Step out** (Shift + F11) are also available.

Conditional breakpoints

A more advanced technology allows the setting of conditional breakpoints – the execution of code is only stopped at a breakpoint if a condition is fulfilled.

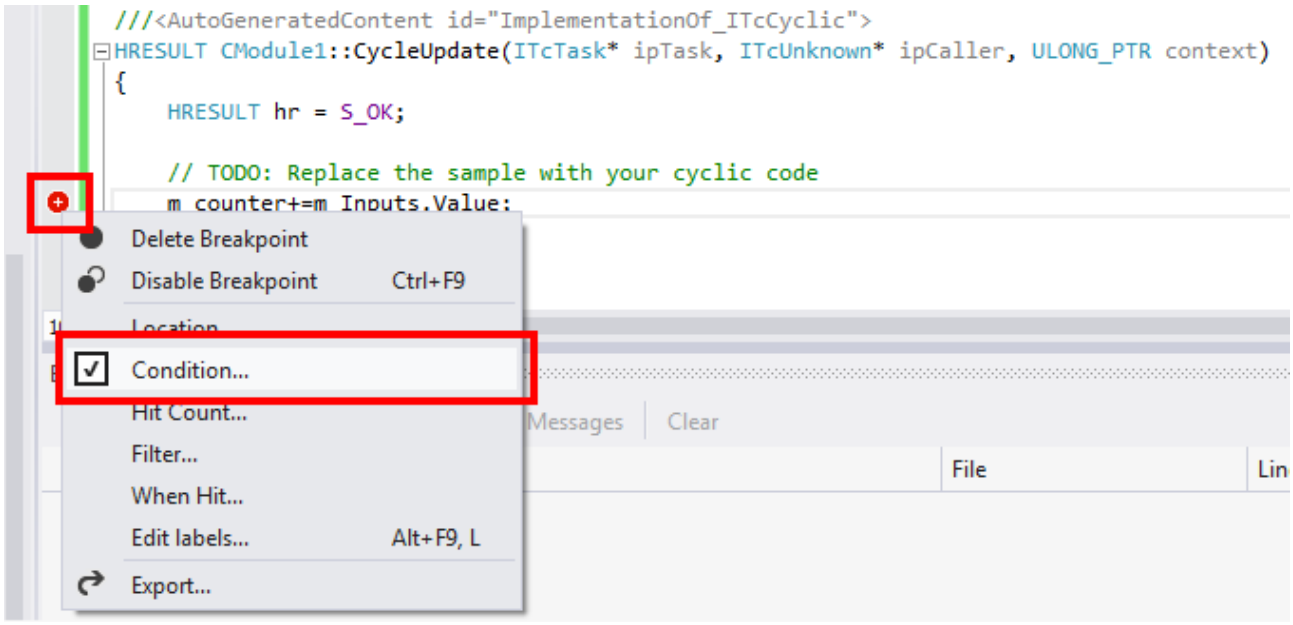
TwinCAT offers the implementation of a conditional breakpoint as part of the Visual Studio Integration. To set a condition, first set a normal breakpoint and then right-click on the red dot in the breakpoint column.

⚠ WARNING

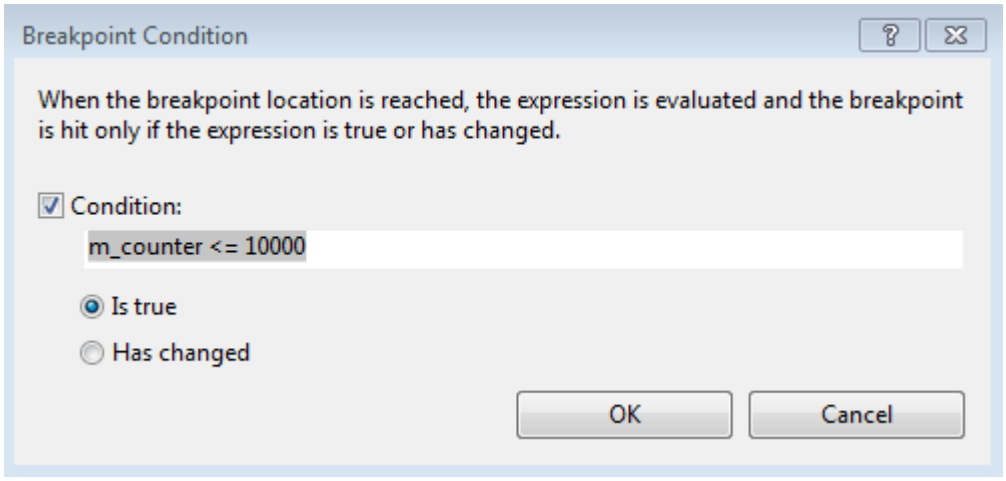
Damage to plants and personal injuries due to unexpected behavior of the machine / plant

Breakpoints change the behavior of the machine or plant. Depending on the machine being controlled, the machine or workpieces may be damaged or the health and life of people may be endangered.

Make sure that the changed behavior of the controlled system does not cause any damage and be sure to note the plant documentation.



Select **Condition…** to open the condition window:



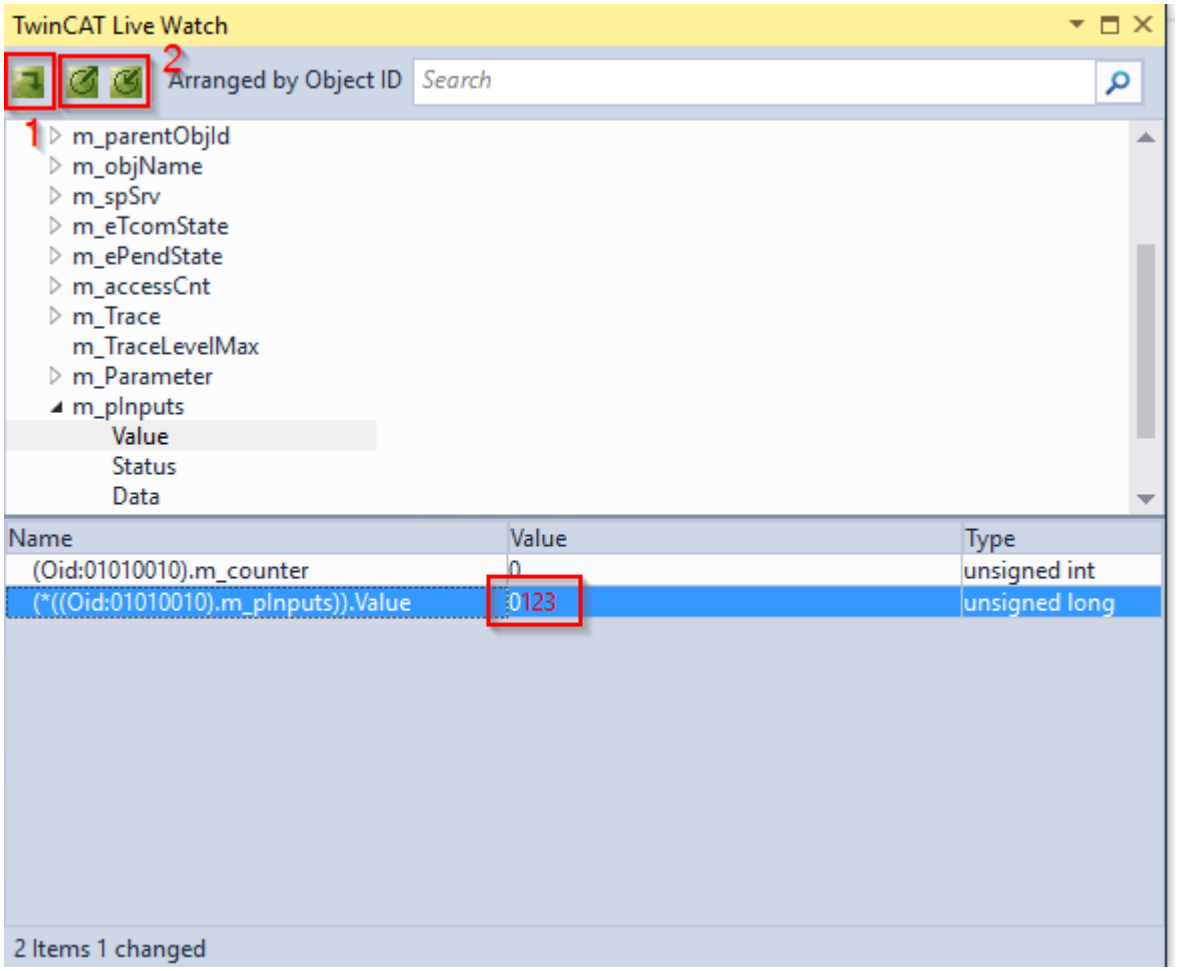
For more information, see the chapter: Details of Conditional Breakpoints [▶ 78].

Live Watch

When engineering and developing machines, it is not always advisable to stop the system at a breakpoint because this will affect the behavior.
The TwinCAT PLC projects offer an online view and handling of the variables in the RUN state, without having to interrupt the real-time.

TwinCAT C++ projects offer a similar behavior for C++ code via the **Live Watch** window.

The **Live Watch** window can be opened via **Debug->Windows->TwinCAT Live Watch**.
To open the window, first establish a connection with the real-time system (press the **TwinCAT Debugger** button), whereupon Visual Studio switches to the debug view, otherwise no data can be provided.



The TwinCAT Live Watch window is divided into two areas.

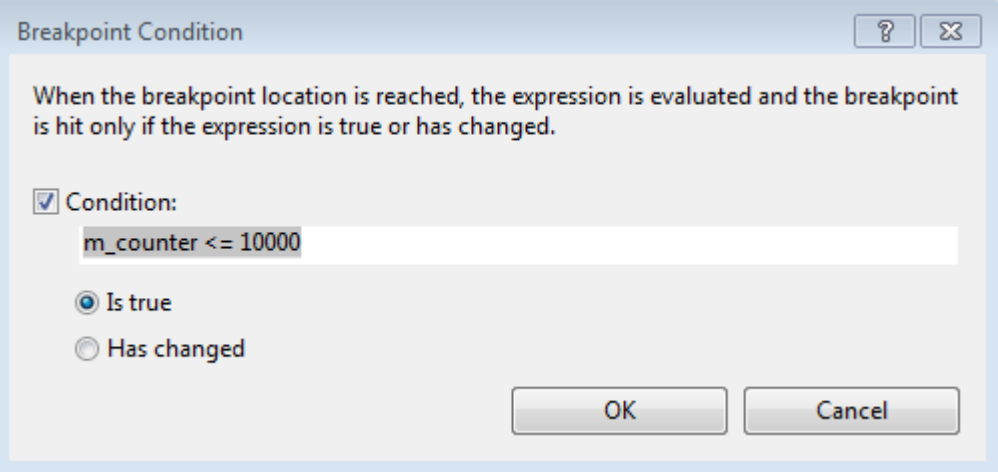
In the upper area all member variables can be explored. By double-clicking on it they are added to the lower area, where the current value is then displayed.

You can edit these values by clicking on the value in the **Value** field. The new value is highlighted in red. To write the value, press the symbol in the upper left corner (1).

Using the import and export symbols under (2), the selected member variables can be saved and later restored.

10.1 Details of Conditional Breakpoints

TwinCAT C++ provides conditional breakpoints. Details of the formulation of these conditions can be found here.



Unlike the Visual Studio C++ conditional breakpoints, the TwinCAT conditions are compiled and subsequently transferred to the target system so that they can be used during short cycle times.

⚠ WARNING

Damage to plants and personal injuries due to unexpected behavior of the machine / plant

Breakpoints change the behavior of the machine or plant. Depending on the machine being controlled, the machine or workpieces may be damaged or the health and life of people may be endangered.

Make sure that the changed behavior of the controlled system does not cause any damage and be sure to note the plant documentation.

The option buttons offer two options that are described separately.

Option: Is true

Conditions are defined with the help of logical terms, comparable to the conjunctive normal forms.
They are formed from a combination of Maxterms connected with "&&":

```
(Maxterm1 && Maxterm2 && ... && MaxtermN)
```

wherein each Maxterm represents a combination of || associated conditions:

```
(condition1 ||condition2 || ... || conditionN )
```

Possible comparison operators: `==`, `!=`, `<=`, `>=`, `<`, `>`

Observe the Live Watch window for the determination of the available variables. All listed variables can be used for the formulation of conditions. This includes TMC-defined symbols as well as local member variables.

Samples:

```
m_counter == 123 && hr !=  0
```

```
m_counter == 123 || m_counter2 == 321 && hr == 0
```

```
m_counter == 123
```

Monitoring module instances

The OID of the object is stored in `m_objId`, so the monitoring of the OID can look like this `m_objId == 0x01010010`

Monitoring of tasks

A special variable `#taskId` is provided to access the OID of the calling task. E.g.: `#taskID == 0x02010010`

Option: Has changed

The option “Has changed“ is simple to understand: By providing variable names, the value will be monitored and execution will be held, if the value has changed from the cycle before.

Samples:

```
m_counter
```

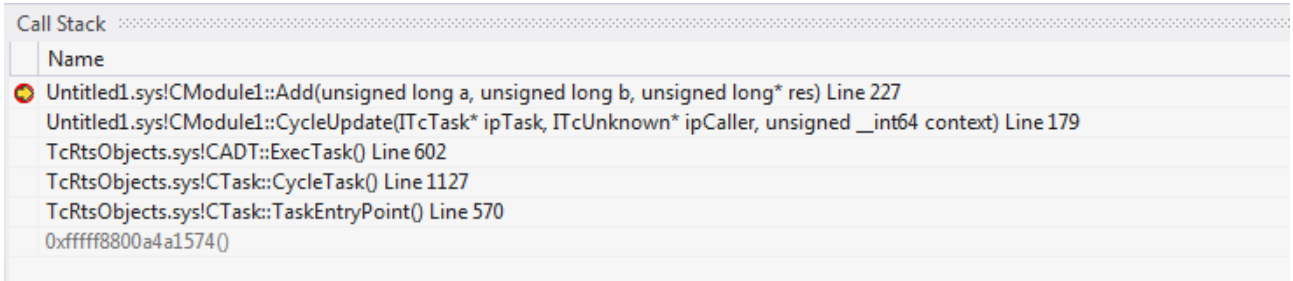
```
m_counter && m_counter2
```

10.2 Visual Studio tools

Visual Studio makes the usual development and debugging tools available for C++ developers. TwinCAT 3 extends these Visual Studio tools, so that debugging of C++ code that runs on a target system is also possible in TwinCAT 3 Engineering with the Visual Studio tools.
The corresponding advanced tools are briefly described here. If the corresponding windows are not visible in Visual Studio, they can be added via the menu item **Debug ->Windows**. The menu is context-dependent, i.e. many of the windows described here only become configurable once a debugger is linked to a target system.

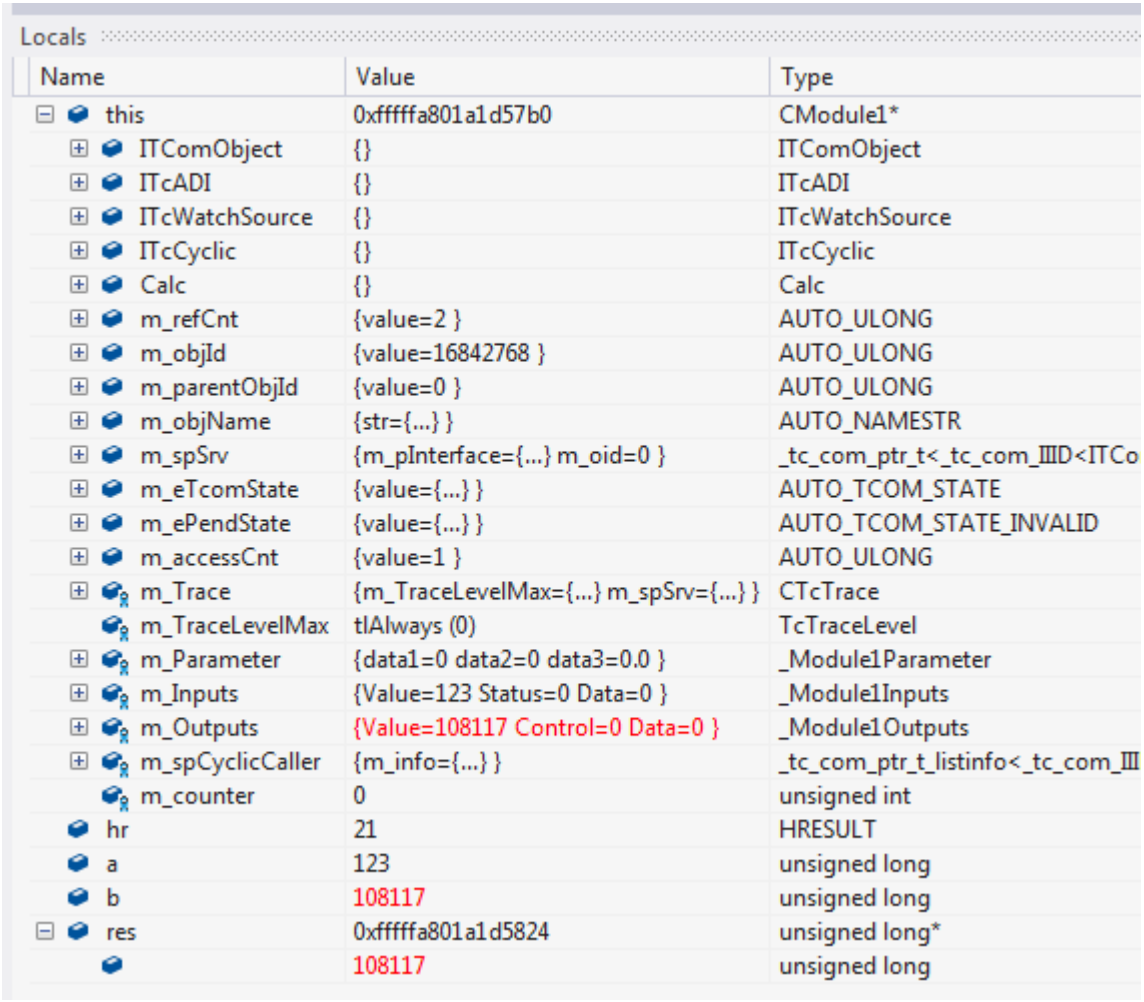
Callstack

The Call Stack is displayed by the **Call Stack** tool window when a breakpoint has been reached.

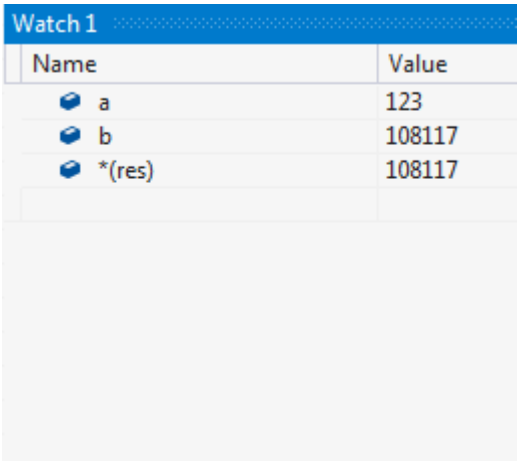


Autos / Locals and Watch

The corresponding variables and values are displayed in the **Autos** / **Locals** window when a breakpoint is reached. Changes are shown in RED.

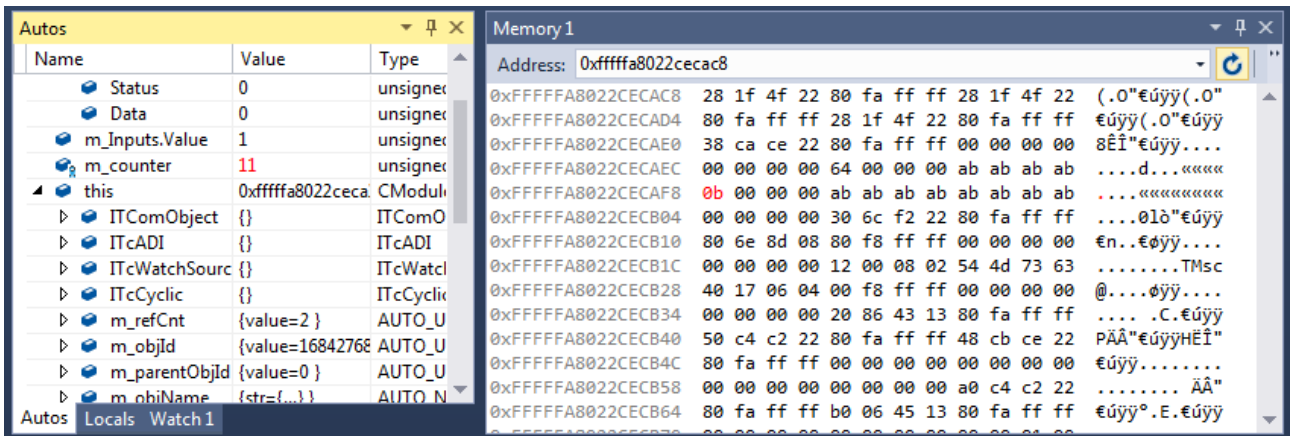


From here, the values can be applied to the **Watch** windows by right-clicking:



Memory View

The memory can be monitored directly. Changes are shown in RED.



10.3 Working with TwinCAT Task Dumps

From TwinCAT 3.1 Build 4026

This function is available from TwinCAT 3.1 Build 4026.

If an exception occurs, TwinCAT runtime automatically writes "TwinCAT Task Dumps" to the boot directory (usually C:\ProgramData\Beckhoff\TwinCAT\3.1\Boot) if it is not handled by the application.

These Task Dumps can be transferred and evaluated by the TwinCAT C++ developer on a development system with Visual Studio for TwinCAT C++ development.

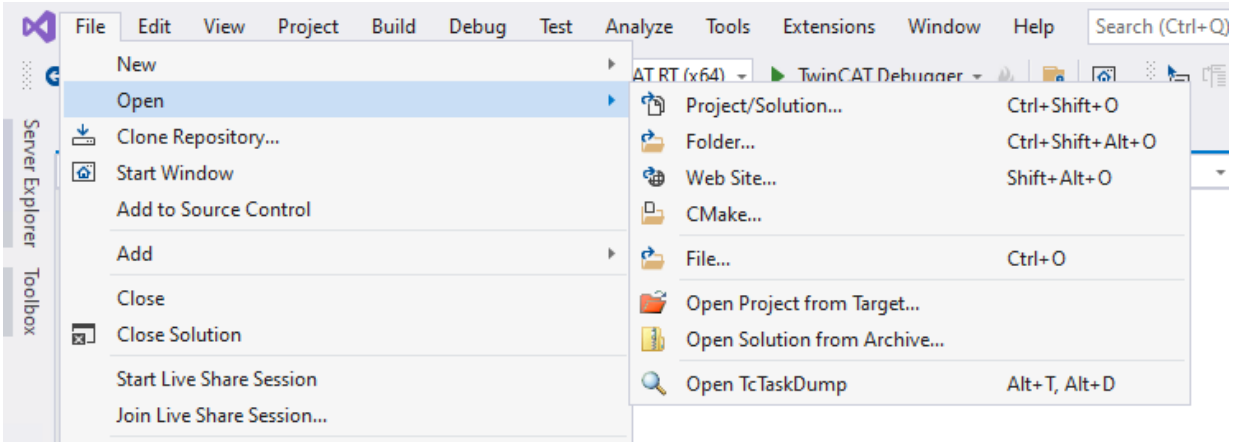
For the investigation, it is necessary that, in addition to the TwinCAT task dumps, the exact corresponding TMX and PDB files are also available.

If the TwinCAT Task Dump is loaded directly from the appropriate Solution, these files are found automatically. If the TwinCAT Task Dump is loaded from an empty Solution, the associated TMX and PDB files can be placed in the same directory as the TwinCAT Task Dump. Alternatively, it is possible to enter a search path in the Visual Studio options (Tools->Options->Debugging->Symbols: add new Path to Symbol file (.pdb) locations).

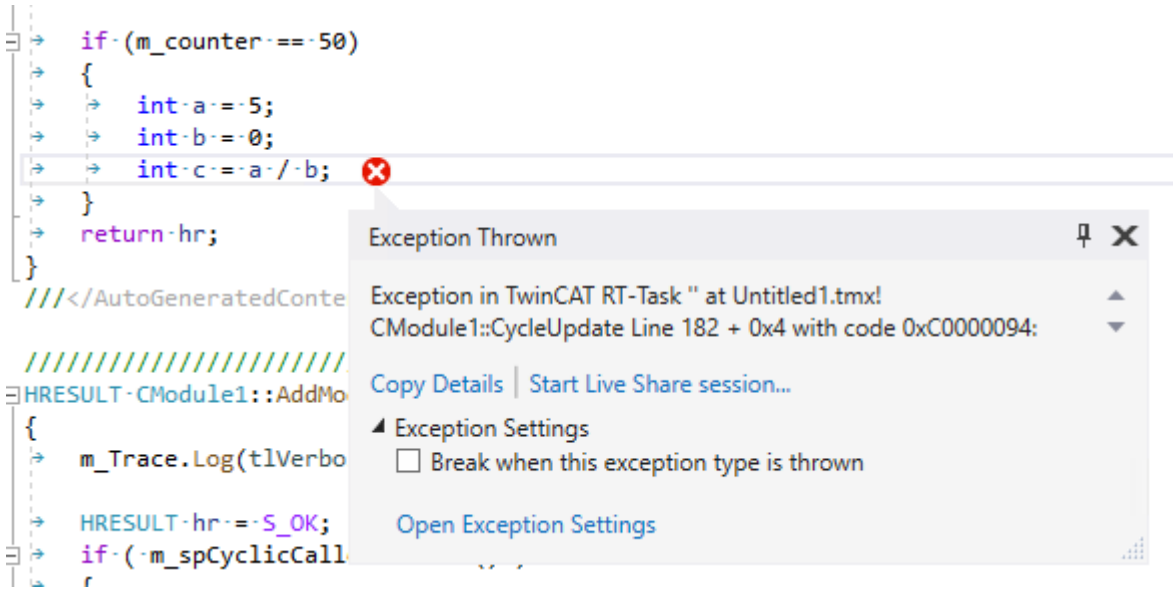
The procedure is then as follows:

✓ An exception has occurred on the target system.

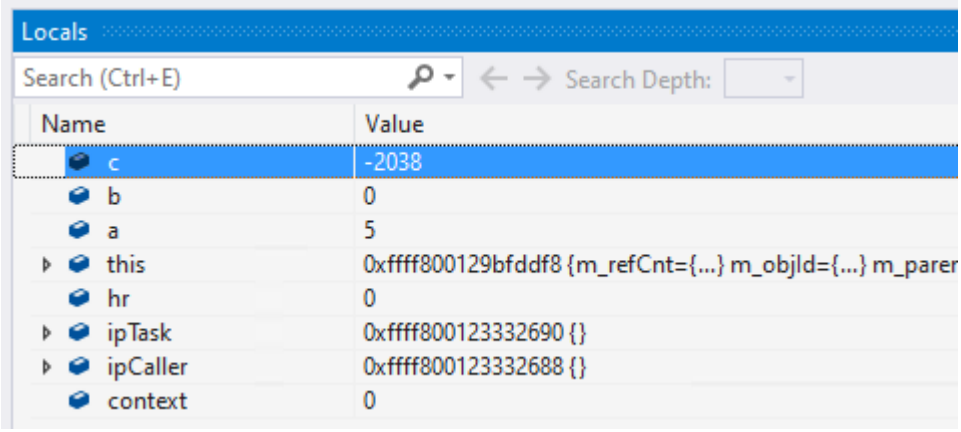
1. The TwinCAT Task Dump file is transferred from the target system to the engineering system. It has a name like taskmemory-*.dmp
2. Ideally, the appropriate sources of the TwinCAT project are available and are opened in Visual Studio.
3. The file can be opened via File->Open->Open TcTaskDump:



4. The editor jumps directly to the appropriate code location that triggered the exception:



In addition, standard Visual Studio windows can be used to control variable values, for example:



10.4 Examining dump files using WinDbg

TwinCAT 3.1 Build 4026.21 required

The function described here requires at least TwinCAT 3.1 4026.21

For analyzing dump files, additional information is added to task dumps and kernel dumps from Windows to facilitate the analysis.

The analysis can be carried out using the WinDbg tool, for which an extension is provided that can extract the following information:

- Listing the loaded TMX drivers
- Reloading the symbols of all loaded TMX modules
- Restoring the crash location of a TwinCAT real-time task
- Export of logging data for the real-time monitor.
  The recording of this data must be activated in the System Manager via **System/Real-Time /Debugging/Realtime Logging**.

Beckhoff recommends using the classic WinDbg version from Microsoft, as undesirable behavior has occasionally been observed with the newer version (e.g. from the Windows AppStore): https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/debugger-download-tools

The extension can be installed using the following command:

```
tcpkg install TwinCAT.XAE.WinDbgExtension
```

It is installed appropriately so that WinDbg recognizes it directly.

After starting WinDbg, the extension is loaded by entering it in WinDbg:
`.load TcDbgExt`

The command
`!tc`
is then available, which outputs a help text with the functions described above.

Sample output

```
kd> .load TcDbgExt
kd> !tc
Beckhoff TwinCAT WinDBG Extension:
available commands:
!tc list modules
    Lists the above elements of runtime.
    Use -h to get specific information for list.
!tc reload
    Reloads the symbols of all loaded TwinCAT modules.
    The command can operate without XAR debug symbols,
    if TwinCAT.XAR.Realtime 2.0.60 or later is used.
!tc ctx
    restore twincat exception context for twincat bugchecks
(code=BE00XXXX)
    the command can operate without symbols for XAR-Runtime,
    if TwinCAT.XAR.Realtime 2.0.60 or later is used.
!tc wrlog <project name> [--path <path>]
    Save TE1010 TwinCAT 3 Realtime Monitor project with
    recorded data to file(s) with name <project name>    TwinCAT.XAR.Realtime
2.1.1 or above is required
    for this command to work.
!tc help <cmd>
    Print this help text. If <cmd> is set, the help request is    forwared to
the given command.
```

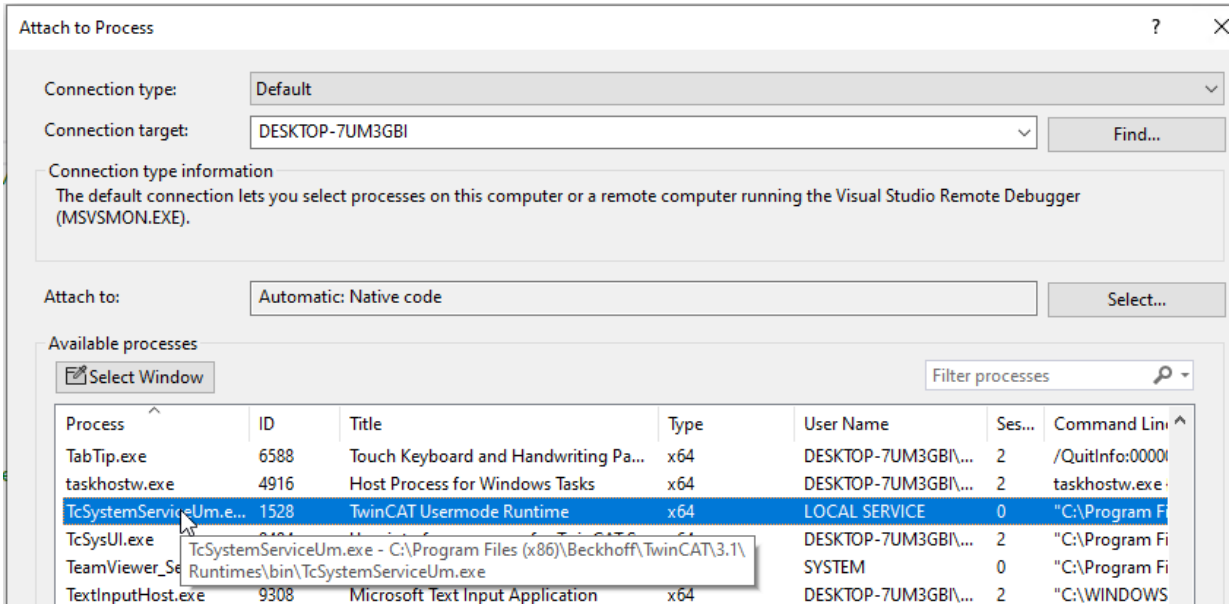
10.5 Debugging the state machine with the Usermode Runtime

The state machine of the TcCOM modules cannot be debugged in the real-time environment, as the debugger itself is still being started at that point.
The Usermode Runtime can be used for this.

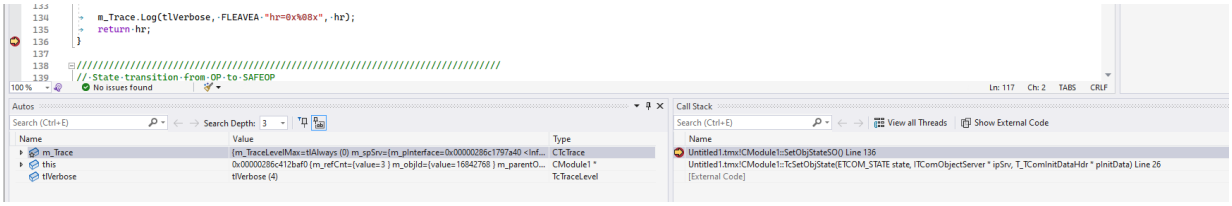
✓ A TwinCAT C++ project is loaded in Visual Studio

1. Activate the project in a TC1700 Usermode Runtime on Windows.
2. Open a second Visual Studio instance

3. Use the normal Visual Studio C++ debugger to connect to the Usermode Runtime process. Debug -> Attach to Process ... -> Selection of TcSystemServiceUm.exe



4. In this instance, add the TwinCAT C++ project that you want to debug. Set breakpoints in the transitions.
5. Start the Usermode Runtime - the breakpoints are reached.



11 Wizards

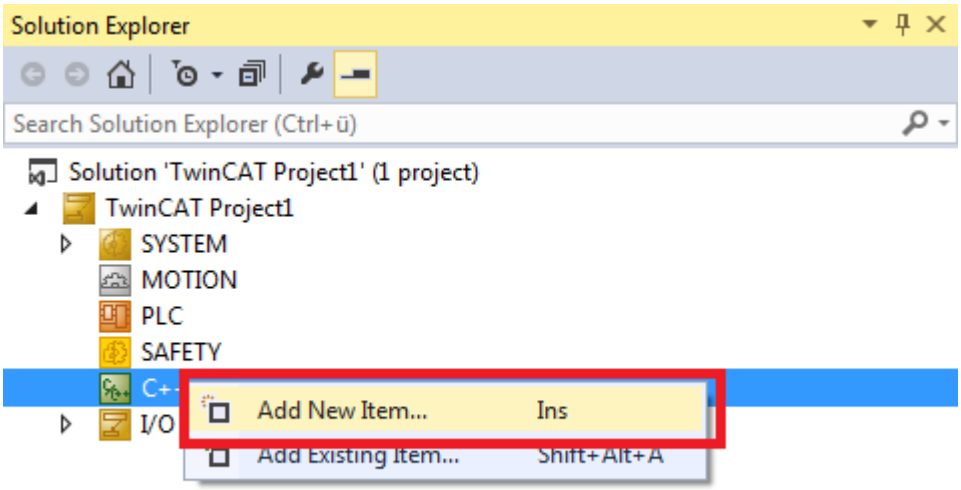
Wizards are available to facilitate familiarization with the engineering procedures of the TwinCAT C++ system.

- The TwinCAT Project Wizard [▶ 85] creates a TwinCAT C++ project. For driver projects the TwinCAT Class Wizard is then started.
- The TwinCAT Module Class Wizard [▶ 86] is started automatically when a C++ module is created. This wizard provides various "ready-to-use" projects as a starting point for your own developments.
- The TwinCAT Module Class Editor [▶ 88] (TMC) is a graphical editor for defining data structures, parameters, data areas, interfaces and pointers. It creates a TMC file that is used by the TMC Code Generator.
- Instances are generated based on the defined classes, which can be configured via the TwinCAT Module Instance Configurator [▶ 129].

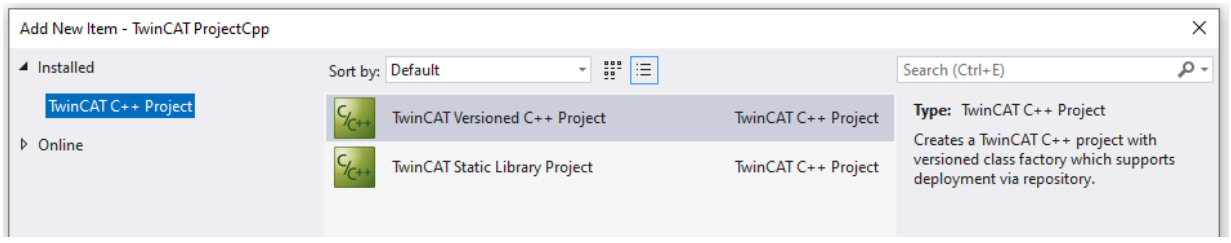
11.1 TwinCAT C++ Project Wizard

After the creation of a TwinCAT project you can add a C++ project with the help of the TwinCAT C++ Project Wizard:

1. Right-click on the C++ icon and select **Add new Item...** to start the C++ project wizard.



TwinCAT offers two C++ projects:
- Versioned C++ projects: Projects that involve versioning may also have the option of switching between versions at runtime.
- Static library: Projects with C++ functions that are used by (different) TwinCAT C++ drivers.



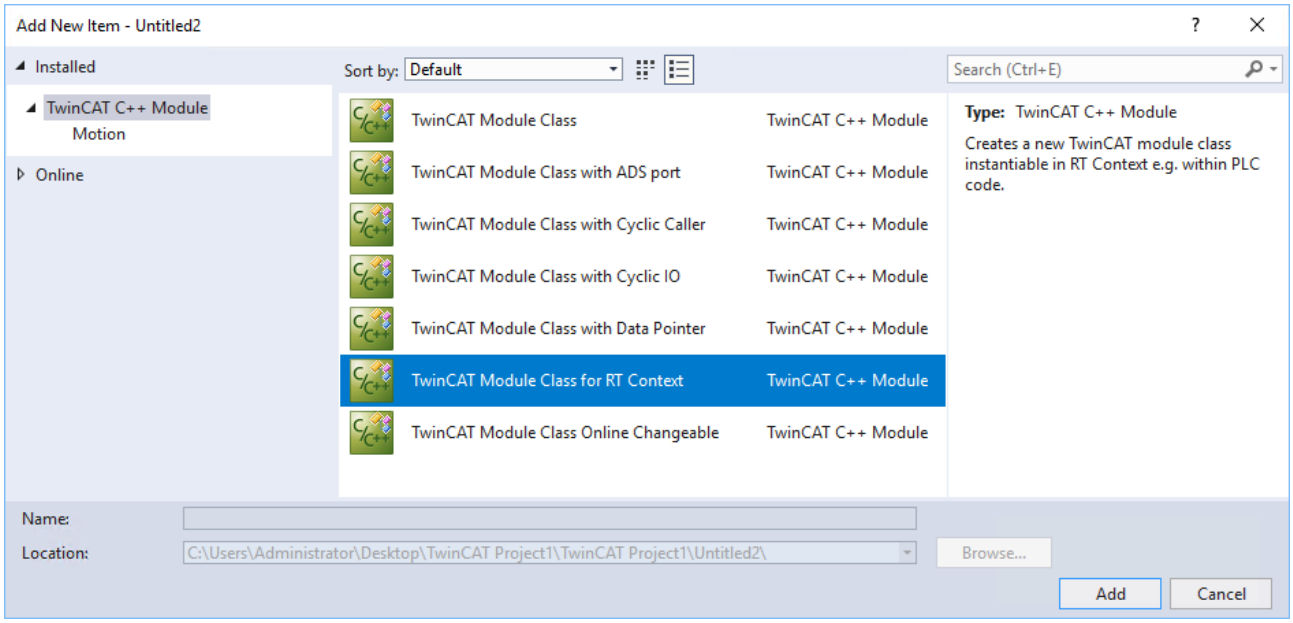
2. Select one of the project templates and specify a name and location.
   ⇨ The TwinCAT C++ project is created

⇨ In the case of a driver, the TwinCAT C++ Class Wizard [▶ 86] is started.

11.2 TwinCAT Module Class Wizard

TwinCAT 3 offers various class templates that can be used to create TcCOM object classes:

- TwinCAT Module Class
- TwinCAT Module Class with ADS port
- TwinCAT Module Class with cyclic caller
- TwinCAT Module Class with cyclic input/output
- TwinCAT Module Class with data pointer
- TwinCAT Module Class for real-time context
- TwinCAT Module Class with Online Changeable capability



TwinCAT Modules Class

Creates a new TwinCAT module class.

This is a template generates a basic core module. It has no cyclic caller and no data area, instead it's good as a start point for implementing services called on demand from a caller.

For example when creating a C++ method that will be called from a PLC module or another C++ module.

See Sample11 [▶ 257]

TwinCAT Module Class with ADS port

This template offers the C++ module as well as the functionality of an ADS server and ADS clients.

- ADS server:
  Can run as a single instance of this template of the C++ module and can be preconfigured with a specific ADS port number (e.g. 25023).
  Enables several instances of this template, whereby each C++ module is assigned its own unique ADS port number by TwinCAT 3 (e.g. 25023, 25024, 25025, ...).
  The ADS messages can be analyzed and processed thanks to the implementation of the C++ module. ADS handling for accessing input/output data areas does not have to be implemented using its own ADS Message Handling.
- ADS Client:
  This template provides sample codes to initiate an ADS call by sending an ADS message to an ADS partner.

Since the modules behave like ADS clients or ADS servers that communicate with each other via ADS messages, the two modules (Caller=Client and Called=Server) can run in the same or different real-time contexts on the same or different CPU cores.
Because ADS can work across networks, the two modules can also run on different machines in the network.

See Sample03 [▶ 237], ADS Communication [▶ 189]

TwinCAT Module Class with cyclic caller

Enables the cyclic call of a C++ program which is cut off from the outside world.

Not often used, a module class with cyclic caller and cyclic I/O is preferred.

TwinCAT Module Class with cyclic input/output

Creates a new TwinCAT module class, which implements the cyclically calling interface and has an input and output data area.

The input and output data areas can be linked with other input/output images or with physical I/O terminals.

Important:

The C++ module has its own logical input/output data storage area. The data areas of the module can be configured with the System Manager.

If the module is mapped with a cyclic interface, copies of the input and output data areas exist in both modules (the caller and the called). In this way, the module can run under a different real-time context and even on another CPU core in relation to another module.
TwinCAT will continuously copy the data between the modules.

See Quick Start [▶ 58], sample 01 [▶ 236].

TwinCAT Module Class with data pointer

Just like the TwinCAT Module Class with Cyclic IO, this template also generates a new TwinCAT module class that implements a calling interface with an input and output data area for linking with other logic input/output images or with physical I/O terminals.

In addition, this template provides data pointers that can be used to access data areas from other modules via pointers.

Important:

Unlike in the case of the cyclic I/O data area, where the data is copied cyclically between modules, in the case of the use of C++ data pointers there is only a single data area and this belongs to the destination module. When writing from another C++ module to the destination module via the data pointer mechanism, this will immediately affect the data area of the destination module. (Not necessarily towards the end of a cycle).

If the module is executed at runtime, the call occurs immediately, blocking the original process (it is a pointer...). Therefore, both modules (the caller and the called one) must be in the same real-time context and on the same CPU core.

The data pointer is configured in the TwinCAT Module Instance Configurator [▶ 129].

See sample10 [▶ 256]

TwinCAT Module Class for real-time context

This template creates a module, which can be instantiated in the real-time context.

As described in chapter TwinCAT module state machine [▶ 44], the other modules have transitions for startup and shutdown in a non-real-time context. In some cases modules have to be started when a real-time is already running, so that all transitions are executed in the real-time context. This is a corresponding

template.
The modules with this (modified) state machine can also be used for instantiation directly on startup of TwinCAT. In this case the transitions are executed like for a normal module.

The TcCOM 03 sample [▶ 318] illustrates the application of such a module.

TwinCAT Module Class with Online Change capability

This option can only be used if the module is added to a Versioned C++ Project.

This template creates a module that is capable of Online Change. Due to the revision control of the project, these modules become exchangeable at runtime - it is therefore possible to exchange modules from different versions at runtime.

The procedure for this Online Change is described in chapter Online Change [▶ 148].

The module itself otherwise corresponds to a module with cyclic input/output.

11.3 TwinCAT Module Class Editor (TMC)

The TwinCAT Module Class Editor (TMC Editor) is used for defining the class information for a module. It includes data type definitions and their application, provided and implemented interfaces, and data areas and data pointers.

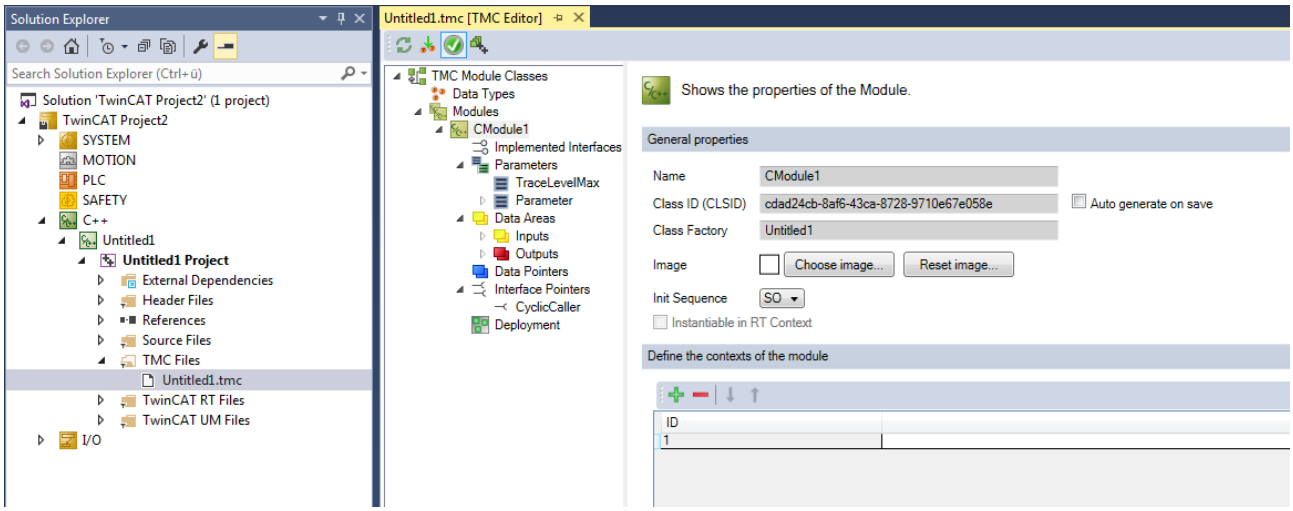
To put it briefly: everything that is visible from outside must be defined with this editor.

The basic idea is:

1. The TMC Editor can be used to modify the module description file (TMC file). This contains all information that is accessible in the TwinCAT system itself. These are for example symbols, implemented interfaces and parameters.
2. The TwinCAT Code Generator, which can also be called from the TMC Editor, is used to generate all the required C++ code, i.e. header and cpp files.

Start the TMC Editor

Open the editor by double-clicking on the TMC file of a module. The graphical editor opens:



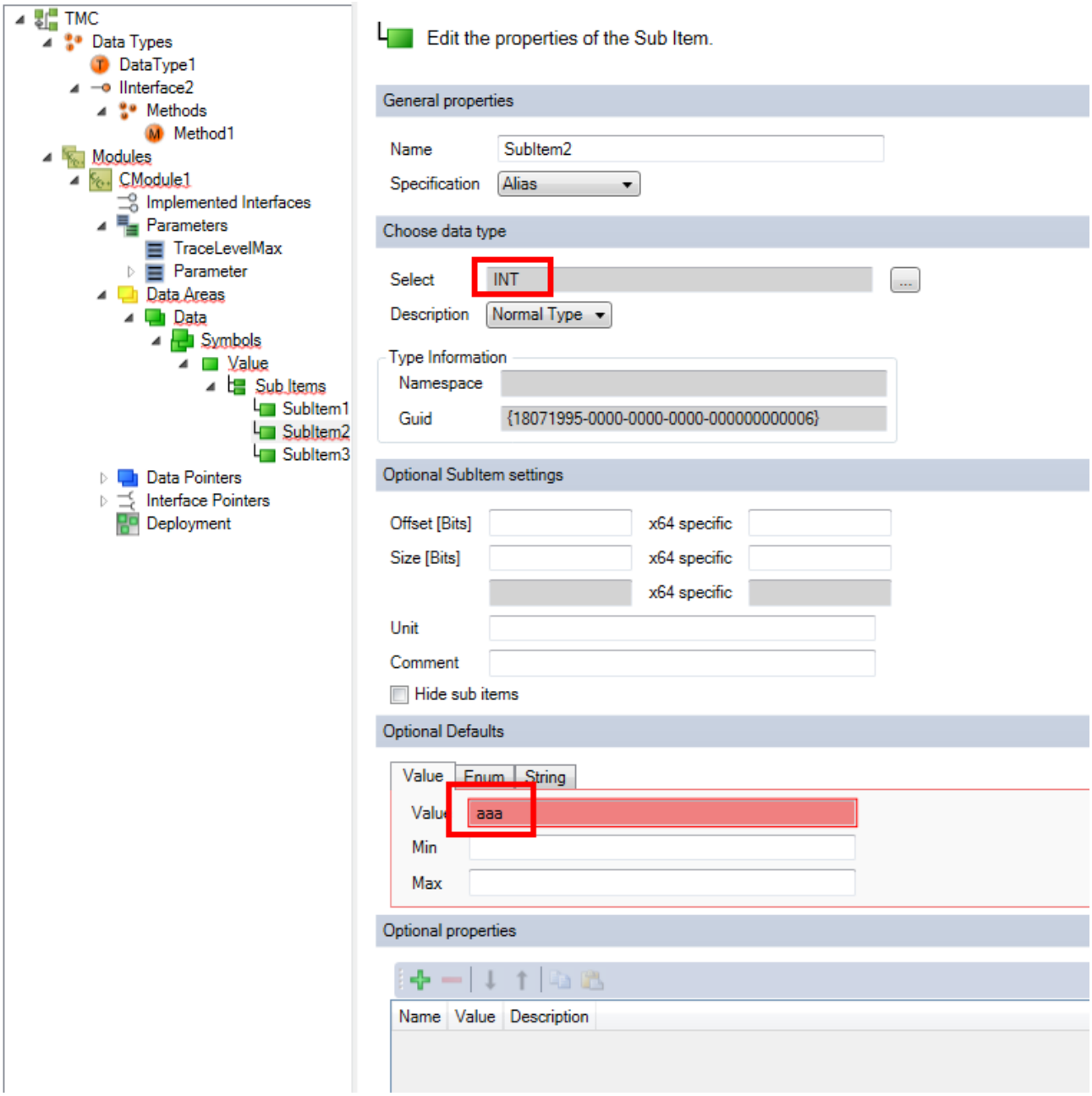
Functionalities of the TMC Editor:

- Create/delete/edit symbols in the data areas, e.g. the logical input or output process images of a module.
- Create/delete/edit user-defined data type definitions.
- Create/delete/edit symbols in the parameter list of a module.

User Help

The TMC Editor offers user support for the definition of data types and C++ modules.

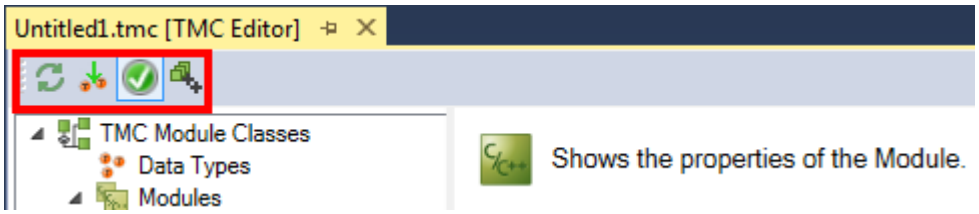
For example, in the event of problems (alignment, invalid standard definitions, ...) within the TMC, the user is guided to the relevant location via red markings within the TMC tree:



The user can nevertheless edit the TMCs directly, since they are XML files and can therefore be created and edited by the user.

Tools

The upper section of the TMC editor contains symbols for the required operations.



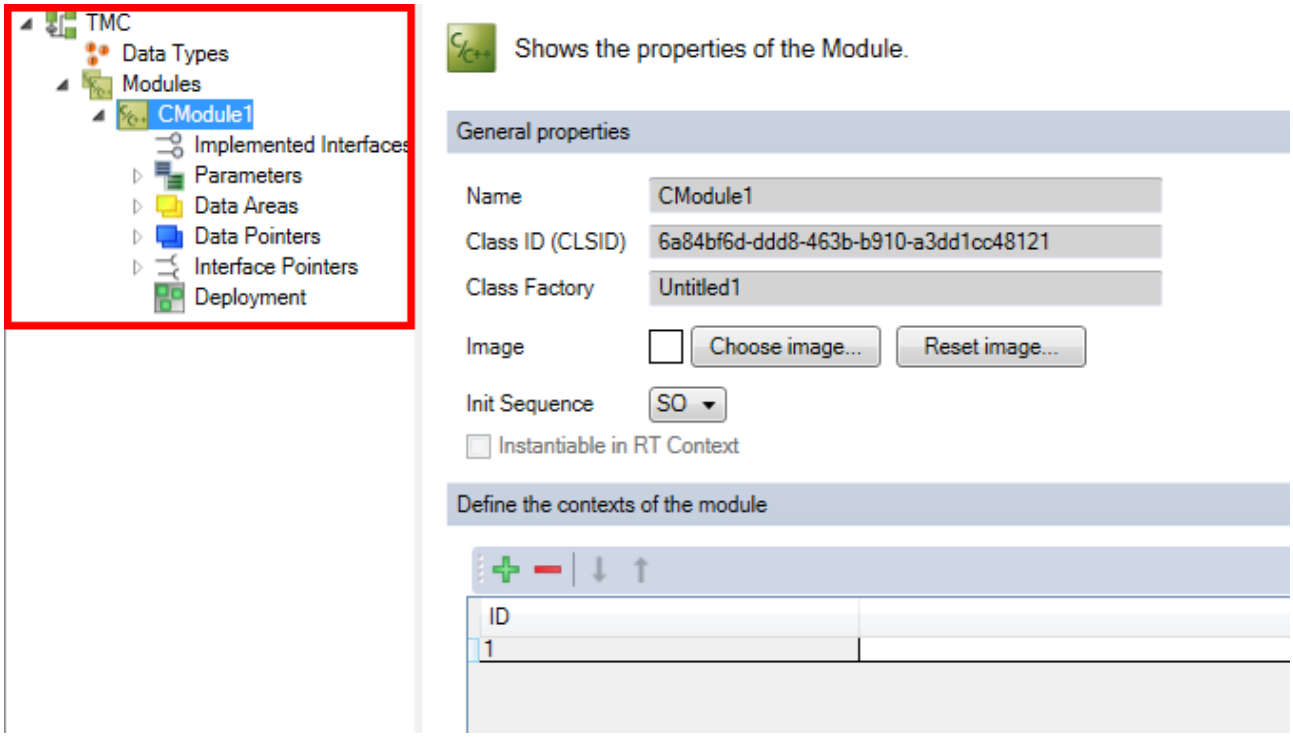
- Reloading of the TMC file and the types from the type system.

- Updating of the higher-level data types.
- Switching the User Help [▶ 89] on/off.
- Start the TwinCAT TMC Code Generator:

The editor will store the entered information in the TMC file. The TwinCAT TMC Code Generator converts this TMC description to source code, which is also available in the context menu of the TwinCAT C++ project.



11.3.1 Overview

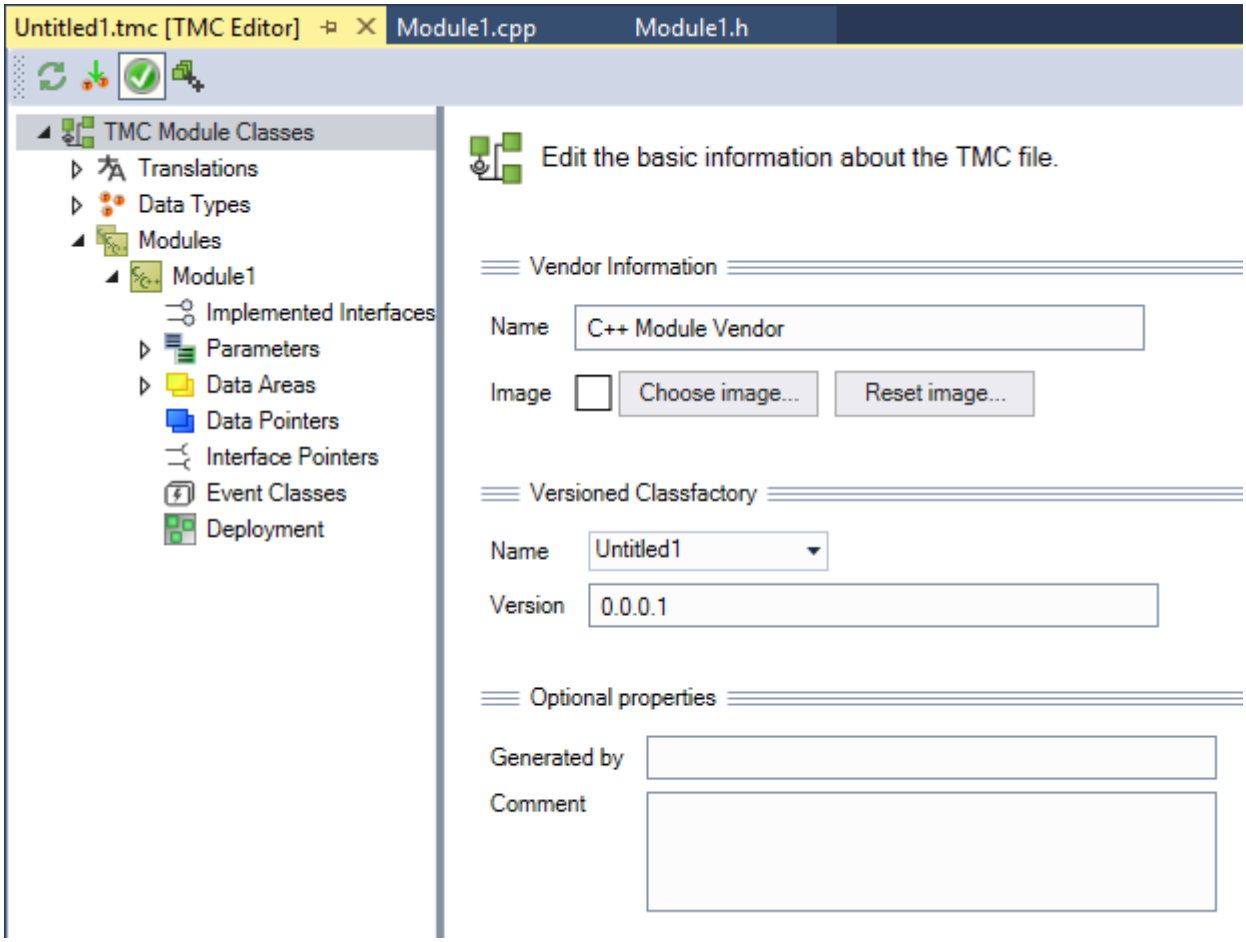


User interface

- TMC [▶ 91]: Here you can edit the C++ module vendor's basic information and add an image.
- Data types [▶ 92]: Data types are added, removed, or re-ordered here.
- Modules [▶ 109]: Shows the modules of the driver.
- Implemented Interfaces [▶ 111]: Shows the implemented interfaces of the module.
- Parameters [▶ 111]: Parameters are added, removed, or re-ordered here.
  - TraceLevelMax [▶ 118]: Parameter that controls the quantity of logged messages; predefined for (almost) every module.
- Data Areas [▶ 119]: Data areas are added, removed, or re-ordered here.
- Data Pointers [▶ 124]: Data pointers are added, removed, or re-ordered here.
- Interface Pointers [▶ 126]: Interface pointers are added, removed, or re-ordered here.
- Deployment [▶ 127]: Determines the files that are provided.

11.3.2 Basic Information

Basic information on the TMC file can be found here:



Information about the provider

Name: This is the name of the provider.

Choose Image: A 16 x 16 pixel bitmap icon is entered here.

Reset image: Resets the module image to the standard value.

Versioned Classfactory

Name: Displays all class factories referenced from the TMC file. The class factory that implements the C++ project must be set. Typically this is the name of the project.
Used only for versioned C++ projects; otherwise "not set" appears here.

Version: The current version, consisting of four digits, each separated by a ".". At least one digit must not be 0.

Optional features

Generated by: This field indicates who created the file and who will maintain it. Please note that changes are no longer possible in the TMC Editor when filling out this field (deactivates all editing procedures).

Comment: Optionally, you can enter a comment here.

11.3.3 Data Types

The user can define data types via the TwinCAT Module Class (TMC) editor.

These data types can be type definitions, structures, areas, enumerations or interfaces, e.g. methods and their signatures.

The TwinCAT Engineering system (XAE) publishes these data types in relation to all other nested projects of the TwinCAT project, so that they can also be used in PLC projects, for example (as described in chapter Sample11: Module communication: method call PLC module to C++ module [▶ 257]).

NOTICE

Name conflict

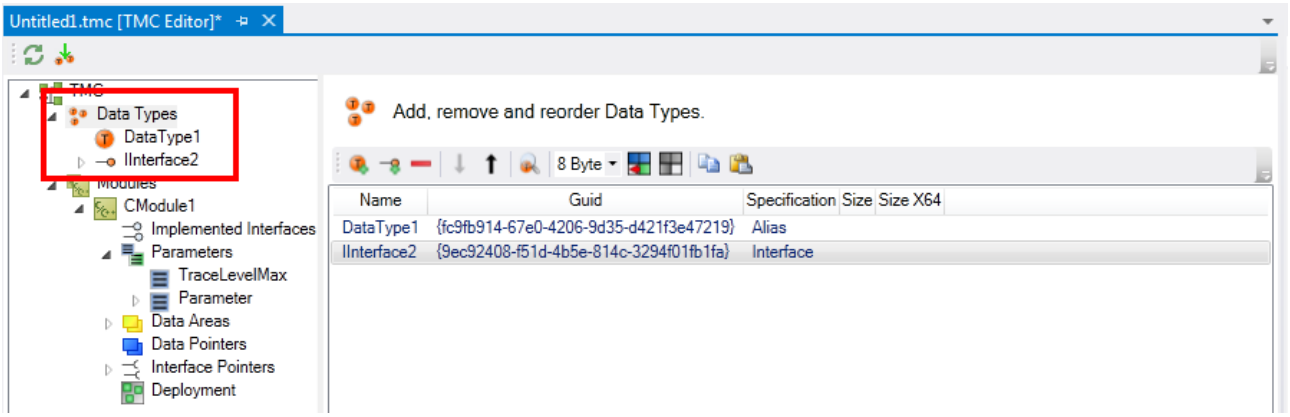
A name collision can occur if the driver is used in combination with a PLC module.

- Do not use any of the keywords that are reserved for the PLC as names.

This chapter describes how to use the capabilities of the TMC Editor for defining data types.

11.3.3.1 Overview

User interface



Symbol	Function
	Add a new data type.
	Add a new interface.
	Delete the selected type.
	Move the selected element one place downwards.
	Move the selected element one place upwards.
	Search for unused types.
	Select byte alignment.
	Align selected data type (Alignment) . This function runs through all data types used (recursion). Is this is not desired, a step-by-step approach can be adopted, by using the function within the data types.
	Reset data format of the selected data type.



Copy



Paste

Data type properties

Name: user-defined name of the data type.

GUID: unique ID of the data type.

Specification: specification of the data type.

Size: size of the data type, if expressly specified.

Size X64: different size of the data type for the x64 platform.

11.3.3.2 Add / modify / delete data types

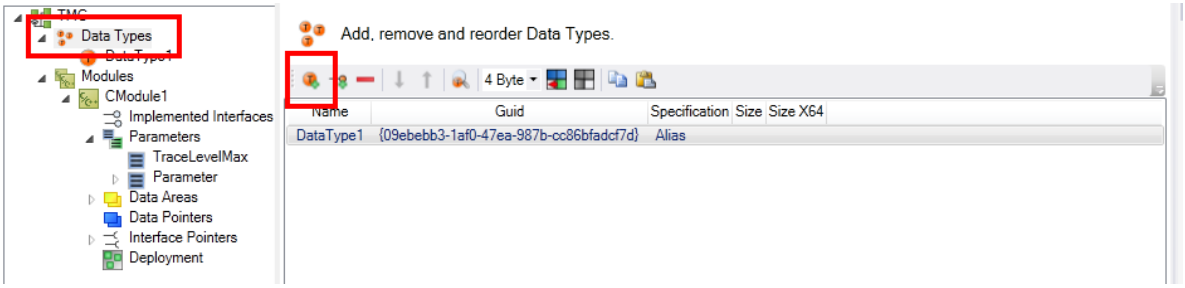
Data types used by TwinCAT C++ modules can be added, edited and deleted with the help of the TwinCAT Module Class (TMC) editor.

This article describes:

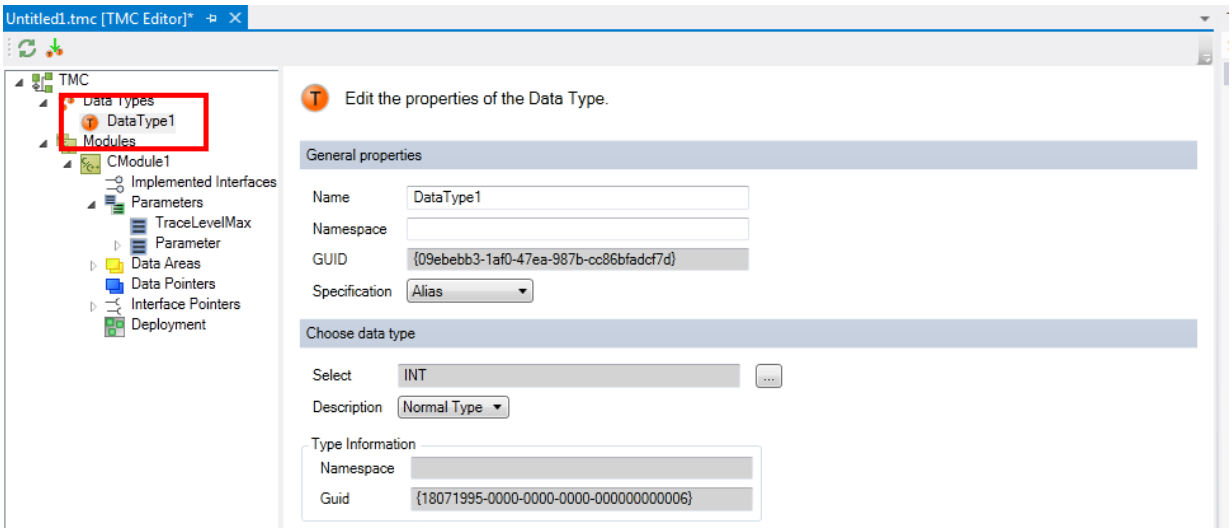
- Step 1: Create a new data type [▶ 93] in the TMC file.
- Step 2: Start the TwinCAT TMC Code Generator [▶ 96] in order to generate C++ code on the basis of a module description in the TMC file.
- Using [▶ 109] the data types.

Step 1: Generate a new data type

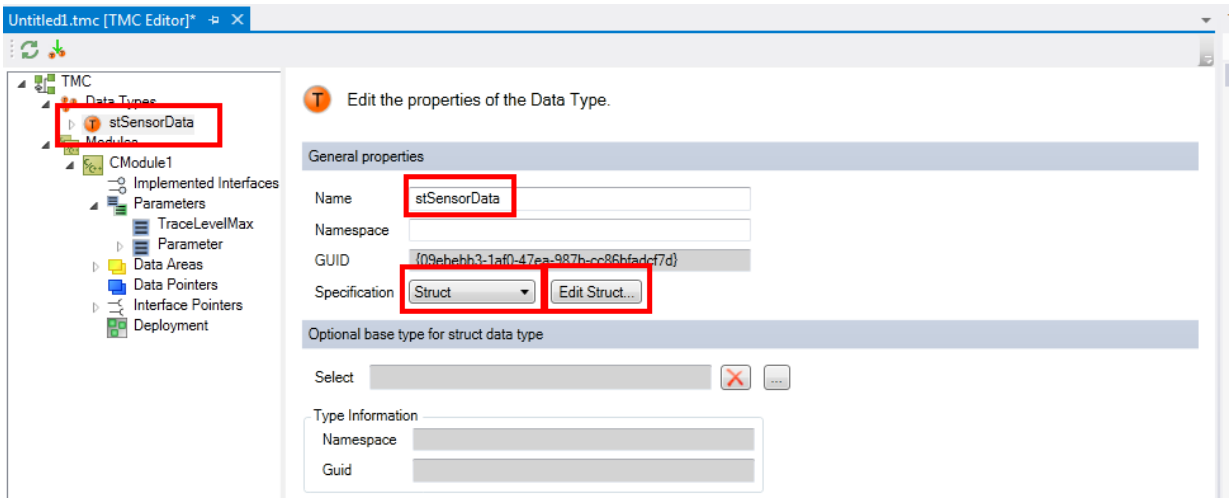
1. After starting the TMC Editor, select the **Data Types** node.
2. Extend the list of data types and interfaces by a new data type by clicking on the **+** button **Add a new data area**.
   ⇨ A new **data type** is then listed as a new entry:



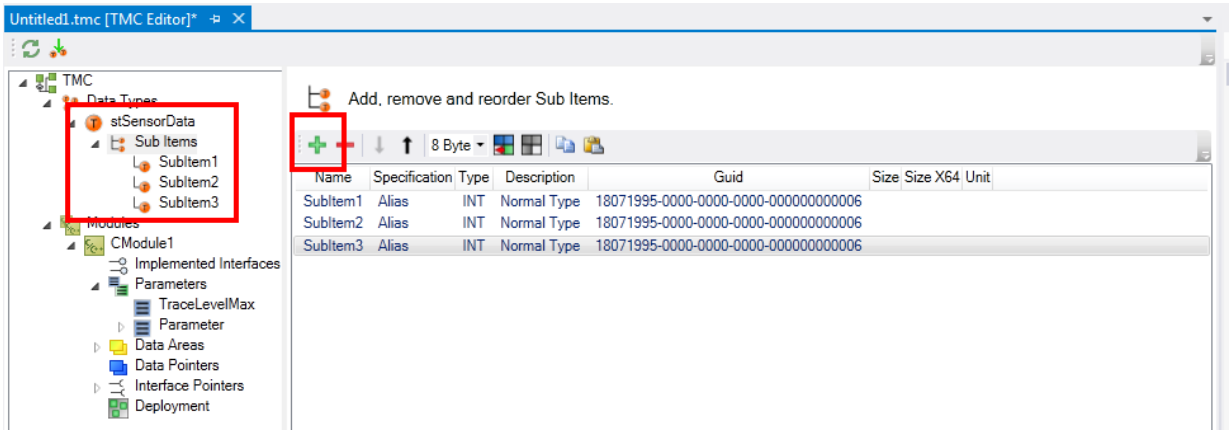
3. Select the generated "Data Type1" in order to obtain details of the new data type.



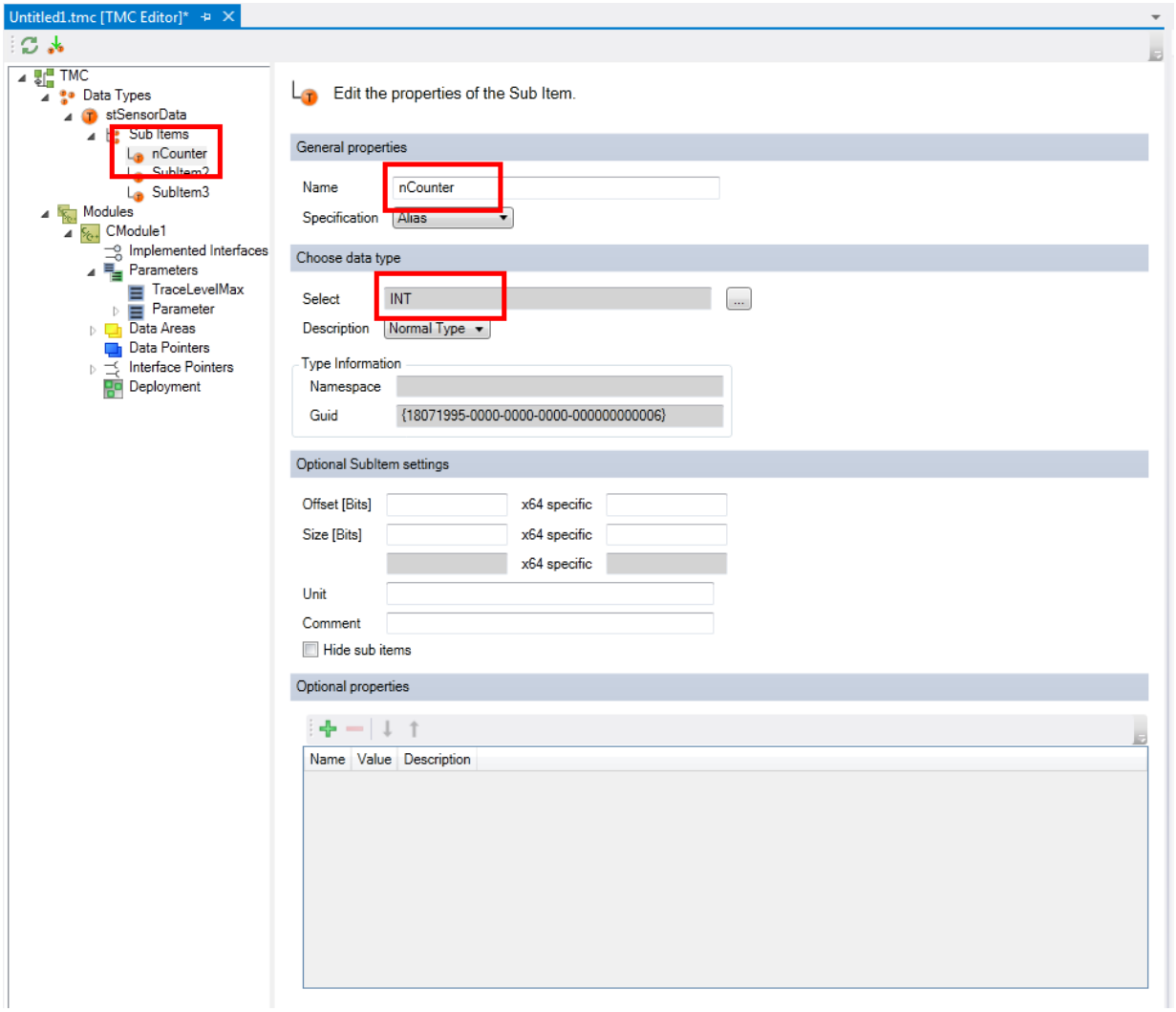
4. Specify [▶ 103] the data type.
5. Rename the data type.
   In this sample **stSensorData**, select the specification **STRUCT** and click on **Edit Struct**.



6. Insert new sub items in the structure by clicking on the **Add a new sub item** button.



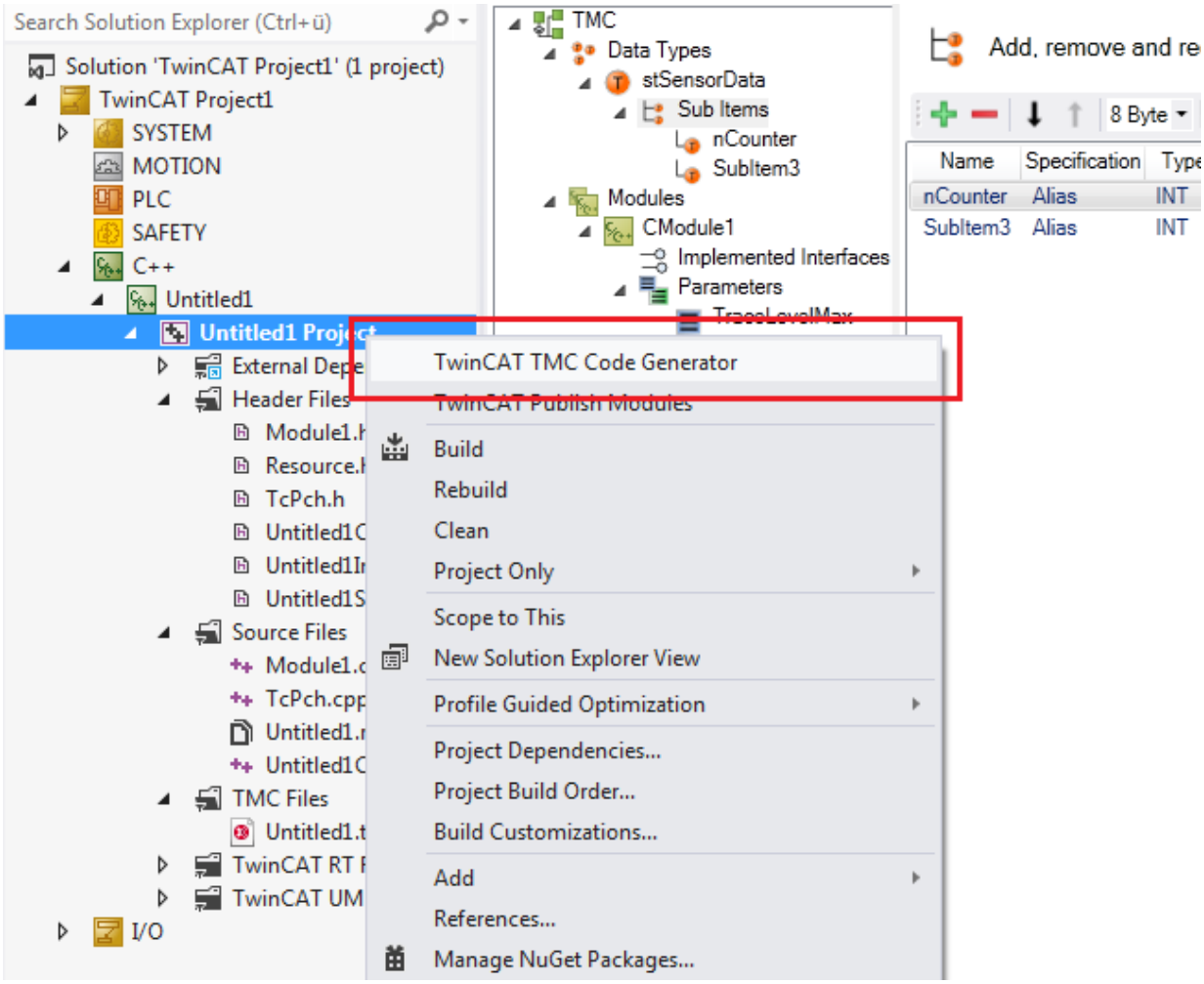
7. You can edit the properties by double-clicking on the sub item. Give the sub item a new name and select a suitable data type.



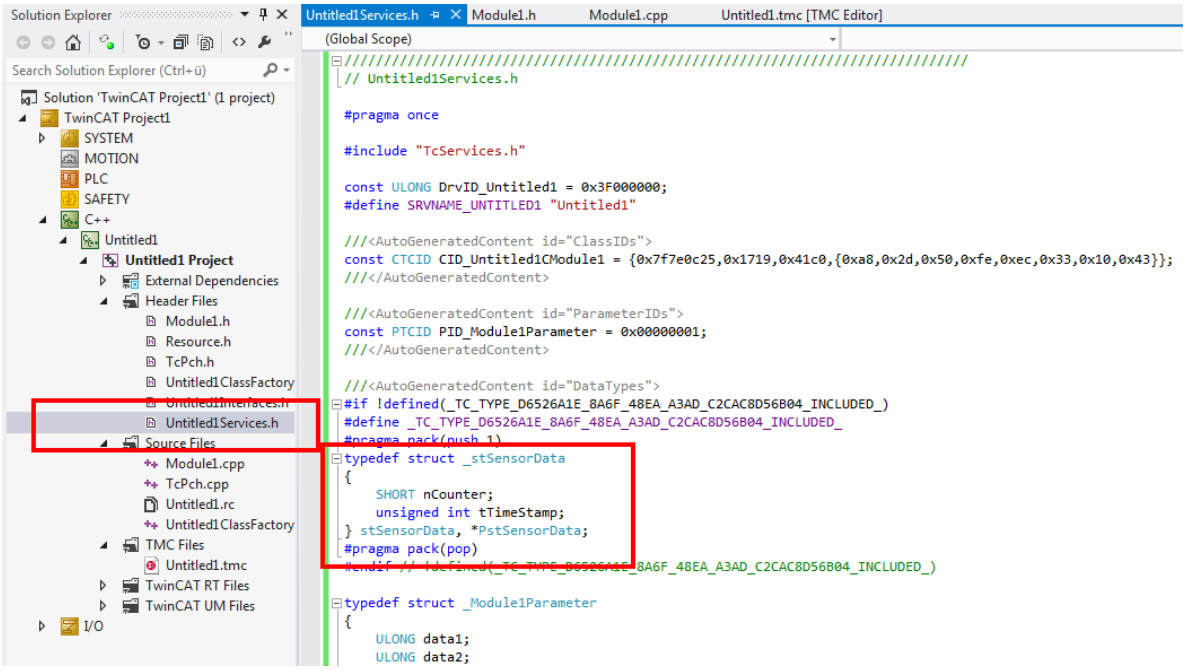
8. Give the other sub items a new name and select a suitable data type.
9. Save the changes you have made in the TMC file.

Step 2: Start the TwinCAT TMC Code Generator to generate code for the module description.

1. Right-click on your project file and select **TwinCAT TMC Code Generator** to generate the source code for your data type:



⇨ You can see the data type declaration in the module header file "Untitled1Services.h"



⇨ If you add a further data type or a further sub-element, run the TwinCAT TMC Code Generator again.

11.3.3.3 Add / modify / delete Interfaces

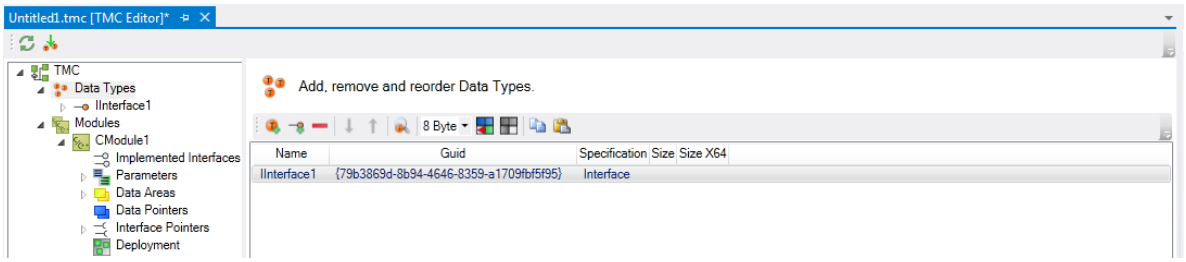
Interfaces of a TwinCAT module can be added, edited and deleted with the help of the TwinCAT Module Class (TMC) editor.

This article describes:

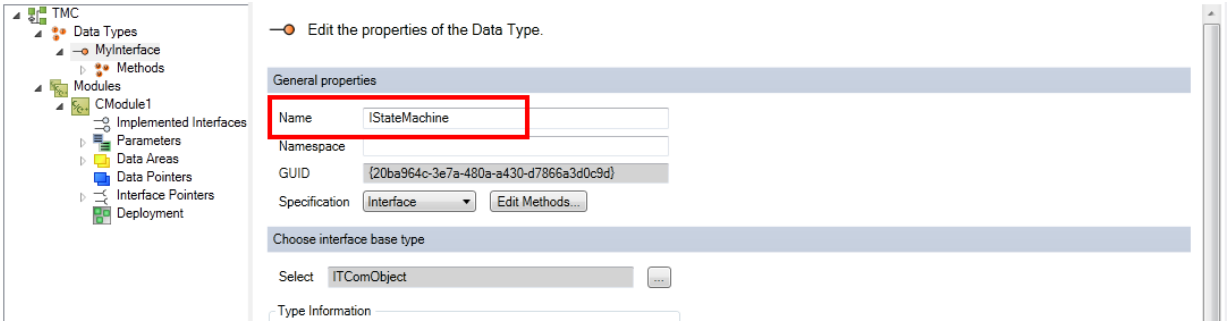
- Step 1: Create a new interface [▶ 97] in the TMC file.
- Step 2: [▶ 97]Add methods [▶ 97] to the interface in the TMC file.
- Step 3: Use the interface [▶ 99] by adding it to the "Implemented Interfaces" of the module.
- Step 4: Start the TwinCAT TMC [▶ 101] Code Generator to generate code for the module description.
- Optional change of the interface [▶ 101].

Step 1: generate a new interface

1. After starting the TMC Editor, select the **Data Types** node.
2. Click on **Add a new interface** to extend the list of interfaces by a new interface.
   ⇨ A new entry **IInterface1** is then listed:



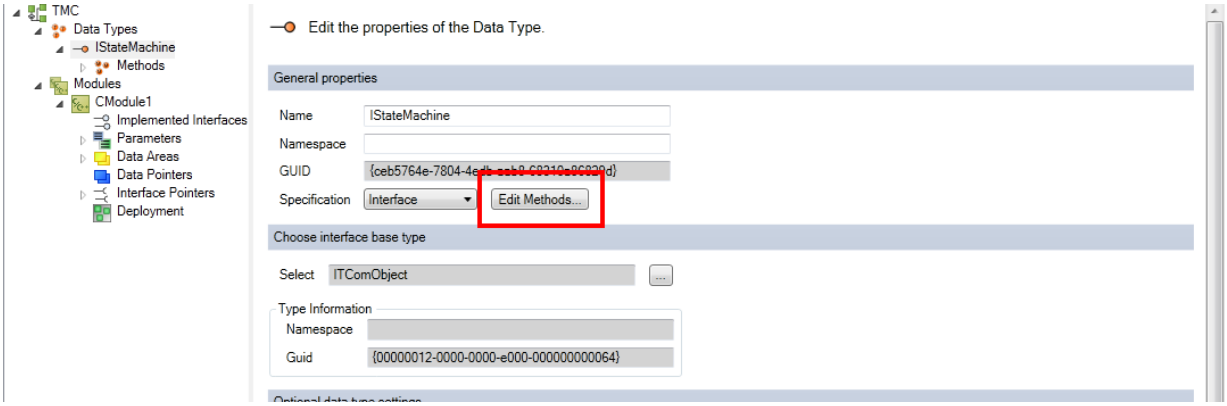
3. To open the details you can either select the appropriate node in the tree or double-click on the row in the table.



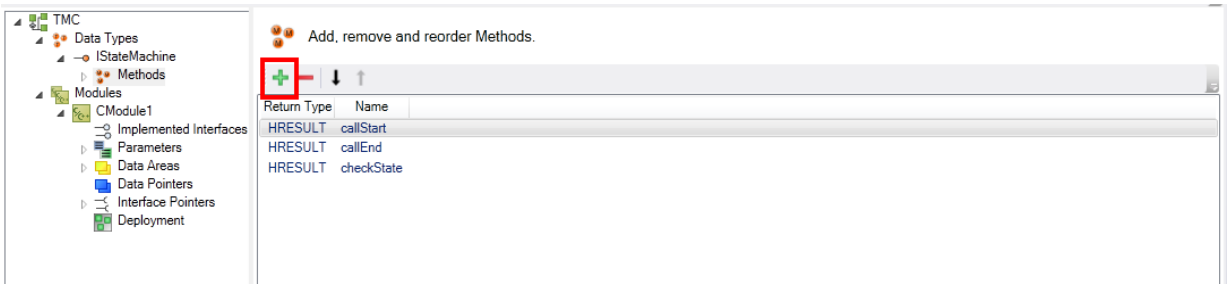
4. Enter a meaningful name - in this sample "IStateMachine".

Step 2: add methods to the interface

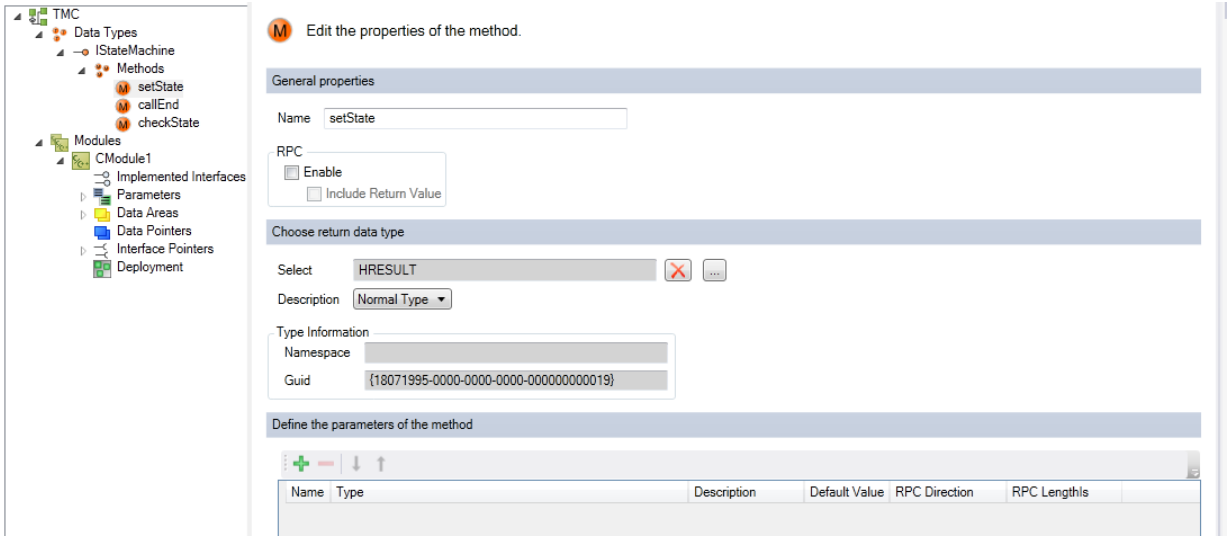
1. Click on **Edit Methods...** to get a list of the methods of this interface:



2. Click on the **+** button to generate a new default method, "Method1".



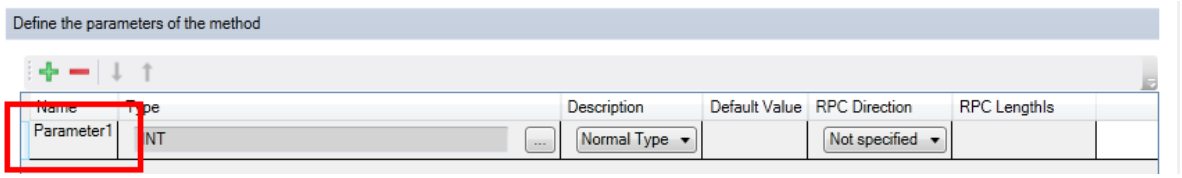
3. Double-click on the method or select a node in the tree to open the details.
4. Give the default "Method1" a more meaningful name.
5. Subsequently, you can add parameters by clicking on **Add a new parameter** or edit parameters of the "SetState" method.



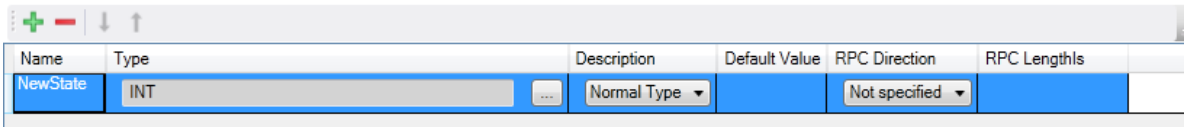
⇨ The new parameter, "Parameter1", is generated by default as "Normal Type" "INTEGER".

6. Edit the parameter by clicking on the name "Parameter1".

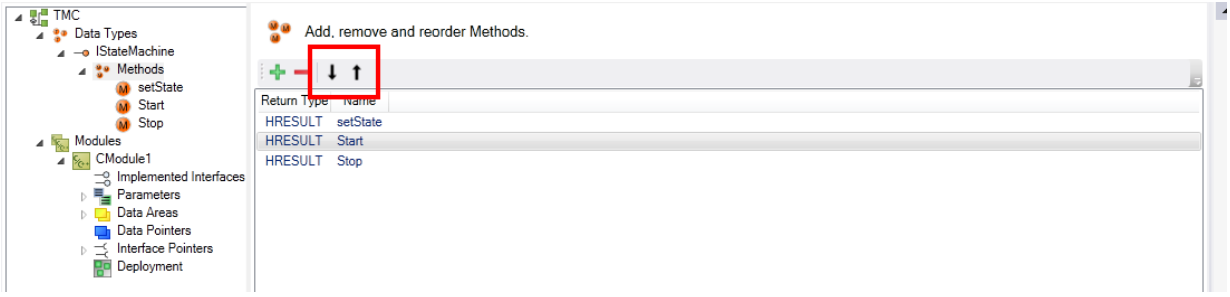
⇨ The "Normal Type" can also be changed to "Pointer" and so on – the data type itself can also be selected.



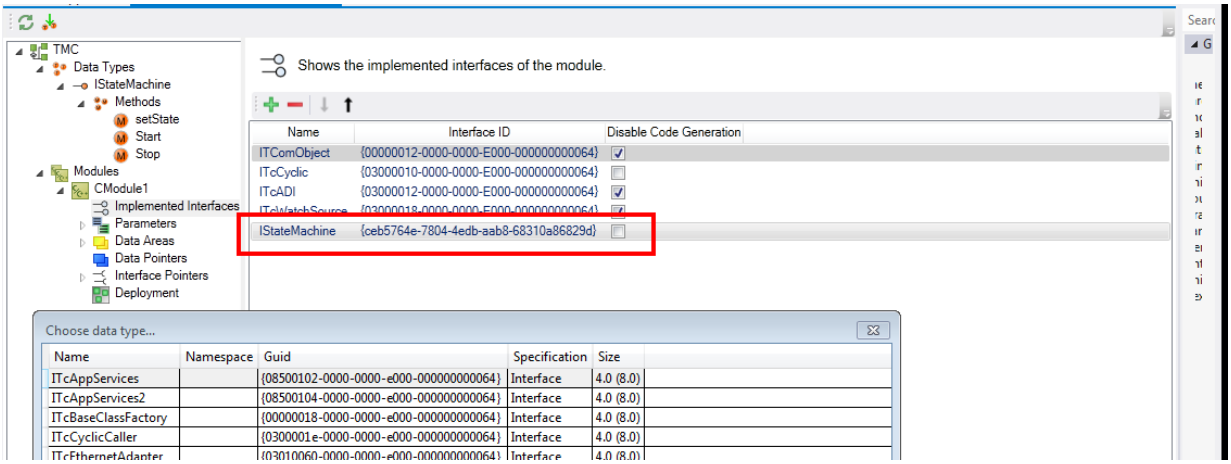
⇨ In this case "NewState" is the new name – the rest of the settings are not changed.



7. By repeating step 2 "Add methods to interface", all methods are listed – you can re-order the methods with the help of the **move up** / **move down** button.

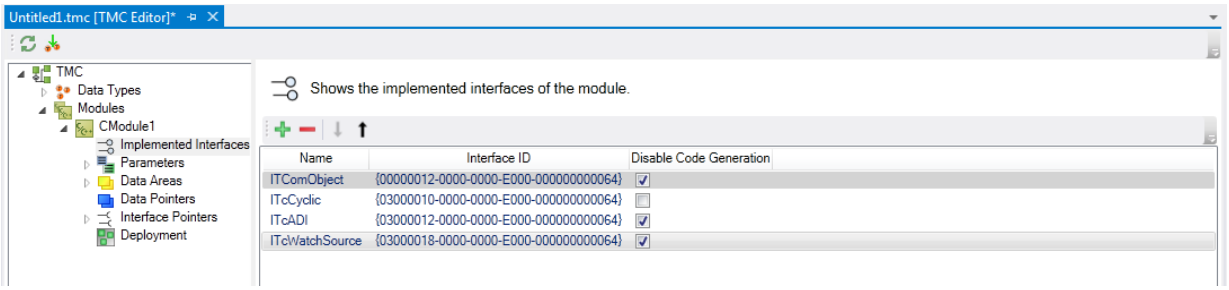


8. The interface is ready to be implemented by your module.

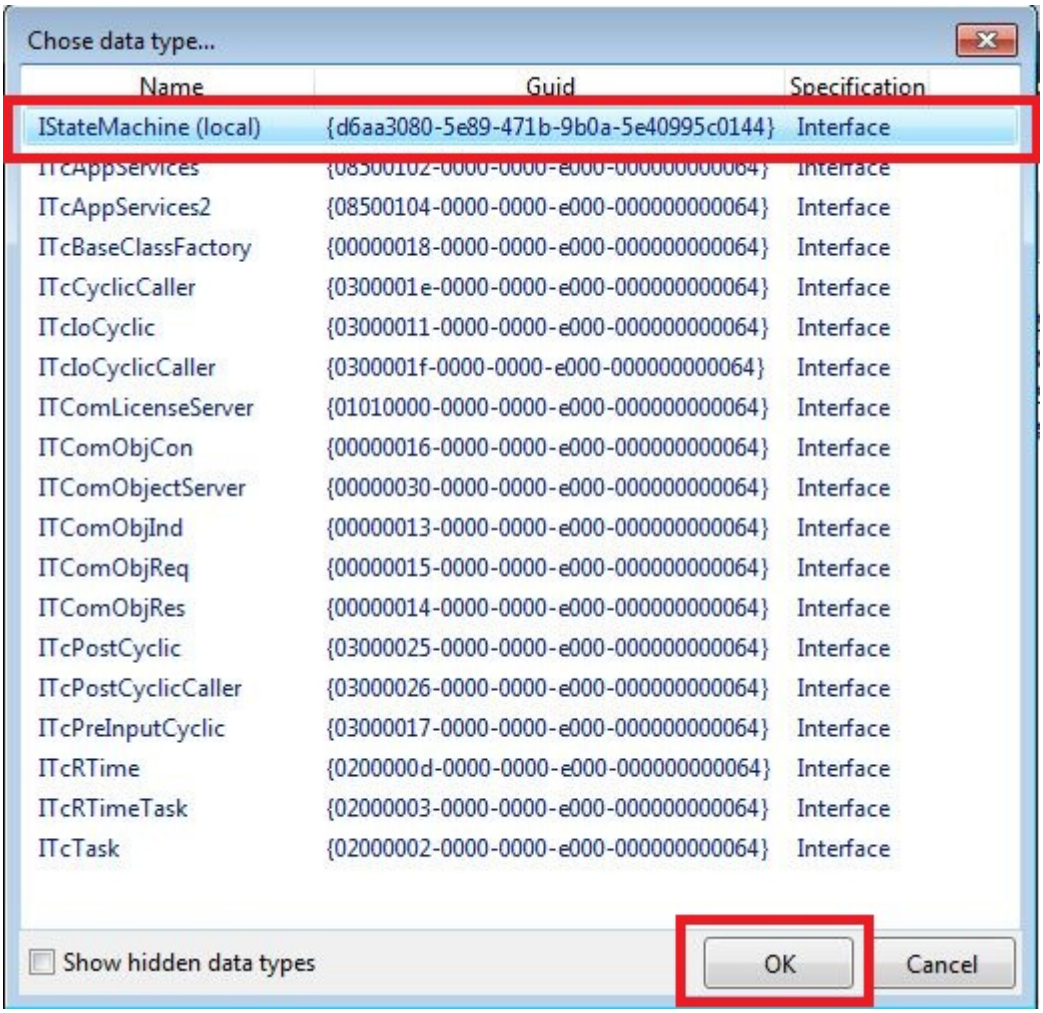


Step 3: Add the new interface to Implemented Interfaces

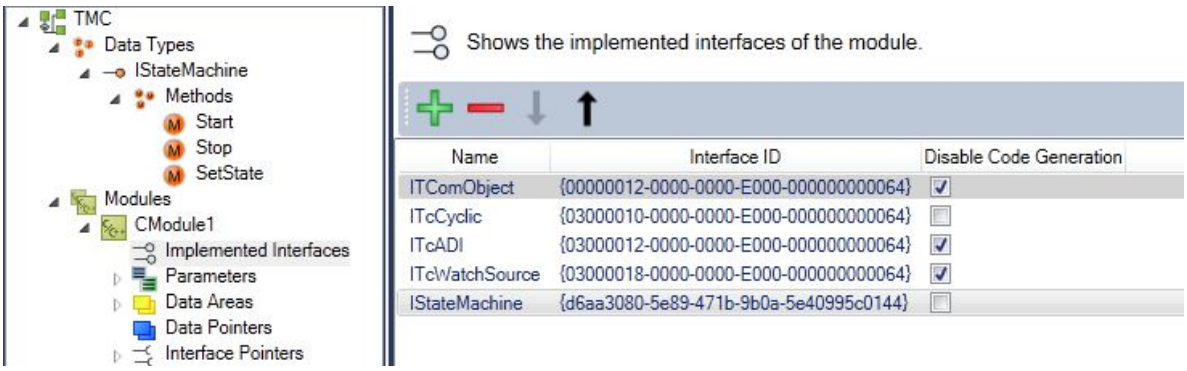
1. Select the module that is to be extended by the new interface - in this case select the destination **Modules->CModule1**.
2. Extend the list of implemented interfaces by a new interface with **Add a new interface to the module** by clicking on the **+** button.



3. All available interfaces are listed - select the new template "IStateMachine" and end with **OK**.

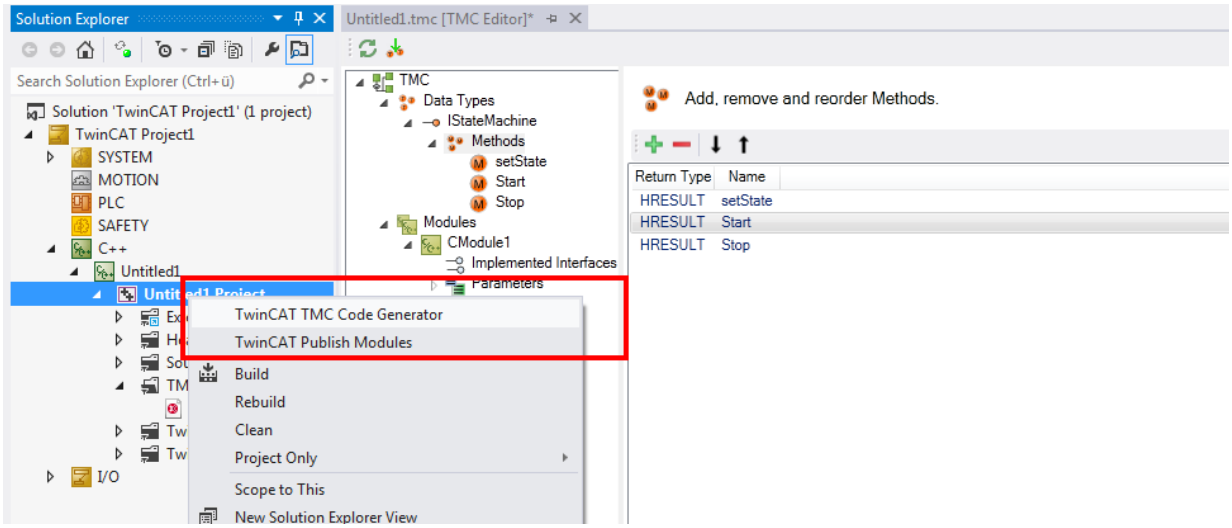


⇨ The new interface "IStateMachine" is part of the module description.

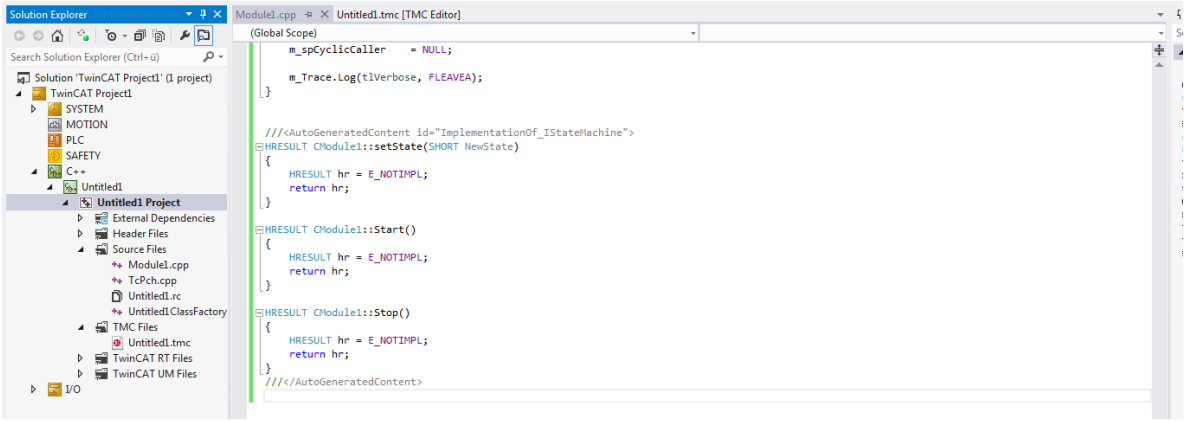


Step 4: Start the TwinCAT TMC Code Generator to generate code for the module description.

1. In order to generate the C/C++ code on the basis of this module description, right-click in the C/C++ project and then select the **TwinCAT TMC Code Generator.**



⇨ The module "Module1" then contains the new interfaces
CModule1: Start()
CModule1: Stop()
CModule1: SetState(SHORT NewState).



⇨ Done – the user-defined code can now be inserted in this area.

Optional change of the interface

User-defined code will never be deleted

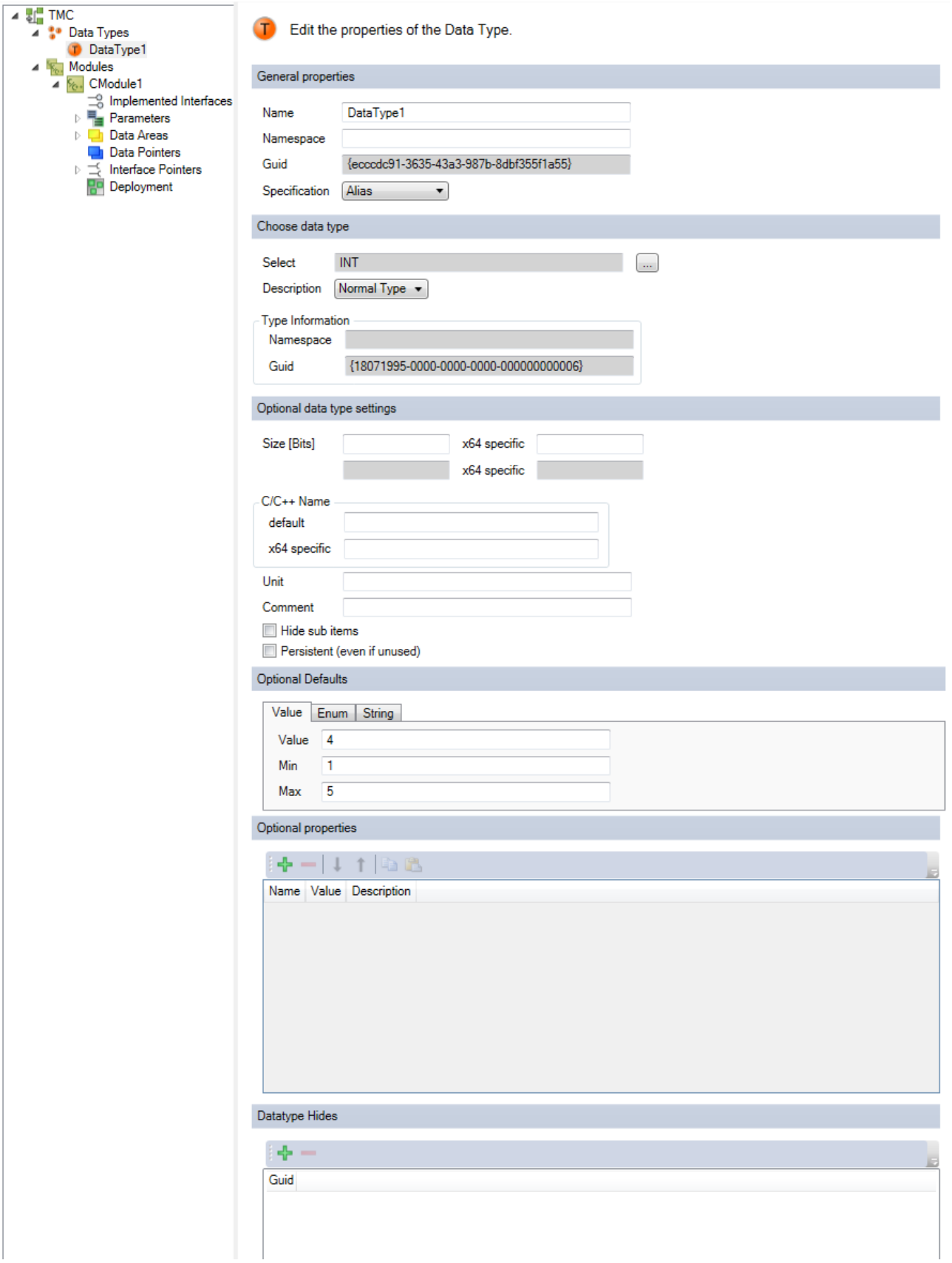
In the case of changes to the interface (e.g. the parameters of a method will be extended later), the user-defined code will never be deleted. Instead, the existing method will merely be provided with a comment if the TMC Code Generator cannot map the methods.

```
///<AutoGeneratedContent id="ImplementationOf_IStateMachine">
HRESULT CModule1::SetState(SHORT SetState, bool bRun)
{
    HRESULT hr = E_NOTIMPL;
    return hr;
}
///</AutoGeneratedContent>

///<AutoGeneratedContent id="Obsolete_ImplementationOf_IStateMachine">
//HRESULT CModule1::SetState(SHORT SetState)
//{
//  HRESULT hr = E_NOTIMPL;
//
//  //custom code
//  nState = SetState;
//
//  return hr;
//}
//
///</AutoGeneratedContent>
```

11.3.3.4 Data type properties

Editing the properties of data types



General properties

Name: user-defined name of the data type.

NOTICE

Name conflict

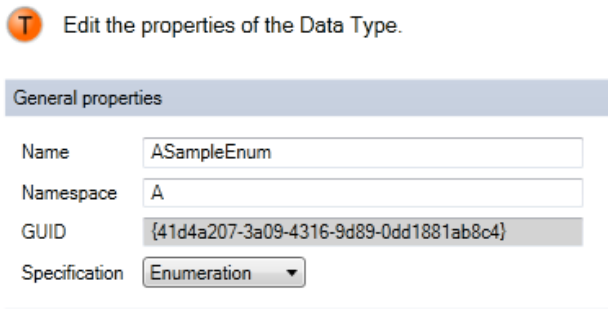
A name collision can occur if the driver is used in combination with a PLC module.

- Do not use any of the keywords that are reserved for the PLC as names.

Namespace: user-defined namespace of the data type.

Please note that this is **not** assigned to a C namespace. It is used as the prefix to your data type.

Sample: an enumeration with a namespace "A":



The following code is generated:

```
///<AutoGeneratedContent id="DataTypes">
#if !defined(_TC_TYPE_41D4A207_3A09_4316_9D89_0DD1881AB8C4_INCLUDED_)
#define _TC_TYPE_41D4A207_3A09_4316_9D89_0DD1881AB8C4_INCLUDED_
enum A_ASampleEnum : SHORT {
One,
Two,
Three
};
#endif // !defined(_TC_TYPE_41D4A207_3A09_4316_9D89_0DD1881AB8C4_INCLUDED_)
```

You may wish to manually append the namespace name to the enumeration element as a prefix:

```
#if !defined(_TC_TYPE_C26FED5F_AC13_4FD3_AC6F_B658CB5604E0_INCLUDED_)
#define _TC_TYPE_C26FED5F_AC13_4FD3_AC6F_B658CB5604E0_INCLUDED_
enum B_BSampleEnum : SHORT {
B_one,
B_two,
B_three
};
#endif // !defined(_TC_TYPE_C26FED5F_AC13_4FD3_AC6F_B658CB5604E0_INCLUDED_)
```

GUID: unique ID of the data type.

Specification: specification of the data type.

- **Alias**: generate an alias of a standard data type (e.g. INT).
- Array [▶ 106]: create a user-defined array.
- Enumeration [▶ 106]: create a user-defined enumeration.
- Struct [▶ 107]: generate a user-defined structure.
- Interface [▶ 107]: generate a new interface.

Select data type

Select: Select data type – it can be a basic TwinCAT data type or a user-defined data type.
Data types equivalent to the PLC data types are defined (like TIME, LTIME, etc.). See Data Types of the PLC for further information.

Description: Define the type as pointer, reference or value by means of the appropriate selection.

- Normal type
- Pointer

- Pointer to pointer
- Pointer to pointer to pointer
- a reference

Type information

- **Namespace:** Defined for selected data type.
- **GUID**: Unique ID of the selected data type.

Optional data type settings

Size [Bits]: Size in bits (white fields) and in "Byte.Bit" notation (grey fields). A different size can be defined for the x64 platform.

C/C++ Name: name used in the generated C++ code. The TMC code generator will not generate the declaration, so that user-defined code can be provided for this data type. Beyond that a different name can be defined for x64.

Unit: a unit of the variable.

Comment: comment that is visible, for example, in the instance configurator.

Hide sub items: If the data type has sub-elements, the System Manager will not allow the sub-elements to be accessed. This should be used, for example, in the case of larger arrays.

Persistent (even if unused): Persistent type in the global type system (cf. System->Type System->Data Types).

Optional Defaults

Depending on data type the default could be defined.

Optional Properties

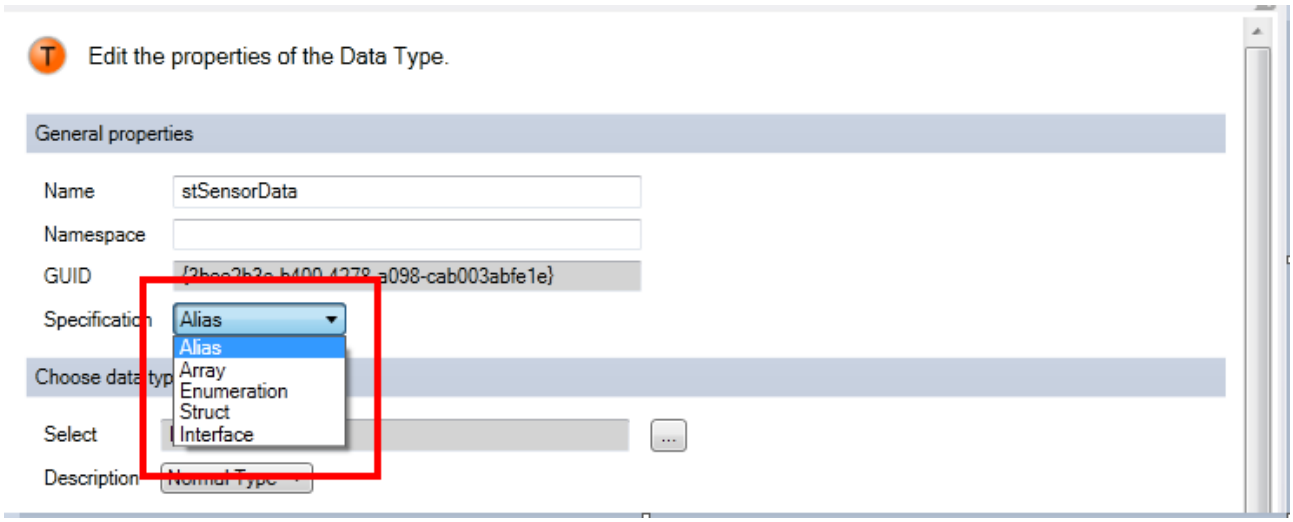
A table of name, value and description for annotating the data type.
This information is provided within the TMC and also TMI files.
TwinCAT functions as well as customer programs can use these properties.

Datatype Hides

Listed GUIDs refer to data types which are hidden by this data type. Normally, GUIDs of previous versions of this data type are inserted here automatically on each change.

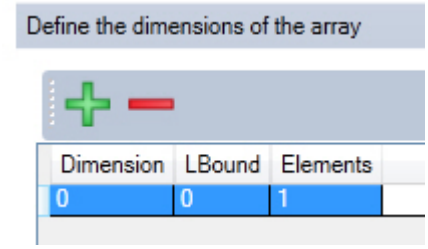
11.3.3.5 Specification

This section describes the specification of data types.



11.3.3.5.1 Array

Array: Create a user-defined array.



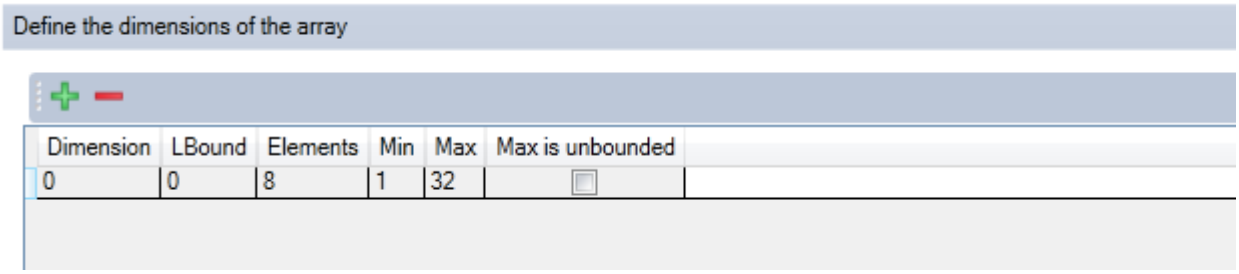
A new dialog is shown for adding (**+**) or removing (**-**) array elements.

Dimension: Dimension of the array.

LBound: Left limit of the array (default value = 0).

Elements: Number of elements.

Dynamic arrays for parameters and data pointers



In the case of parameters [▶ 111] and data pointers [▶ 124], TwinCAT 3 supports arrays with a dynamic size.

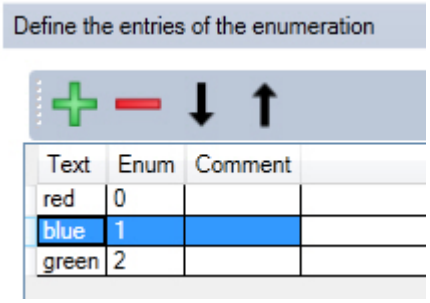
Min: Minimum size of the array.

Max: Maximum size of the array.

Max is unbounded: Indicates that there is no upper limit for the array size.

11.3.3.5.2 Enum

Enumeration: Create a user-defined enumeration.



A new dialog is shown for adding (**+**) or removing (**-**) an element. Edit the order with the help of the arrows.

NOTICE

Unique names are required for enumeration elements

Please note that the enumeration elements must have unique names, as otherwise the C++ code generated is invalid.

Text: Enumeration element

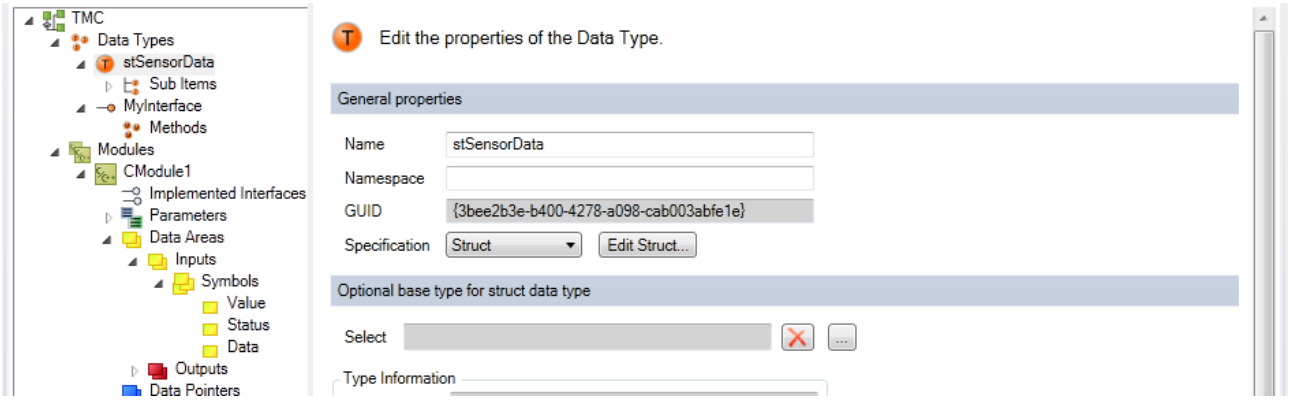
Enum: Suitable integer value.

Comment: Optional comment.

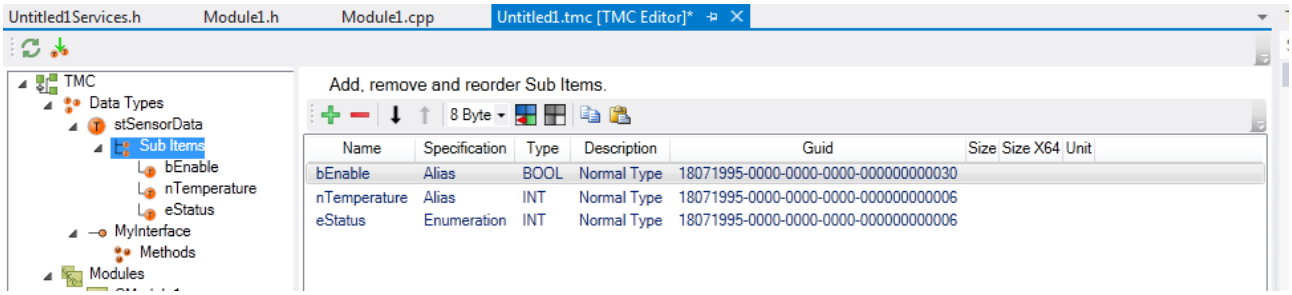
11.3.3.5.3 Struct

Struct: Creating a user-defined structure.

Select the **Sub Items** node or click on the **Edit Struct** button to switch to this table:



A new dialog is shown for adding (**+**) or removing (**-**) an element. Edit the order with the help of the arrows.



Name: Name of the element.

Specification: A struct can contain aliases, arrays or enumerations.

Type: Type of the variable.

Size: Size and offset of the sub-element.

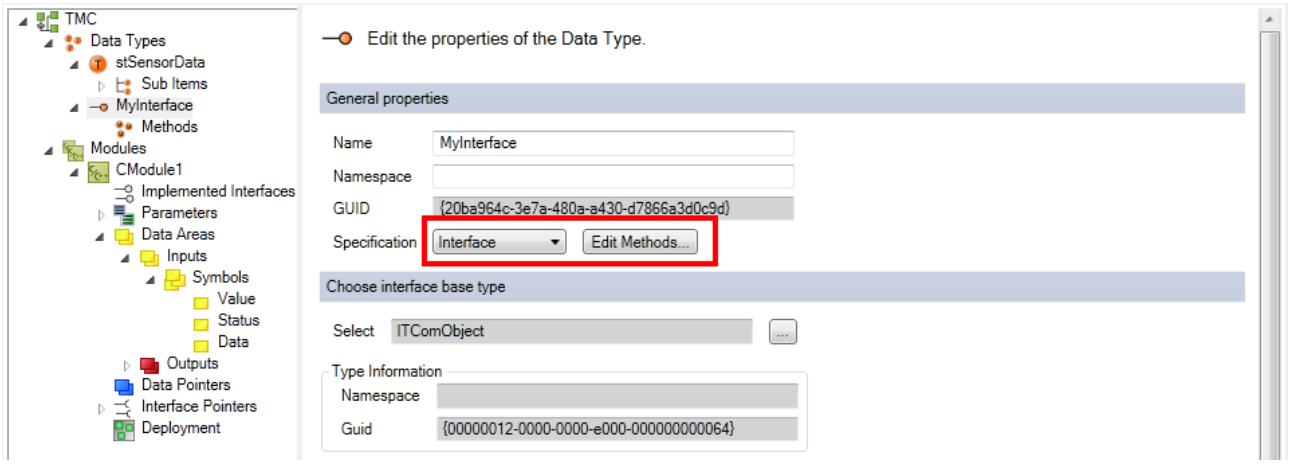
Size X64: Other size for the x64 platform will additionally be provided.

Unit: Optional unit.

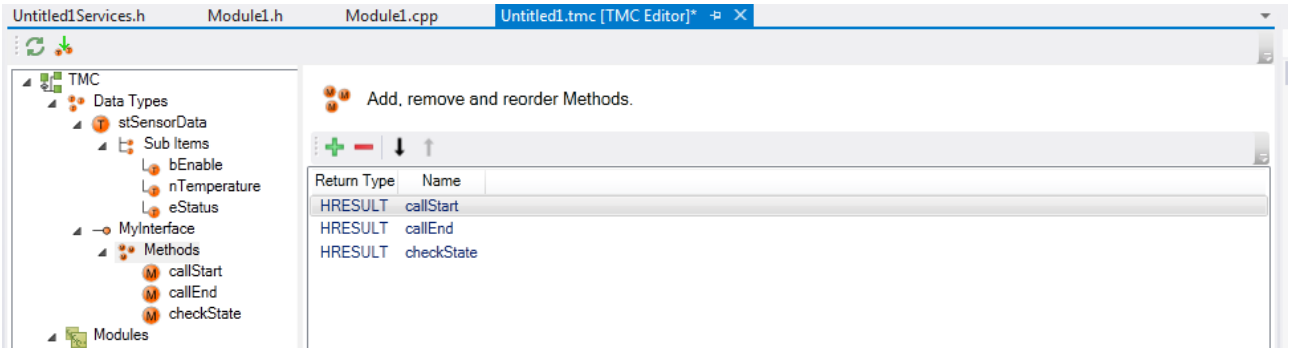
The details of the configuration page of the sub-element are shown by selecting the data type or double-clicking on the table entry. Similar to Data type properties [▶ 103].

11.3.3.5.4 Interfaces

Interfaces: Creating a user-defined interface.

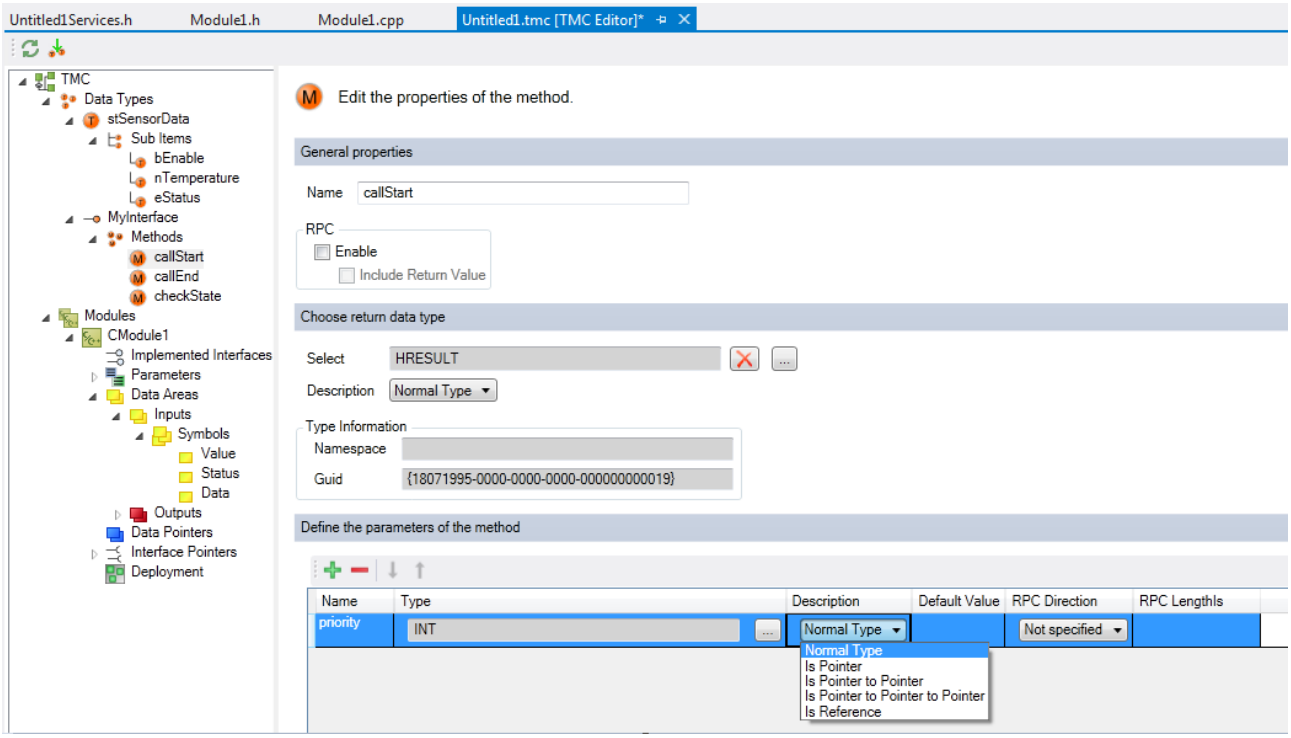


Select the **Methods** node or click on the **Edit Methods** button to switch to this table:



Method parameters

Select the Methods node or double-click on the entry to view the details of the method.



Name: The name of the method.

RPC enable: Enable remote procedure calls from outside this method.

Include Return Value: Enable sharing of the method's return value.

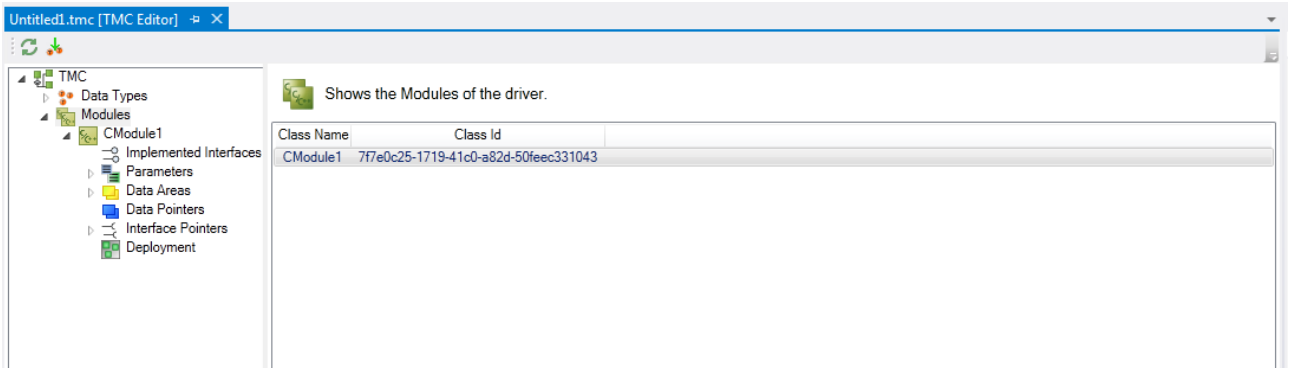
The fields correspond to those of the Data type properties [▶ 103].

Defining the method parameters

- **Name:**
- **Type:** Known from the Data type properties [▶ 103].
- **Description:** Known from the Data type properties [▶ 103].
- **Default Value:** Default value of this parameter; only numbers are allowed.
- **RPC direction:** As in the case of PLC function blocks, each parameter can either be IN, OUT or INOUT. Over and above that, it can be defined as NONE so that this parameter is ignored in the case of remote procedure calls (RPC).

11.3.4 Modules

Modules: Displays the modules of the driver.

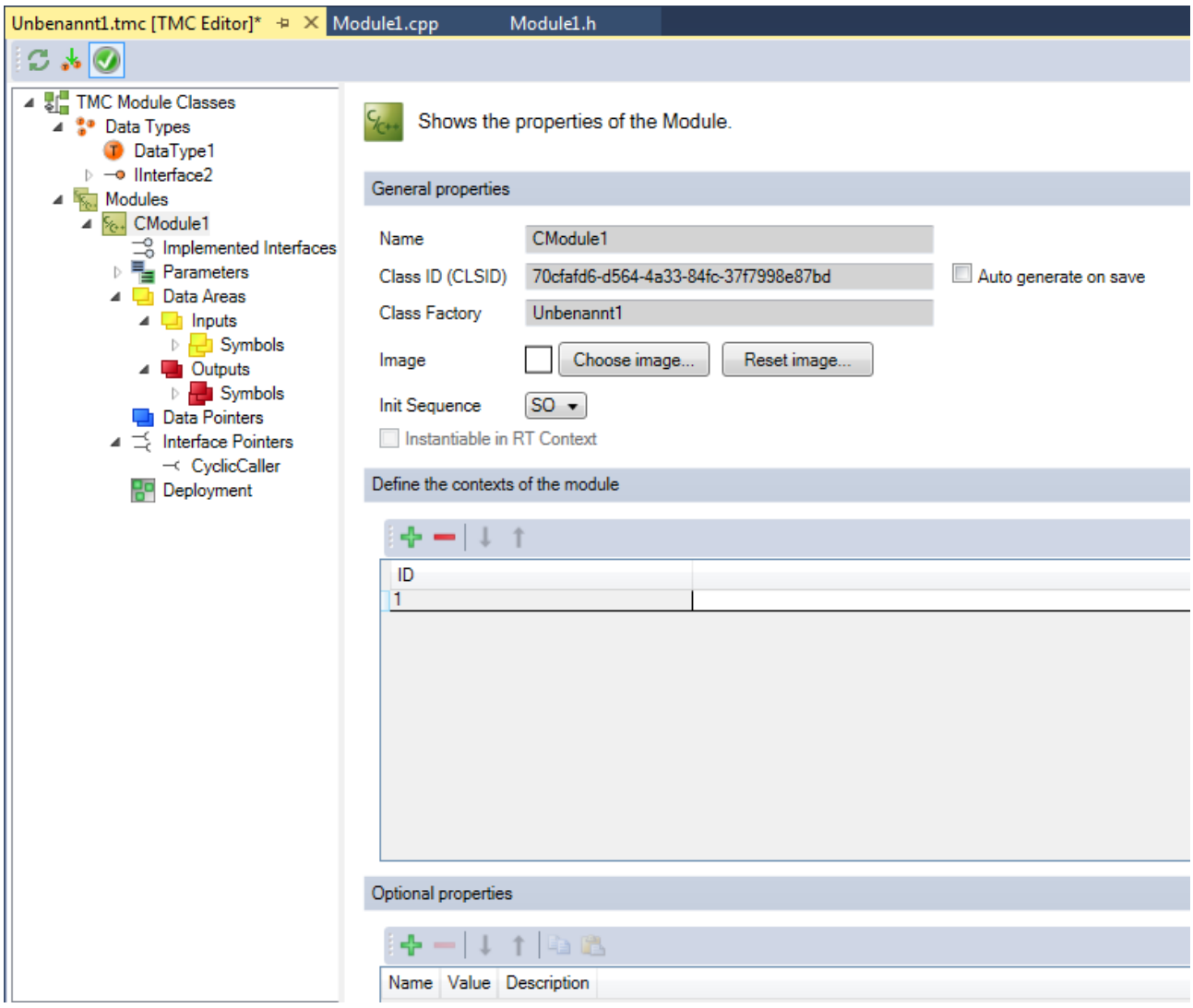


Class Name: Name of the module.

Class ID: Unique ID of the module.

Module Properties:

Click on the node in the tree or the row in the table to open the module properties.



General properties

Name: Name of the module.

Class ID: Unique module ID.

Auto generate on save: Enables TwinCAT to generate the ClassID via the module parameters during saving. If the ClassID changes during import of the binary modules, the corresponding ClassIDs have to be adjusted. Thus, TwinCAT can detect the interface change.

Choose Image: Add a 16x16 pixel bitmap symbol.

Reset image: Reset the module image to the default value.

Init sequence: Start the state machine. The selection options with 'late' in the name are internal. (See Object [▶ 130] of the instance configurator for further information.)

Instantiable in RT Context: Indicates whether this module can be instantiated under real-time context; see TwinCAT Module Class Wizard [▶ 86]

Defining the contexts of the module

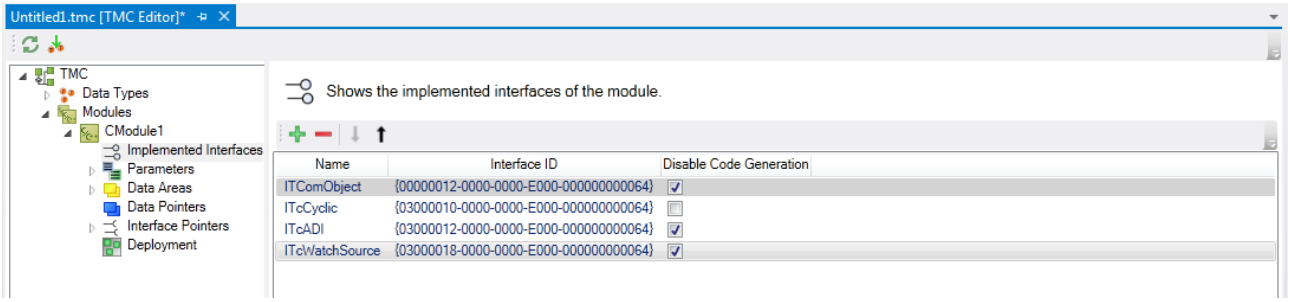
You can add (**+**) or remove (**-**) contexts for the module. Edit the order with the help of the arrows.
The context ID must be an integer other than 0.

Optional Properties

A table of name, value and description for annotating the module.
This information is provided within the TMC and also TMI files.
TwinCAT functions as well as customer programs can use these properties.

11.3.4.1 Implemented Interfaces

Implemented Interfaces: View and edit the implemented interfaces of the module.

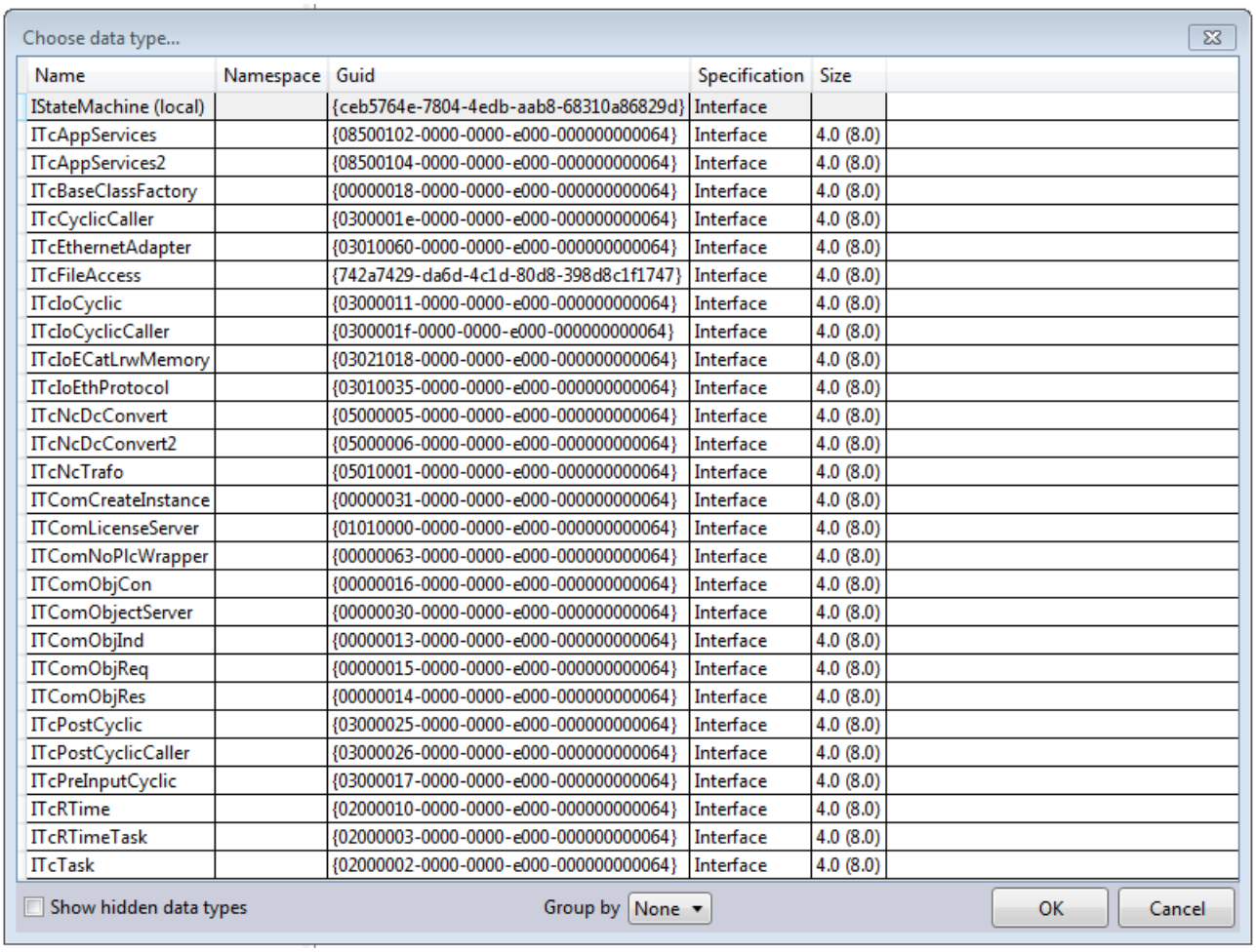


Name: Name of the interface.

Interface ID: Unique ID of the interface.

Disable Code Generation: Enable/disable code generation.

You can add (**+**) or remove (**-**) contexts for the module. Edit the order with the help of the arrows.



11.3.4.2 Parameters

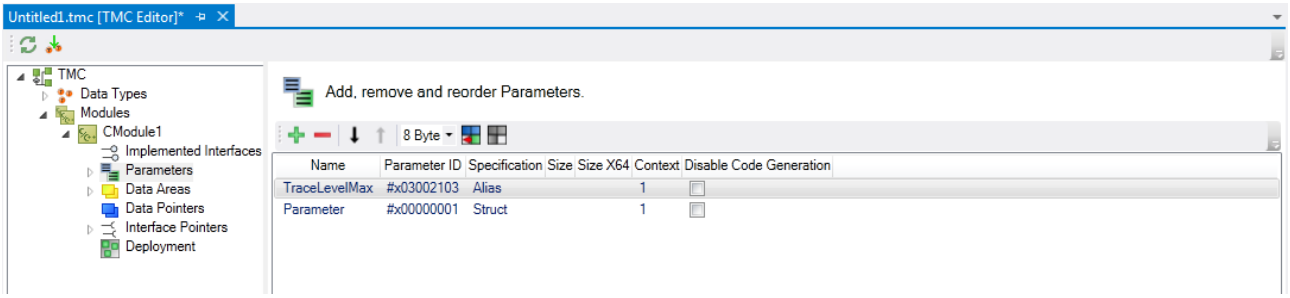
A TcCOM module instance is defined through various parameters.

TwinCAT supports three types of Parameter IDs (PTCID) in the section Configuring the parameter ID [▶ 117].

- "User defined" (default value for new parameters): A unique parameter ID is generated, which can be used in the user code or in the instance configuration for specifying the parameter.
- "Predefined...": Special PTCIDs provided by the TwinCAT 3 system (e.g. TcTraceLevel).
- "Context-based…": Automatically assign values of the configured context [▶ 131] to this parameter. The selected property is applied to the PTCPID. It overwrites the defined standard parameters and the instance configuration parameter (parameter (Init)).

The parameters and their configuration are described in more detail below.

Parameters: Shows the implemented parameters of the module.



Symbol	Function
	Add a new parameter
	Deletes the selected type
	Moves the selected element down one position
	Moves the selected element up one position
	Select byte alignment
	Align the selected data type
	Reset data format of the selected data type

Name: Name of the interface.

Parameter ID: Unique ID of the parameter.

Specification: Data type of the parameter.

Size: Size of the parameter. Other sizes are possible for x64.

Context: Context ID of the parameter.

Disable Code Generation: Enable/disable code generation.

11.3.4.2.1 Add / modify / delete parameters

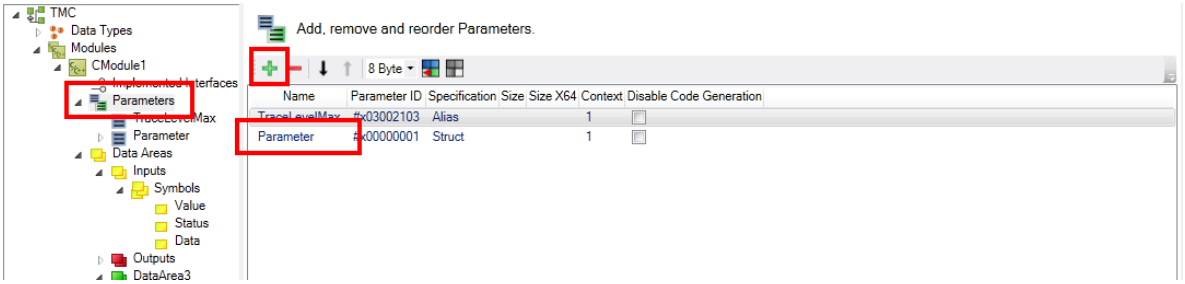
The properties and functionalities of a TwinCAT class can be added, edited and deleted with the aid of the TwinCAT Module Class (TMC) Editor.

This article describes:

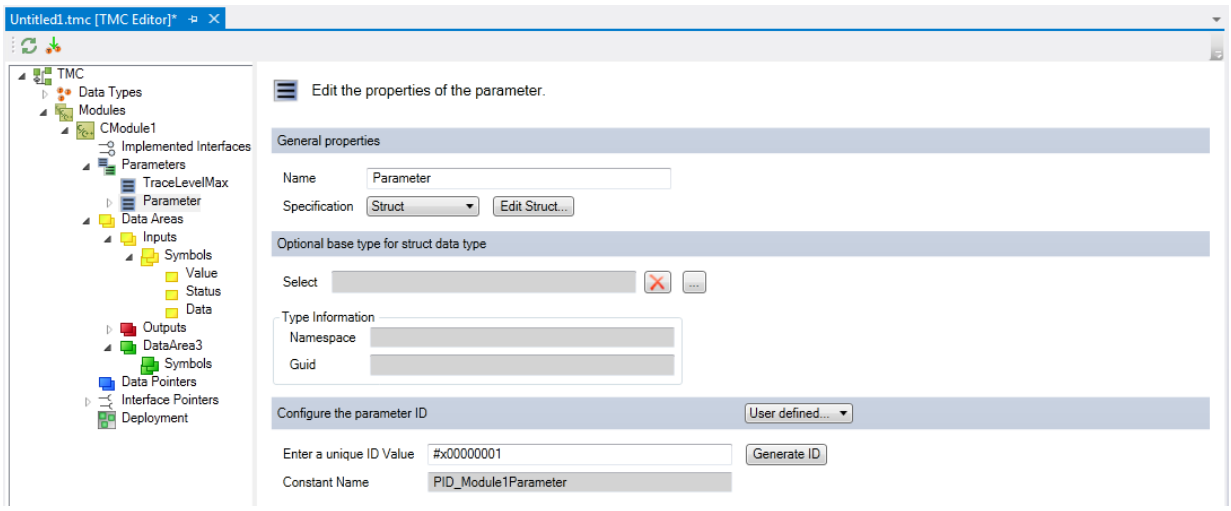
- Step 1: Create a new parameter [▶ 113] in the TMC file.
- Step 2: Start the TwinCAT TMC Code Generator [▶ 114] to generate code for the module description in the TMC file.
- Step 3: Observe the transitions of the state machine [▶ 115]

Step 1: Create new parameters

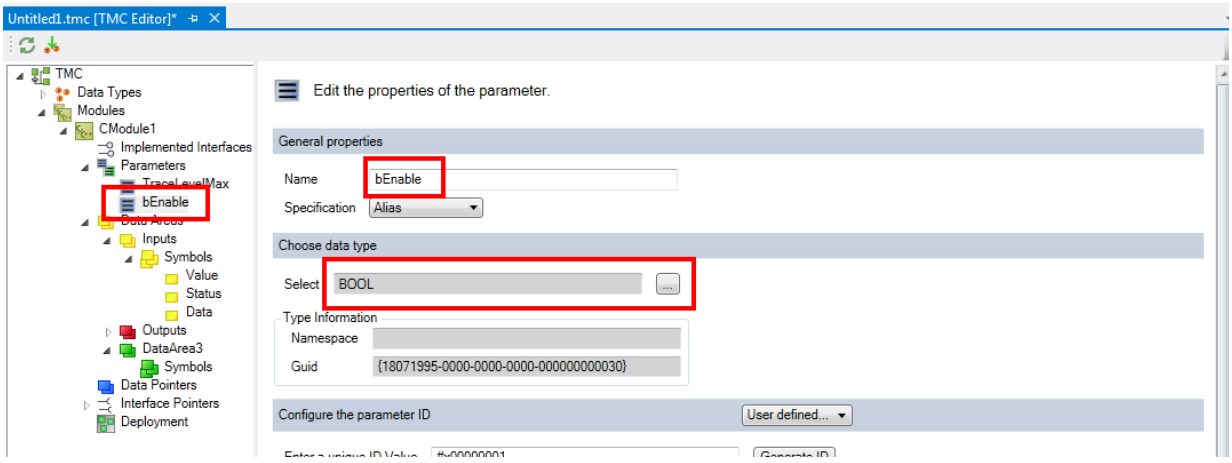
1. After starting the TMC Editor, select the target **Parameters**.
2. Extend the list of parameters by a new parameter by clicking on the **+** button **Add a new parameter**.
   ⇨ A new "Parameter" is then listed as a new entry:



3. Select **Parameter** in the left-hand tree or double-click on the red-marked "Parameter3" or select the node in the tree to obtain details of the new parameter.



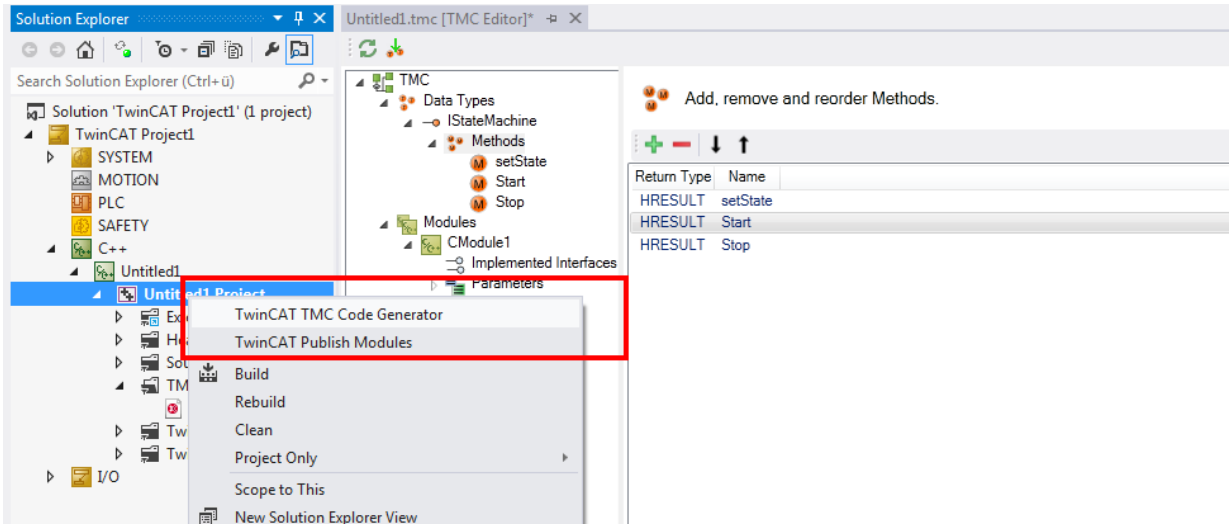
4. Configure the parameter as well as the Data Types [▶ 91].
5. Give it a more meaningful name – in this sample "bEnable" – and select the data type "BOOL".



6. Save the changes you have made in the TMC file.

Step 2: Start the TwinCAT TMC Code Generator to generate code for the module description.

1. Right-click on your project file and select **TwinCAT TMC Code Generator** to receive the parameters in your source code:



⇨ You can see the parameter declaration in the header file "Module1.h" of the module.

```
///<AutoGeneratedContent id="Members">
    TcTraceLevel m_TraceLevelMax;
    bool m_bEnable;
    Module1Inputs m_Inputs;
    Module1Outputs m_Outputs;
    Module1DataArea3 m_DataArea3;
    ITcCyclicCallerInfoPtr m_spCyclicCaller;
///</AutoGeneratedContent>
```

⇨ The implementation of the new parameter can be found in the get and set methods of the module class "module1.cpp".

```
IMPLEMENT_ITCOMOBJECT(CModule1)
IMPLEMENT_ITCOMOBJECT_SETSTATE_LOCKOP2(CModule1)
IMPLEMENT_ITCADI(CModule1)
IMPLEMENT_ITCWATCHSOURCE(CModule1)

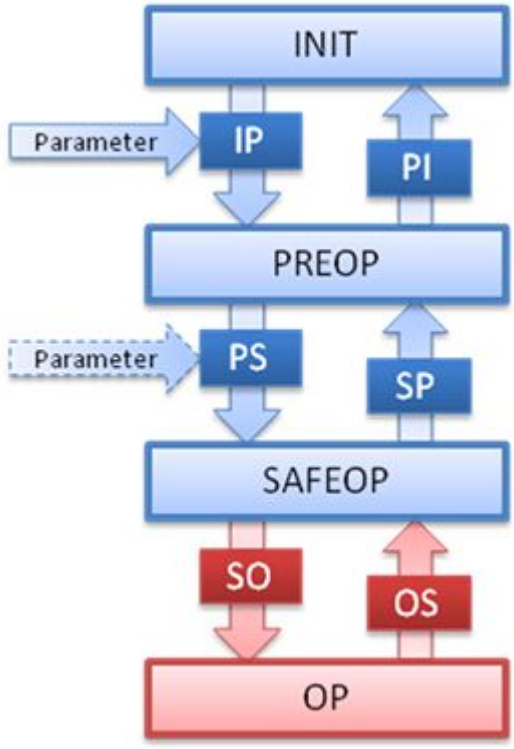

////////////////////////////////////////////////////////////////////////////////
// Set parameters of CModule1
BEGIN_SETOBJPARA_MAP(CModule1)
    SETOBJPARA_DATAAREA_MAP()
///<AutoGeneratedContent id="SetObjectParameterMap">
    SETOBJPARA_VALUE(PID_TcTraceLevel, m_TraceLevelMax)
    SETOBJPARA_VALUE(PID_Module1bEnable, m_bEnable)
    SETOBJPARA_ITFPTR(PID_Ctx_TaskOid, m_spCyclicCaller)
///</AutoGeneratedContent>
END_SETOBJPARA_MAP()

////////////////////////////////////////////////////////////////////////////////
// Get parameters of CModule1
BEGIN_GETOBJPARA_MAP(CModule1)
    GETOBJPARA_DATAAREA_MAP()
///<AutoGeneratedContent id="GetObjectParameterMap">
    GETOBJPARA_VALUE(PID_TcTraceLevel, m_TraceLevelMax)
    GETOBJPARA_VALUE(PID_Module1bEnable, m_bEnable)
    GETOBJPARA_ITFPTR(PID_Ctx_TaskOid, m_spCyclicCaller)
///</AutoGeneratedContent>
END_GETOBJPARA_MAP()
```

To add a further parameter, use the TwinCAT TMC Code Generator again.

Step 3: Transitions of the state machine

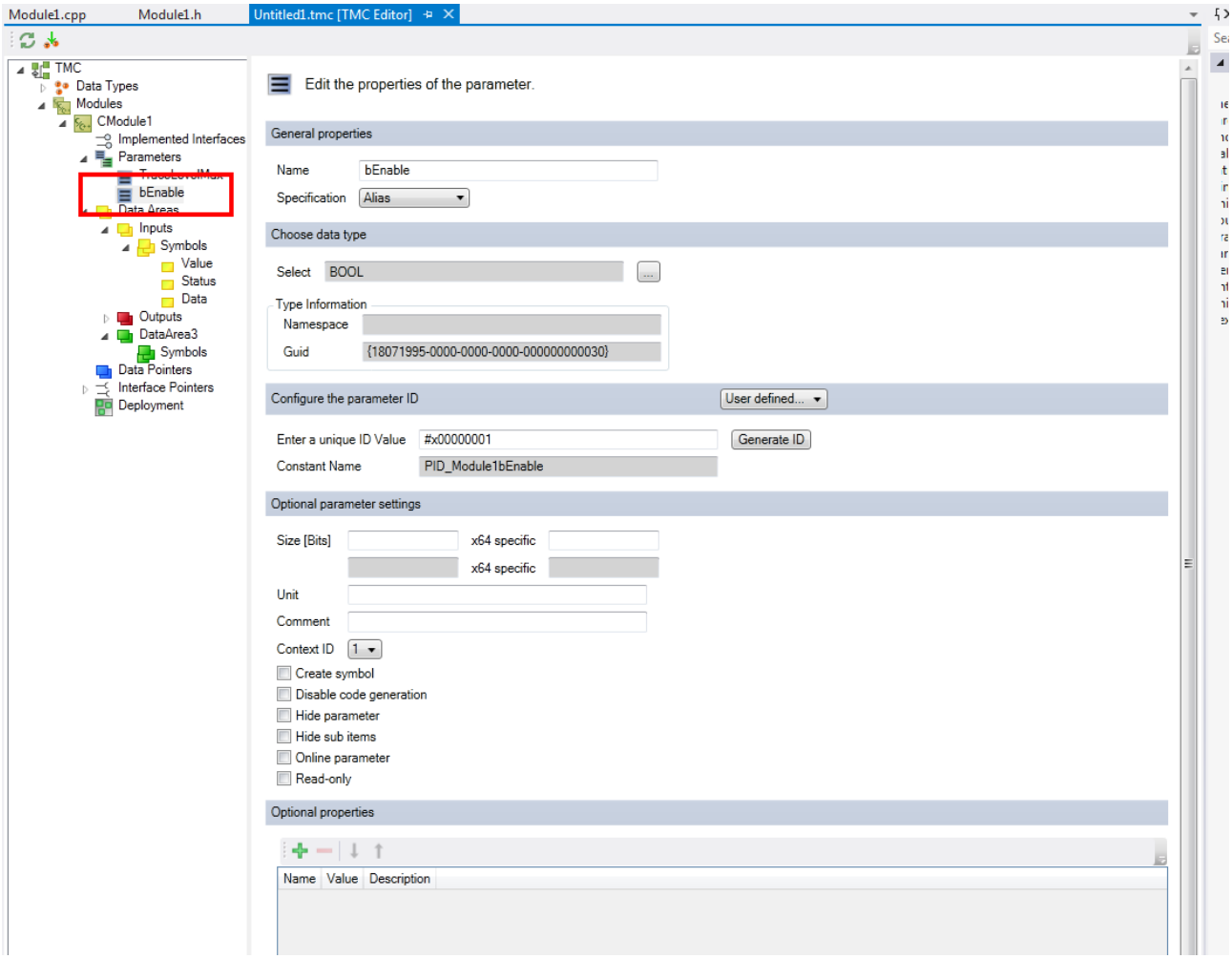
Note the different state transitions of your state machine [▶ 44]:



The parameters are set during the transition from Init to Preop and, if applicable, from Preop to Safeop.

11.3.4.2.2 Parameter properties

Parameter properties: Edit the properties of the parameter.



General properties

Name: Name of the interface.

Specification: Data type of the parameter, see specification.

Select data type

Select: Select data type.

Type information

- **Namespace:** user-defined namespace of the data type.
- **GUID**: unique ID of the data type.

Enter a unique ID Value: Enter a unique ID value, see Parameter [▶ 111].

Constant Name: source code name of the parameter ID.

Optional parameter settings

Size [Bits]: Calculated size in bits (white fields) and in "Byte.Bit" notation (grey fields). A special size configuration is provided for x64.

Unit: a unit of the variable.

Comment: Comment that is visible, for example, in the instance configurator.

Context ID: Context used when accessing parameters by ADS.

Create Symbol: Default setting for ADS icon creation.

Disable Code Generation: Enable/disable the code generation.

Hide parameter: Switches between showing/hiding parameters in the System Manager view.

Hide sub items: If the data type has sub-elements, the System Manager will not allow the sub-elements to be accessed. This should be used, for example, in the case of larger arrays.

Online parameter: Define as online parameter.

Read-only: Switch to read-only access for System Manager.

Optional Properties

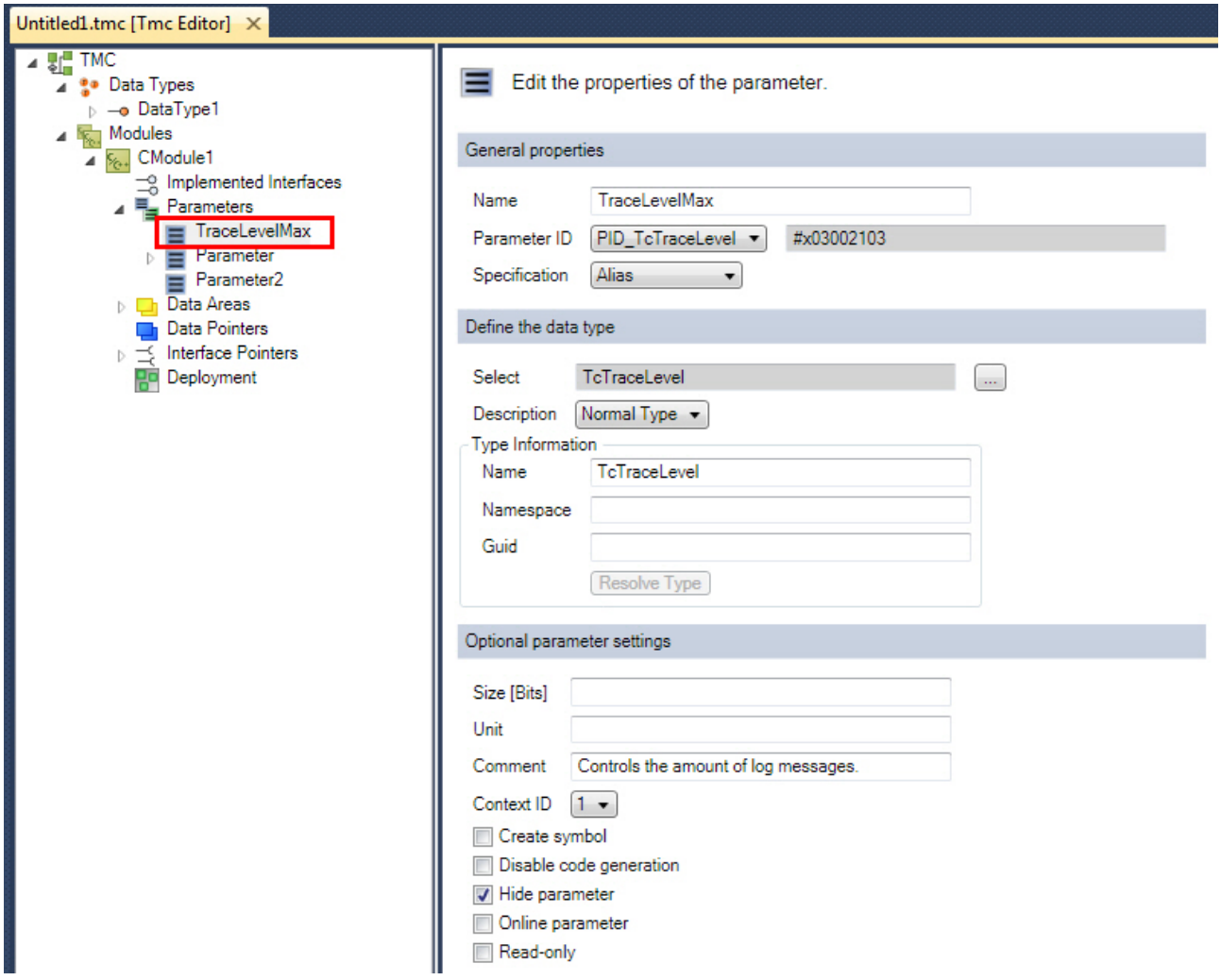
A table of name, value and description for annotating the parameter.
This information is provided within the TMC and also TMI files.
TwinCAT functions as well as customer programs can use these properties.

11.3.4.2.3 TraceLevelMax

TraceLevelMax: Parameter that defines the trace level.
This is a predefined parameter that most TwinCAT module templates provide (except the empty TwinCAT module template).

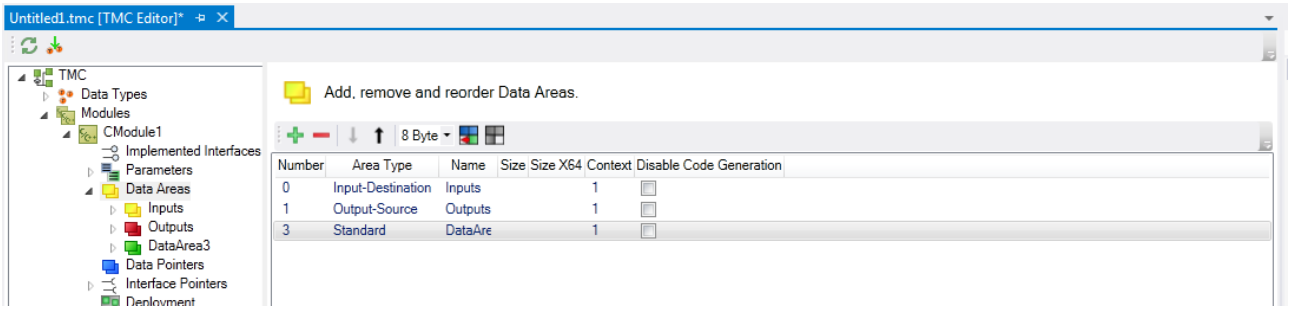
The settings for this parameter should not be changed.

See Module messages for the Engineering (logging / tracing) [▶ 209]



11.3.4.3 Data Areas

Data Areas: Dialog for editing the data areas of your module.



Symbol	Function
	Add a new data area
	Deletes the selected data area
	Moves the selected element down one position
	Moves the selected element up one position
	Select byte alignment
	Align the selected data type
	Reset the data format of the selected area

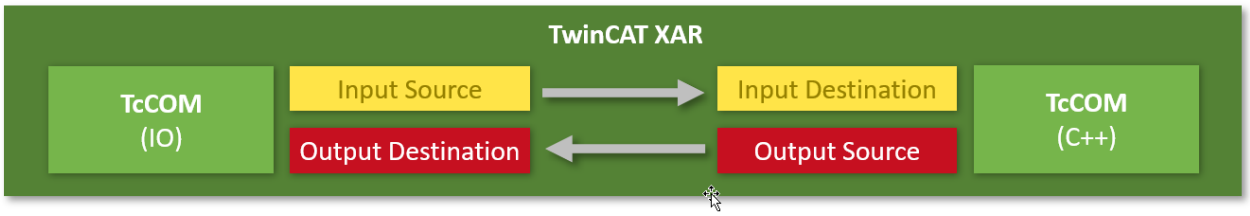
NOTICE

Recursion when setting an alignment

When setting the alignment of a data area, this will be taken as the basis for all of its elements (symbols and also their sub-elements). User-defined alignment will be overwritten.

Number: Number of the data area.

Type: Defines the purpose and location of the data area.
The data areas for mapping (to other modules as well as I/O) are classified into input and output as well as source and destination. The chart illustrates this:



Name: Name of the data area.

Size: Size of the parameter; other sizes are possible for x64.

Context: Displays the context ID.

Disable Code Generation: Enable/disable code generation.

11.3.4.3.1 Add / modify / delete data areas and variables

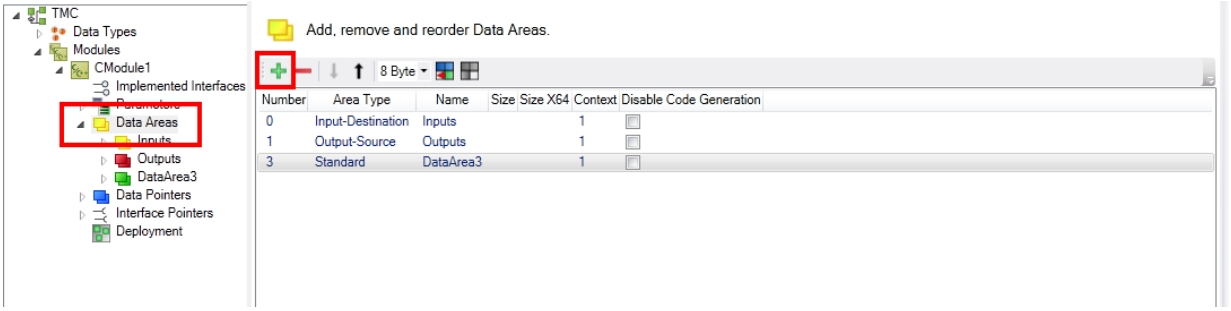
The properties and functionalities of a TwinCAT class can be added, edited and deleted with the aid of the TwinCAT Module Class (TMC) Editor.

This article describes:

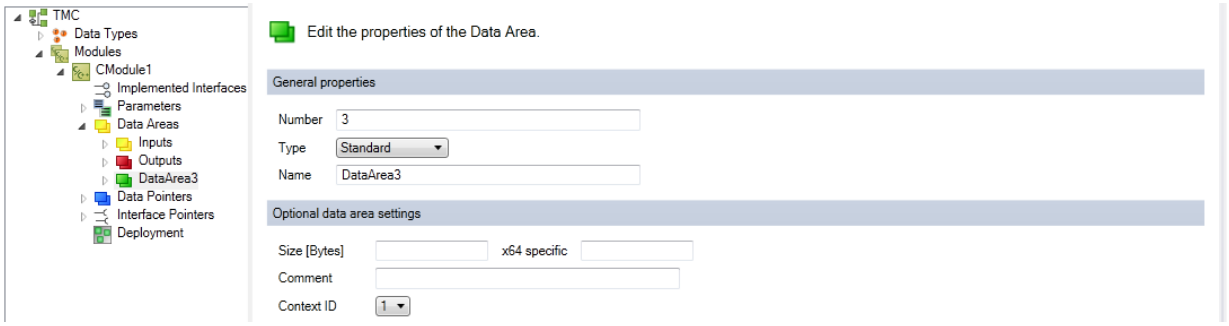
- Creation of a new data area in the TMC file.
- Creating new variables in a data area.
- For example, edit the name or data type of variables existing in the TMC file.
- Delete existing variables from the TMC file.

Creating a new data area

1. After starting the TMC Editor, select the **Data Areas** node of the module.
2. Click on the **+** button, thus creating a new data area.



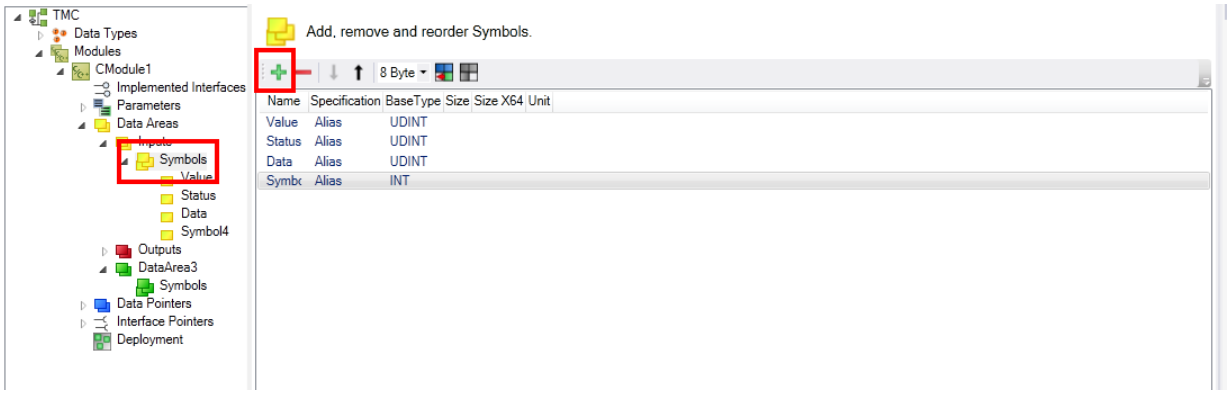
3. In order to obtain the properties of the data area, double-click on the table or on the node.



4. Rename the data area.

Creating a new variable

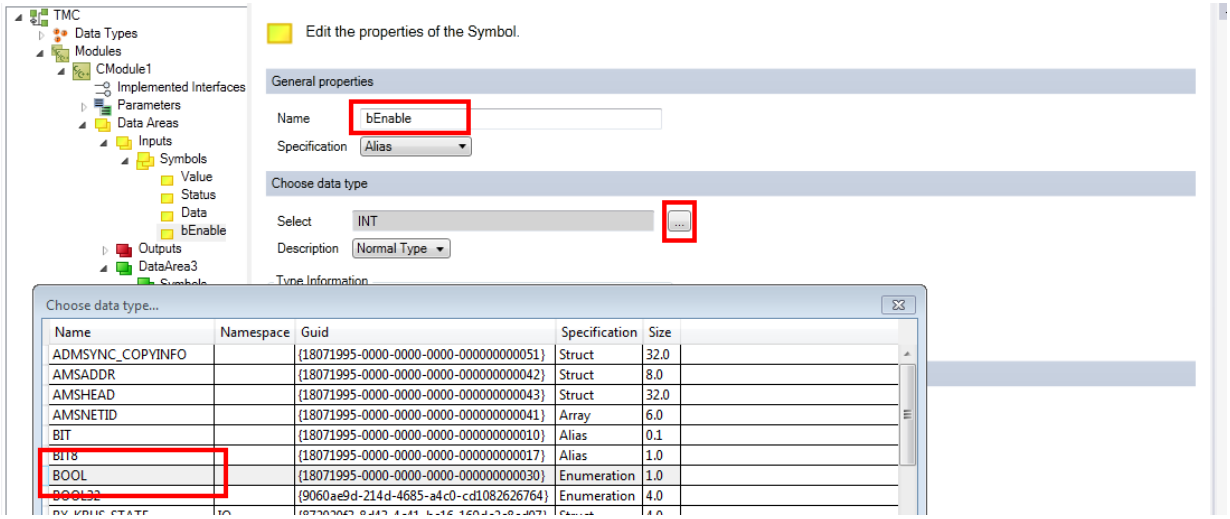
1. Select the sub-node **Symbols** of the data area.
2. Extend this data area by a new variable by clicking on the **+** button. A new entry "Symbol4" is then listed.



Editing the name or data type of existing variables

1. Select the sub-node **Symbol4** or double-click on the row. The variable properties are shown.

2. Enter a new name, e.g. "bEnableJob" and change the type to BOOL.

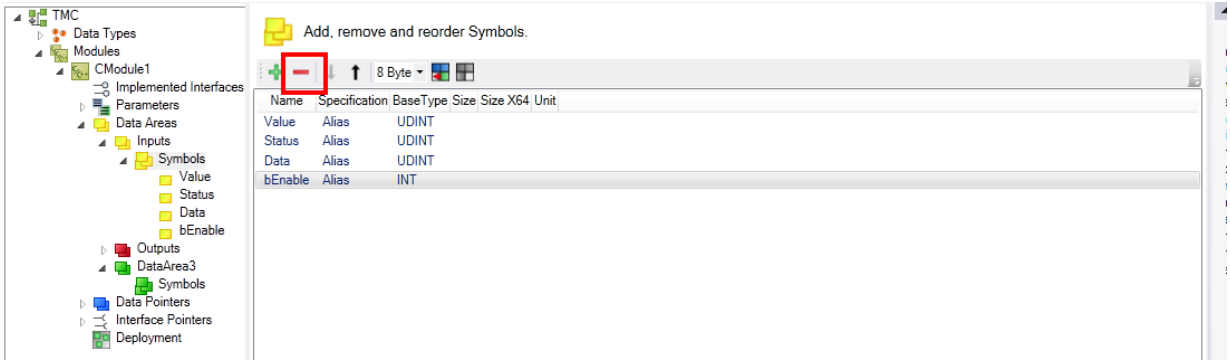


⇨ The new variable "bEnableJob" is created in the data area "Input".

Remember to run the TMC Code Generator again.

Deleting existing variables

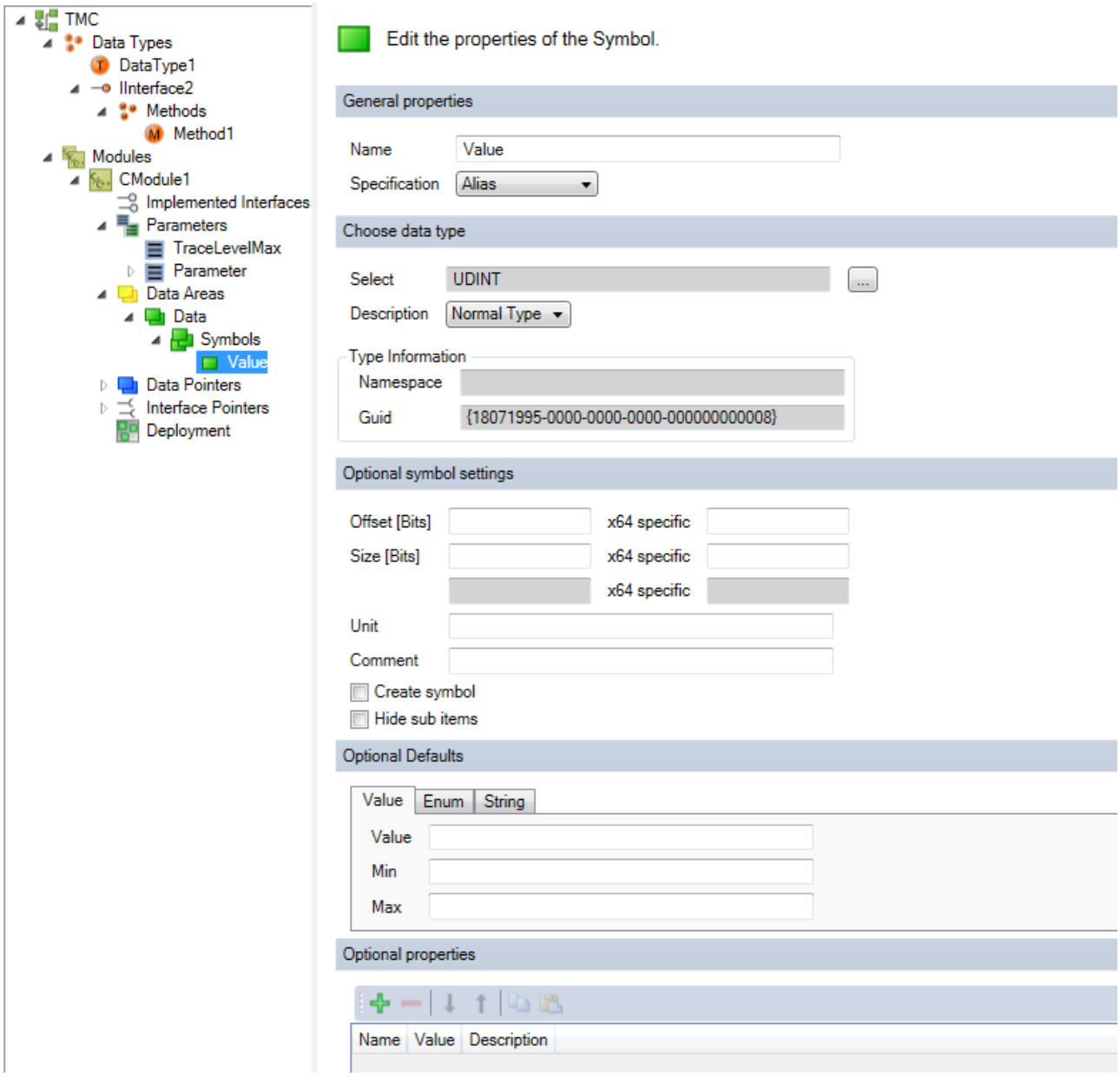
1. To delete existing variables from the data area, select the variable and then click on the delete icon: in this sample, select "MachineStatus1" and click on the **Delete icon**.



2. Run the TMC Code Generator again.

11.3.4.3.2 Data Areas Properties

Data Areas Properties: Dialog for editing the data area properties.



General properties

Number: Number of the data area.

Type: Defines the purpose and location of the data area. The following are available:

Linkable data areas in the System Manager:

- Input-Source
- Input-Destination
- Output-Source
- Output-Destination
- Retain-Source (for use with NOV-RAM memory, see appendix [▶ 326])
- Retain-Destination (for internal use)

Further data areas:

- Standard (visible but not linkable in the System Manager)
- Internal (for internal module symbols, which can be reached via ADS but are not visible in the System Manager)
- MArea (for internal use)

- Not specified (same as standard)

Name: Name of the data area.

Optional parameter settings

Size [Bytes]: Size in bytes. A special size configuration is provided for x64.

Comment: Optional comment that is visible, for example, in the instance configurator.

Context ID: Context ID of all symbols of this data area; used for the determination of the mapping.

Data type name: If specified, a data type with the specified name is created in the type system.

Create Symbol: Default setting for ADS icon creation.

Disable Code Generation: Enable/disable the code generation.

Optional Defaults

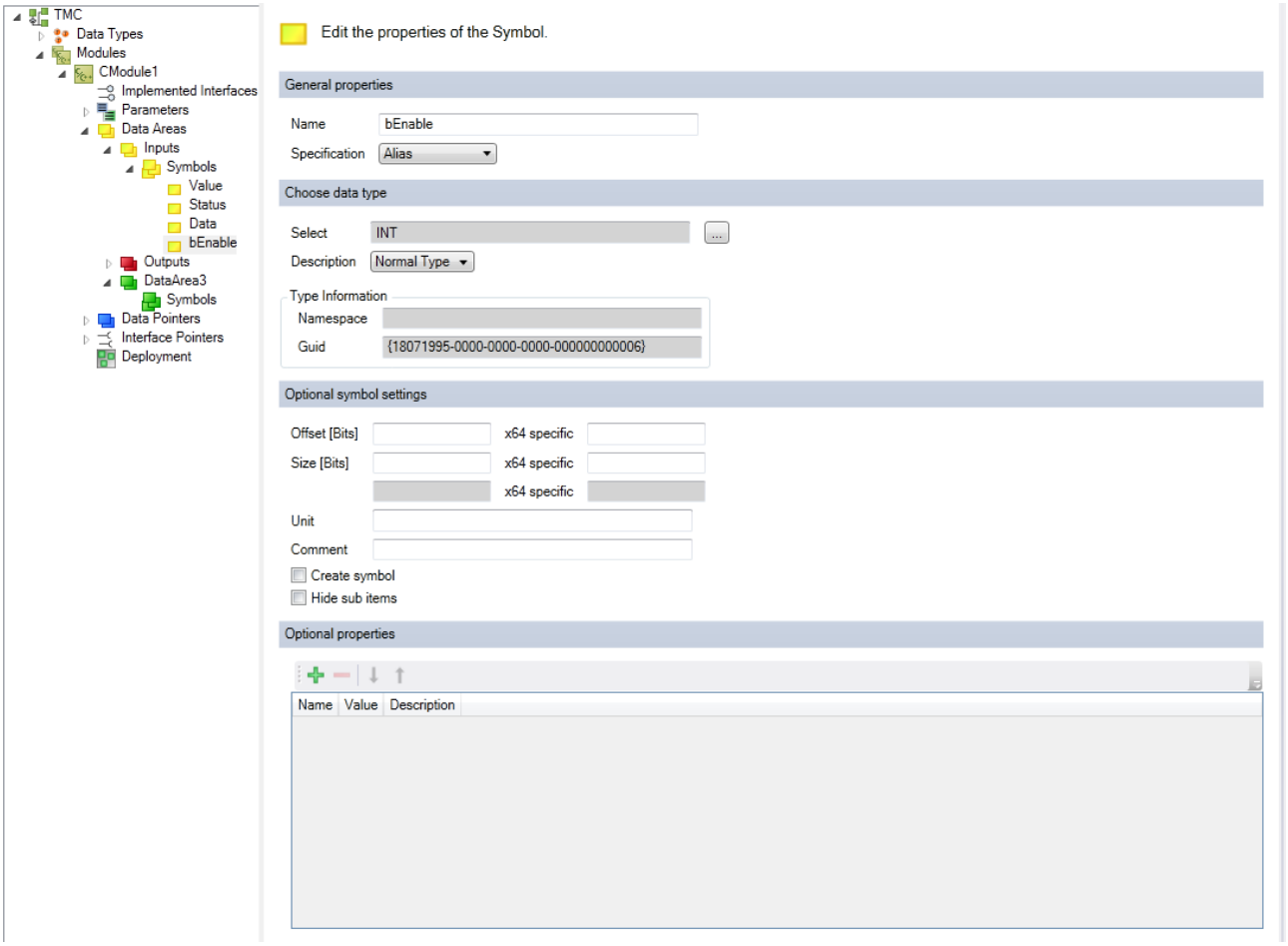
Depending on data type the default could be defined.

Optional Properties

A table of name, value and description for annotating the data area.
This information is provided within the TMC and also TMI files.
TwinCAT functions as well as customer programs can use these properties.

11.3.4.3.3 Symbol Properties

Symbols: Dialog for the editing of the symbols of the data area.



General properties

Name: Name of the symbol.

Specification: Data type of the symbol, see Data type properties [▶ 103].

Select data type

Select: Select data type – it can be a basic TwinCAT data type or a user-defined data type.

Description: Define whether the type is the following:

- Normal type
- Pointer
- Pointer to pointer
- Pointer to pointer to pointer
- a reference

Type information

- **Namespace:** Namespace for selected data type.
- **GUID**: unique ID of the data type.

Optional data type settings

Offset [Bits]: Offset of the symbol within the data area; a different offset can be defined for the x64 platform.

Size [Bits]: Size in bits, if specified. A different size can be defined for the x64 platform.

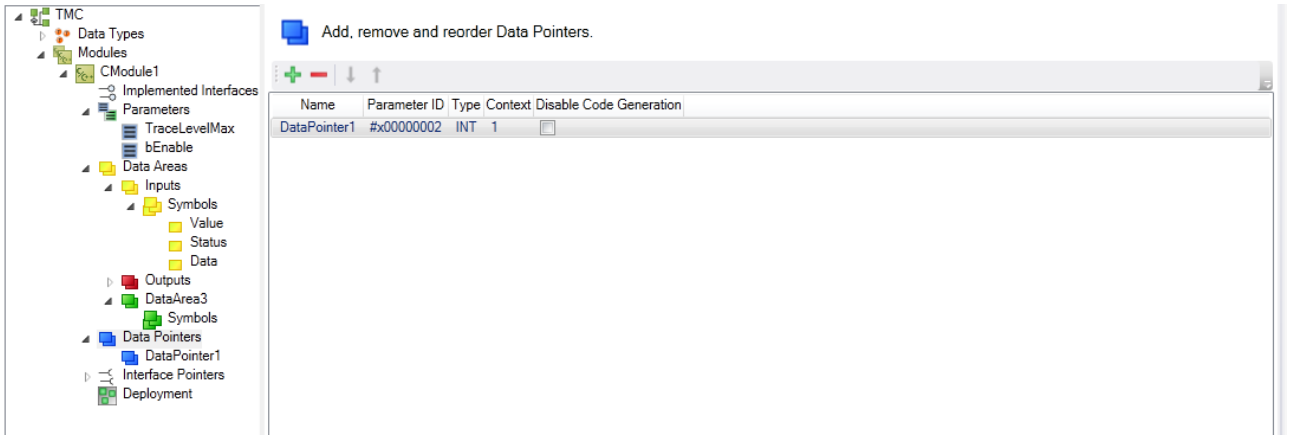
Comment: Optional comment that is visible, for example, in the instance configurator.

Create Symbol: Default setting for ADS icon creation.

Hide sub items: If a variable has sub-elements, then the System Manager will not allow the sub-elements to be accessed. This should be used, for example, in the case of larger arrays.

11.3.4.4 Data Pointers

Data Pointer: Dialog for editing the data pointers of your module.



Symbol	Function
	Add a new data pointer
	Deletes the selected data pointer
	Moves the selected element down one position

↑	Moves the selected element up one position

Name: Name of the data pointer.

Parameter ID: Unique ID of the parameter.

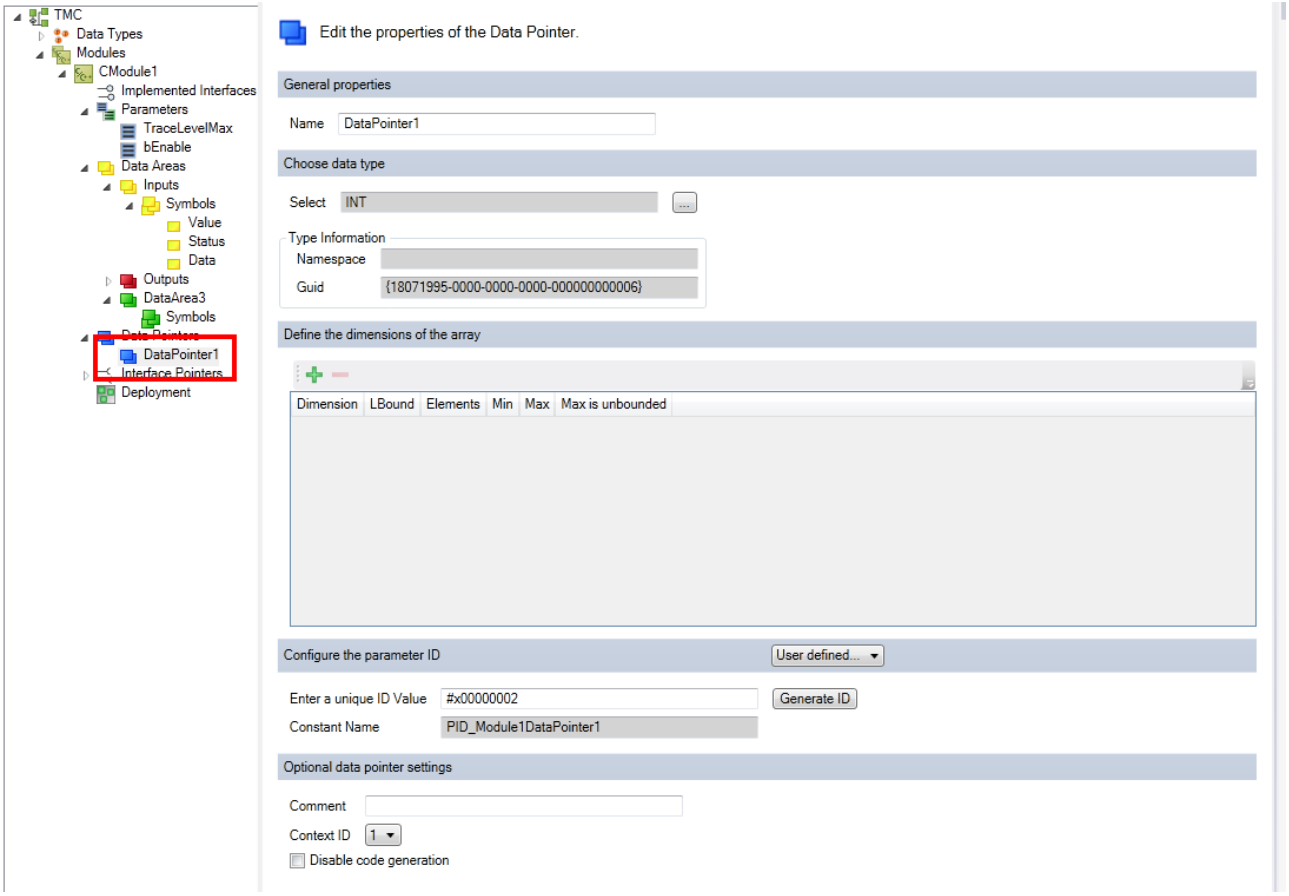
Type: Defines the pointer type.

Context: Displays the context ID.

Disable Code Generation: Enable/disable code generation.

11.3.4.4.1 Properties

Data Pointer Properties: Edit the properties of the data pointer.



General properties

Name: Name of the data pointer.

Define the data type

Select: Select data type.

Type information

- **Name**: Name of the selected data type.
- **GUID**: unique ID of the data type.

Define the dimension of the array

See chapter Array [▶ 106].

Configuring the parameter ID

Enter a unique ID Value: Enter a unique ID value, see Parameter [▶ 111].

Constant Name: source code name of the parameter ID.

Optional data pointer settings

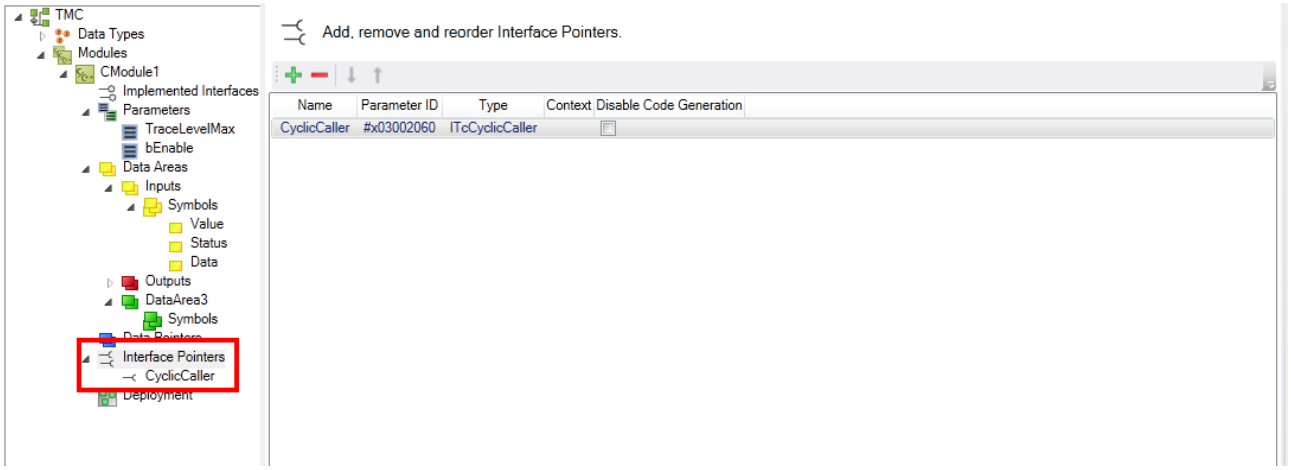
Comment: Comment that is visible, for example, in the instance configurator.

Context ID: Context ID of the data pointer.

Disable Code Generation: Enable/disable the code generation.

11.3.4.5 Interface Pointers

Interface Pointers: Add, remove and rearrange interface pointers.



Symbol	Function
	Add interface pointers
	Deletes the selected pointer
	Moves the selected element down one position
	Moves the selected element up one position

Name: Name of the interface.

Parameter ID: Unique ID of the interface pointer.

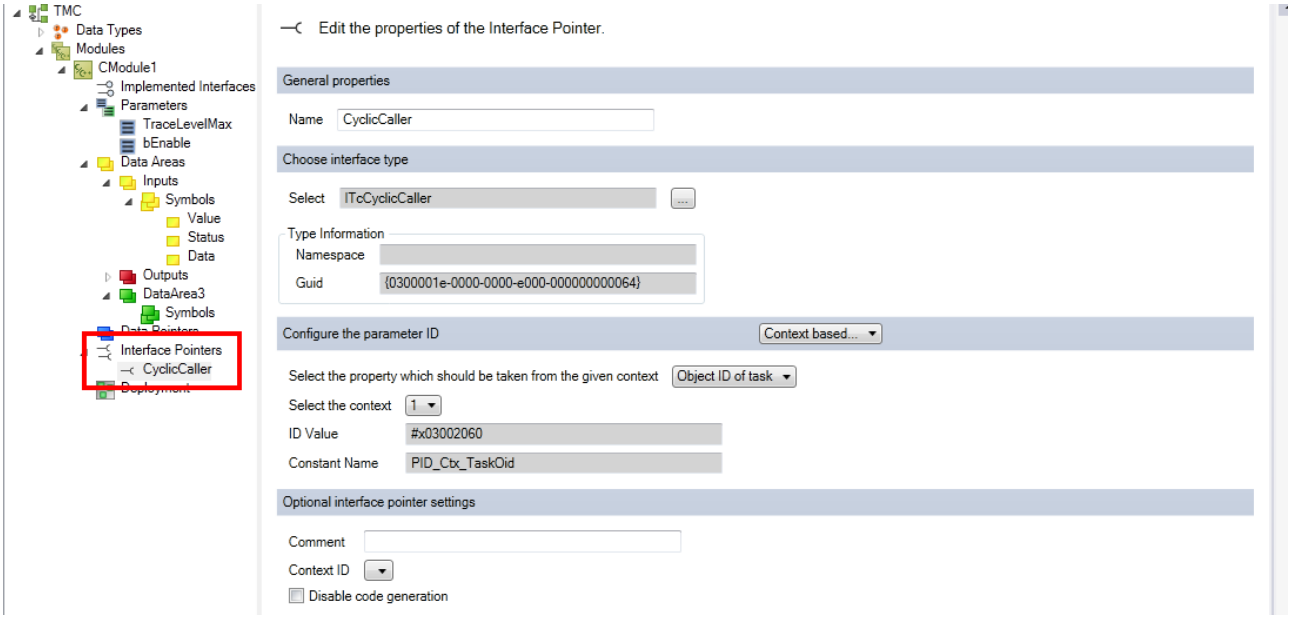
Type: Type of the interface pointer.

Context: Context of the interface.

Disable Code Generation: Enable/disable the code generation.

11.3.4.5.1 Properties

Interface Pointer Properties: Edit the properties of the interface pointer.



General properties

Name: Name of the interface pointer.

Select the basic interface

Select: Selection of the interface.

Type information

- **Namespace:** Namespace of the interface.
- **GUID**: Unique ID of the interface.

Configure the parameter ID

See Parameters [▶ 111].

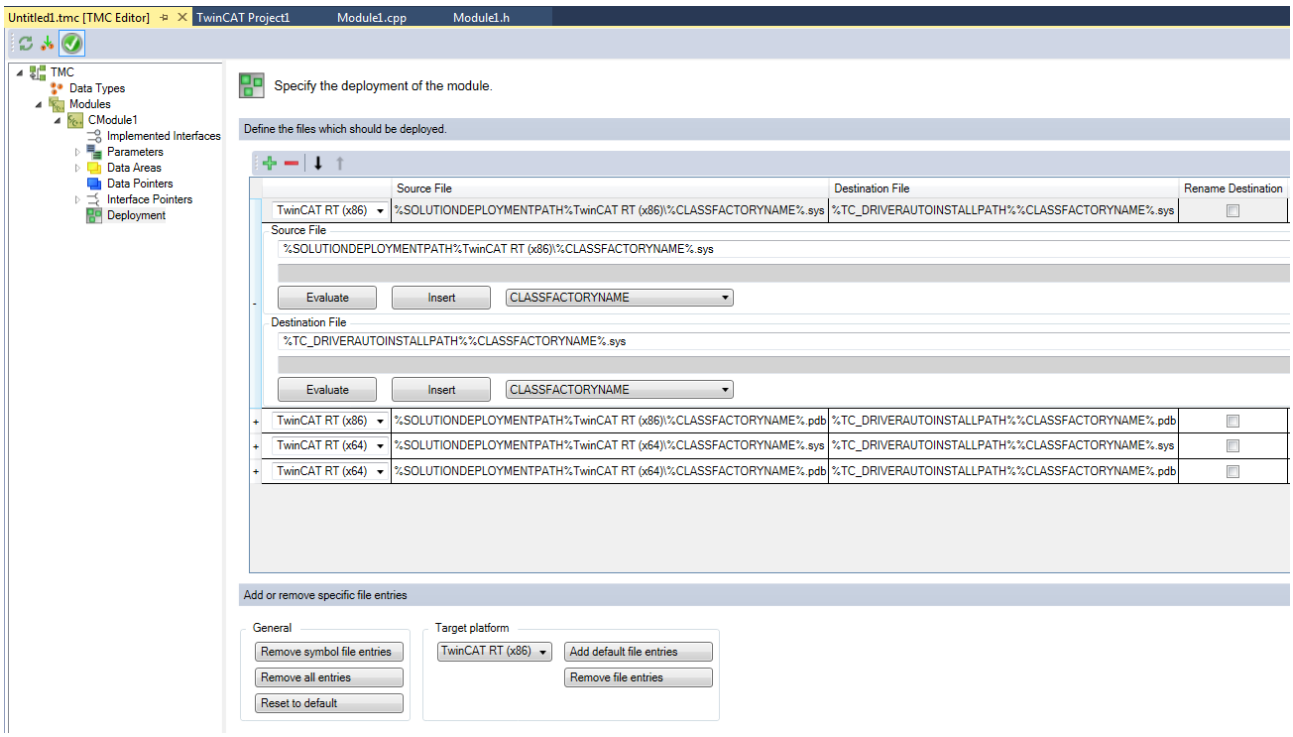
Comment: Optional

Context ID: Context ID of the interface pointer.

Disable Code Generation: Enable/disable the code generation.

11.3.4.6 Deployment

Deployment: Specify storage locations for the provided modules on the target system. The entries are empty for versioned C++ projects with their modules and are not needed.



Symbol	Function
	Add a new file entry
	Delete a file entry
	Moves the selected element down one position
	Moves the selected element up one position

This dialog enables configuration of the source and target file, which are transferred to the target system for the respective platforms.

Define the files, which should be deployed.

Source File: Path to the source files.

Destination file: Path to the binary files.

Rename Destination: Destination file will be renamed before the new file is transferred. Since this is required for Windows 10, it is done implicitly.

The individual entries can be expanded and collapsed by the **+** or **–** respectively at the beginning.

Evaluate: Puts the calculated value into the text field for verification.

Insert: Adds the variable name selected in the dropdown list.

Add or remove specific file entries

Remove symbol file entries: Removes the entries for the provision of symbol files (PDB).

Remove all entries: Removes all entries.

Reset to default: Sets the standard entries.

Add default file entries: Adds the entries for the selected platform.

Remove file entries: Removes the entries for the selected platform.

Source and target paths for the allocation may contain virtual environment variables, which are resolved by the TwinCAT XAE / XAR system.

The following table shows the list of these supported virtual environment variables.

Virtual environment variable	Registry entry (REG_SZ) under key \HKLM\Software\Beckhoff\TwinCAT3	Default value (Windows)	Default value (TwinCAT/BSD®)
%TC_INSTALLPATH%	InstallDir	C:\TwinCAT\3.**1**\	/usr/local/etc/TwinCAT/3.**1**/
%TC_CONFIGPATH%	ConfigDir	C:\TwinCAT\3.**1**\Config\	/usr/local/etc/TwinCAT/3.**1**/Config/
%TC_TARGETPATH%	TargetDir	C:\TwinCAT\3.**1**\Target\	/usr/local/etc/TwinCAT/3.**1**/Target/
%TC_SYSTEMPATH%	SystemDir	C:\TwinCAT\3.**1**\System\	/usr/local/etc/TwinCAT/3.**1**/System/
%TC_BOOTPRJPATH%	BootDir	C:\TwinCAT\3.**1**\Boot\	/usr/local/etc/TwinCAT/3.**1**/Boot/
%TC_RESOURCEPATH%	ResourceDir	C:\TwinCAT\3.**1**\Target\Resource\	/usr/local/etc/TwinCAT/3.**1**/Target/Resource/
%TC_REPOSITORYPATH%	RepositoryDir	C:\TwinCAT\3.**1**\Repository\	/usr/local/etc/TwinCAT/3.**1**/Repository
%TC_DRIVERPATH%	DriverDir	C:\TwinCAT\3.**1**\Driver\	not available
%TC_DRIVERAUTOINSTALLPATH%	DriverAutoInstallDir	C:\TwinCAT\3.**1**\Driver\AutoInstall\	/usr/local/etc/TwinCAT/3.**1**/
%CLASSFACTORYNAME%		<Name of the Class Factory>	<Name of the Class Factory>

11.4 TwinCAT Module Instance Configurator

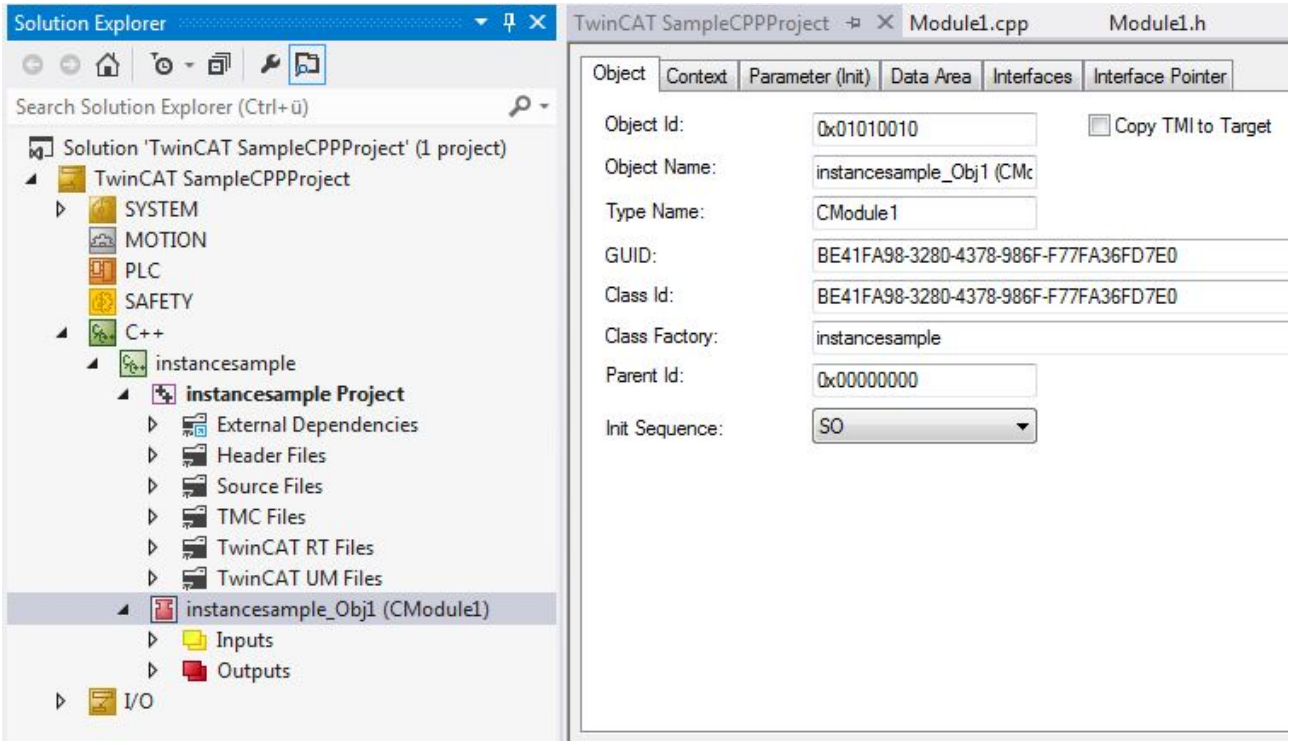
The TwinCAT 3 Modules Class (TMC) editor described above defines drivers at class level. These are instantiated and have to be configured via the TwinCAT 3 instance configurator.

The configuration applies to the context (including the task calling the module), parameters and pointers.

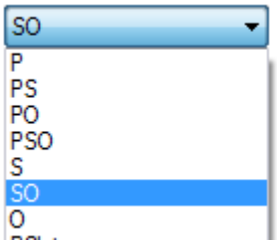
Instances of C++ classes are created by right-clicking on the C++ project folder; see quick start [▶ 58]. This chapter describes the configuration of these instances in detail.

Double-click on the generated instance to open the configuration dialog with several windows.

11.4.1 Object



- **Object Id:** The object ID used for identifying this instance in the TwinCAT system.
- **Object Name:** Name of the object used for displaying the instance in the Solution Explorer tree.
- **Type Name:** Type information (class name) of the instance.
- **GUID:** Module classes GUID.
- **Class Id:** Class ID of the implementation class (GUID and ClassId are usually identical).
- **Class Factory:** Refers to the driver that provides the class factory used to create an instance of the module.
- **Parent Id:** Contains the ObjectID of the parent, if available.
- **Init Sequence:** Specifies the initialization states for determining the startup behavior of the interacting modules. See TwinCAT module state machine [▶ 44] for detailed description of the state machine.



Specifying the startup behavior of several TcCOM instances

TcCOM instances can refer to each other - e.g. for the purpose of interaction via data or interface pointers. To determine the startup behavior, the **Init Sequence** specifies states to be "held" by each TcCOM instance for all other modules.

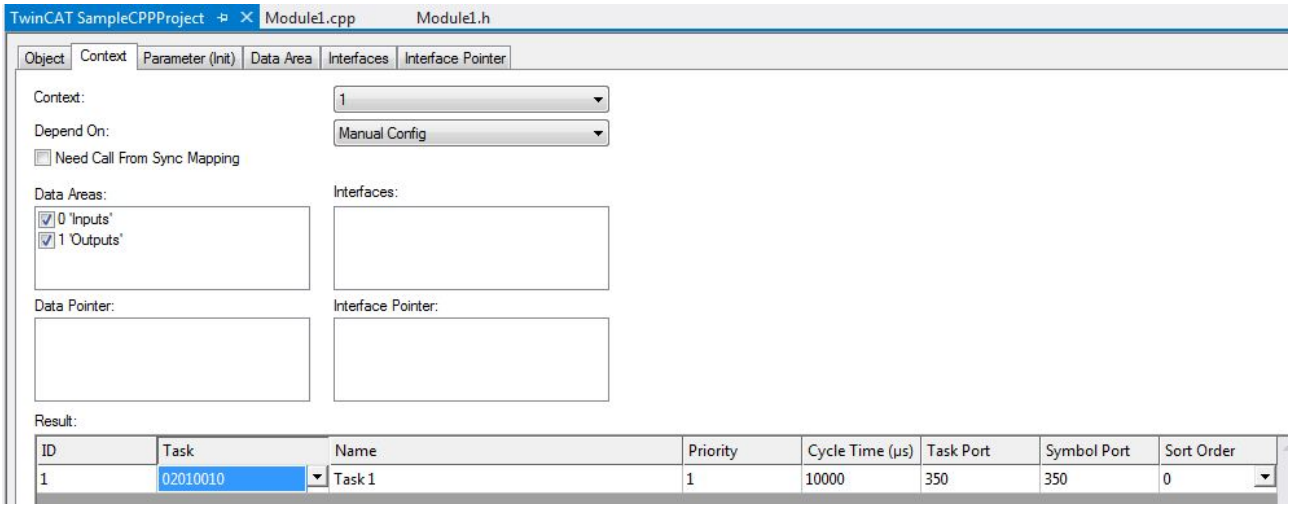
The name of an Init Sequence consists of the short name of the TcCOM state machine. If the short name of a state (I, P, S, O) is included in the name of the Init Sequence, the modules will wait in this state, until all other modules have reached at least this state. In the next transition the module can refer to all other module instances, in order to be in this state as a minimum.

If, for example, a module has the Init Sequence "PS", the IP transitions of all other modules are executed, so that all modules are in "Preop" state.

This is followed by the PS transition of the module, and the module can rely on the fact that the other modules are in "Preop" state.

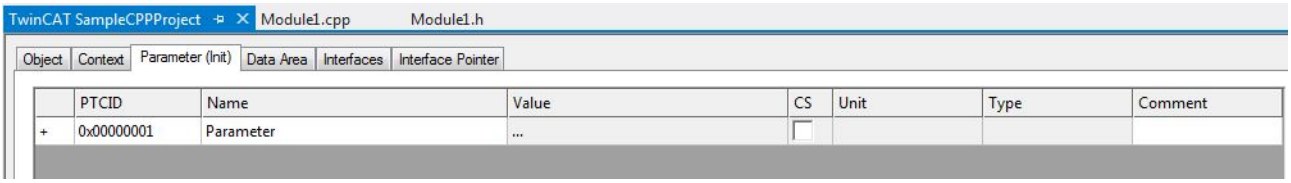
- **Copy TMI to target:** Generating the TMI (TwinCAT Module Instance) file and transferring it to the target.

11.4.2 Context



- **Context:** Select the context to be configured (see TMC Editor for adding different contexts).
  A data area is assigned to a context.
- **Data Areas / Interfaces / Data Pointer and Interface Pointer:** Each instance can be configured to have or not have elements defined in TMC.
- **Result Table:** List of the IDs that need to be configured. At least the context ("Task" column) of the task must be configured accordingly.

11.4.3 Parameter (Init)

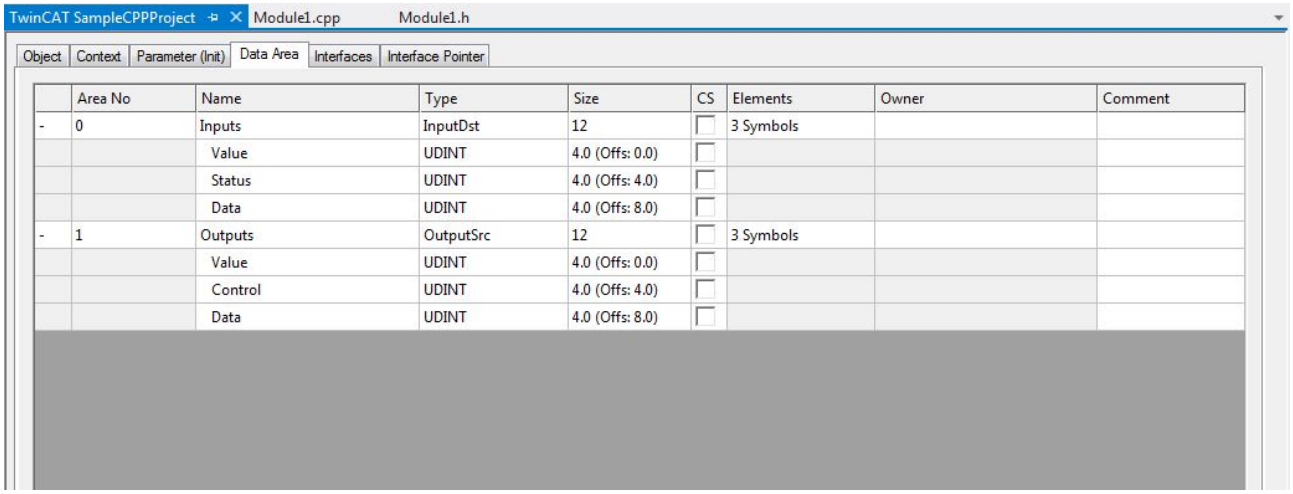


The list of all parameters (as defined in TMC) can be initialized with values for each instance.

Special ParameterIDs (PTCID) are used to set the values automatically. These are configured using the TMC Editor's parameter dialog window as described in Parameters [▶ 111].

The CS (CreateSymbol) checkbox creates the ADS symbol for each parameter, so it can be accessed from outside.

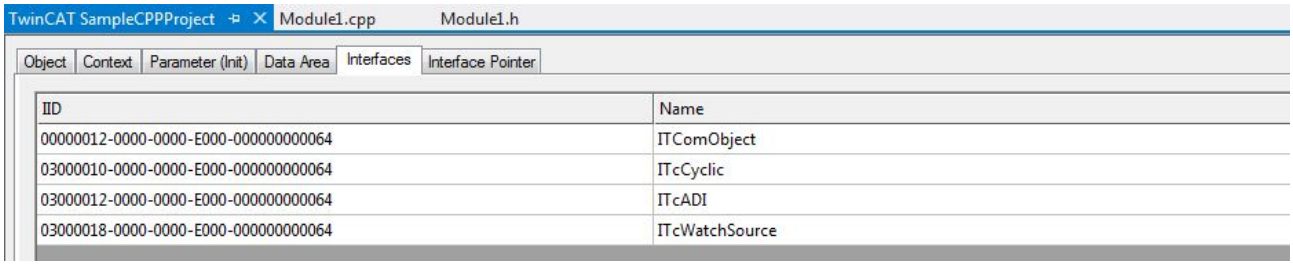
11.4.4 Data Area



List of all data areas and their variables (as defined in TMC).

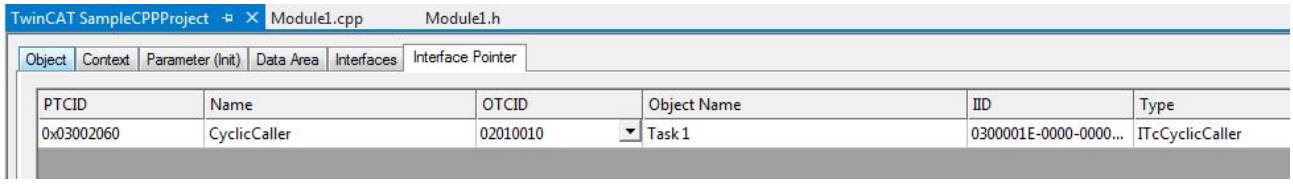
The CS (CreateSymbol) checkbox creates the ADS symbol for each parameter, so the variable is accessible from the outside.

11.4.5 Interfaces



List of all implemented interfaces (as defined in TMC).

11.4.6 Interface Pointer

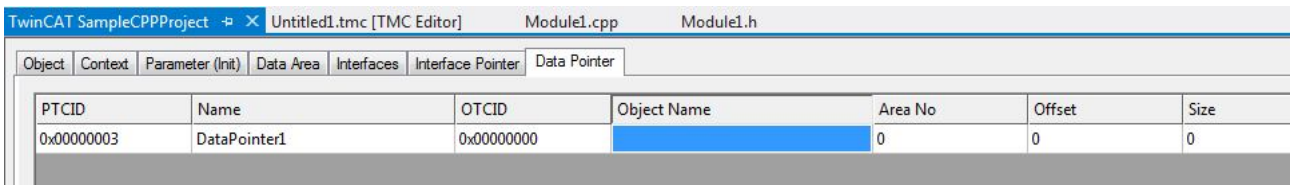


List of all interface pointers (as defined in TMC).

Special ParameterIDs (PTCID) are used to set the values automatically. These are configured using the TMC Editor's parameter dialog window as described in Parameters [▶ 111].

The OTCID column defines the pointer to the instance to be used.

11.4.7 Data Pointer



List of all data pointers (as defined in TMC).

Special ParameterIDs (PTCID) are used to set the values automatically. These are configured using the TMC Editor's parameter dialog window as described in Parameters [▶ 111].

The OTCID column defines the pointer to the instance to be used.

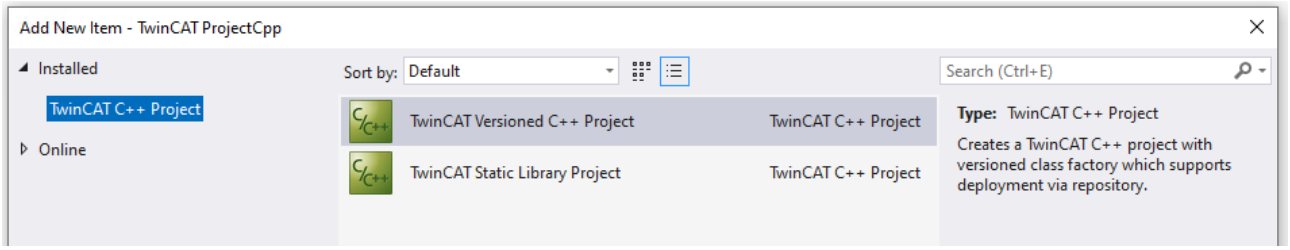
11.5 Customer-specific project templates

TwinCAT 3 is embedded in Visual Studio and thus also uses the project management provided. TwinCAT 3 C++ projects are "nested projects" in the TwinCAT project folder (TwinCAT Solution).

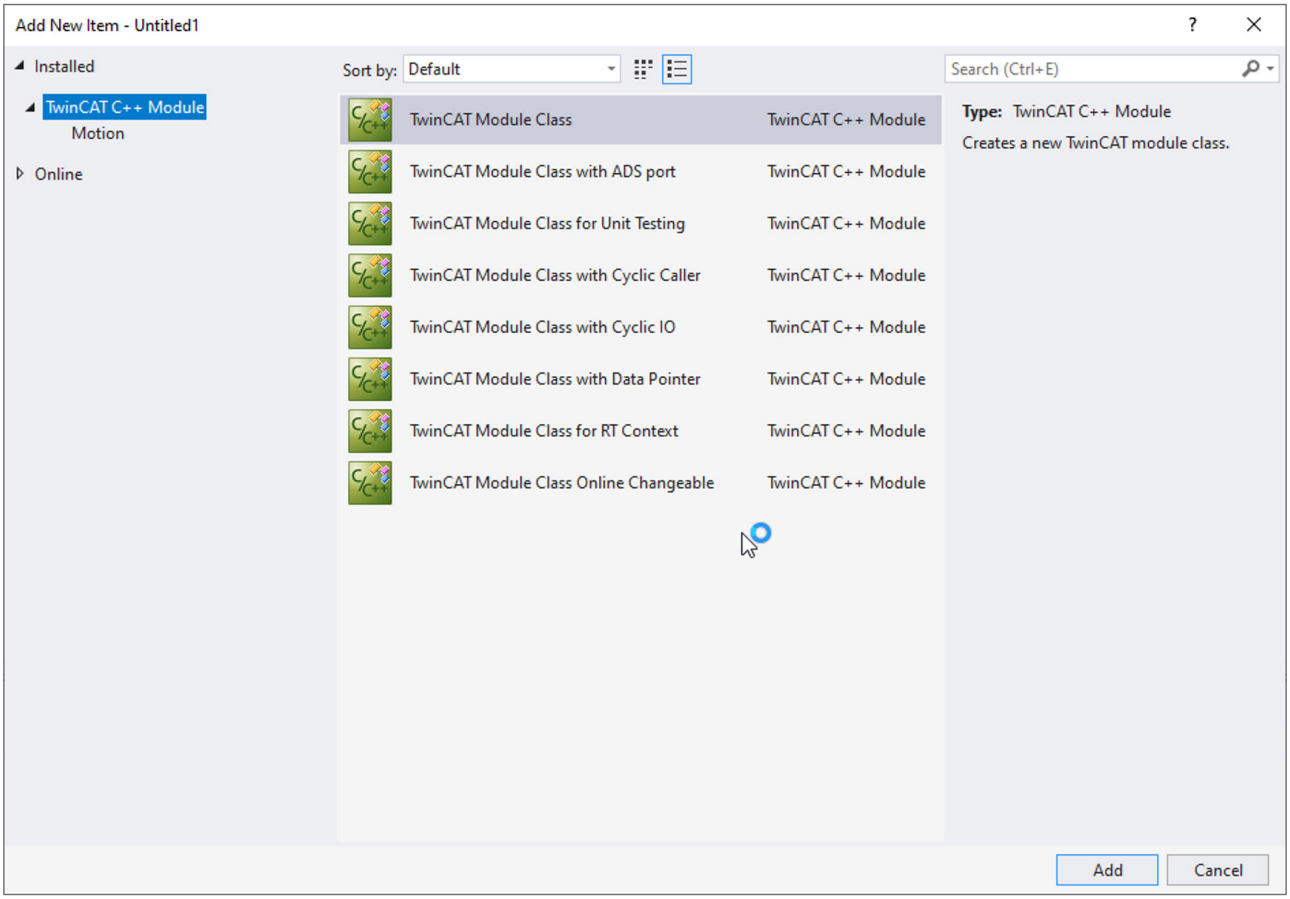
This section of the documentation describes how customers can realize their own project templates.

11.5.1 Overview

When a TwinCAT C/C++ project is created, the TwinCAT C++ Project Wizard is started first. The latter generates a framework for a TwinCAT Versioned C++ Project. The actual function is implemented in TwinCAT modules.



The TwinCAT Class Wizard is automatically started on creating a new project in order to add the first module. The different modules are generated by the same TwinCAT Class Wizard, but the specific design of the module is realized using templates.



11.5.2 Files involved

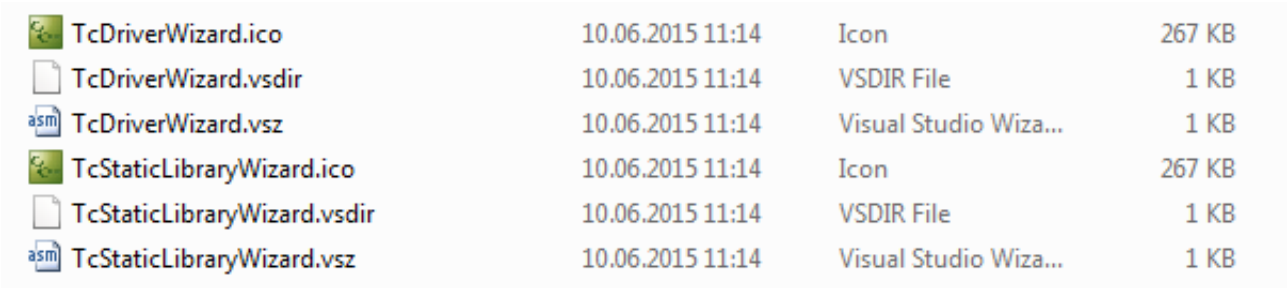
Virtually all relevant information is contained in the directory *C:\TwinCAT\3.1\Components\Base\CppTemplate*:



The TwinCAT C++ Project Wizard calls the TwinCAT Module Class Wizard if a Driver Project is to be created.

Directory: Driver and Class

The respective project types are defined in the Driver directory (for TwinCAT C++ Project Wizard) and Class directory (for the TwinCAT Module Class Wizard), each project type encompassing 3 files:



The *.vsdir* file provides information that is used when the respective assistant wizard is started. This is essentially a name, a brief description and a file name of the type *.vsz* containing details for this project type. The general description in MSDN can be found here https://msdn.microsoft.com/de-de/library/Aa291929%28v=VS.71%29.aspx.

The *.vsz* file referenced in the *.vsdir* file provides information that is needed by the wizard.
The most important information here is the wizard that is to be started and a list of parameters.

Both wizards have a *.xml* file as a parameter that describes the transformations of, for example, source files from the template to the specific project. These are located together with the templates for the source code, etc. in the *Templates* directory.
If a driver is to be created, the TwinCAT C++ Project Wizard starts the TwinCAT Module Class Wizard via the **TriggerAddModule** parameter.

The general description in the MSDN can be found here: https://msdn.microsoft.com/de-de/library/Aa291929%28v=VS.71%29.aspx.

The *.ico* file merely provides an icon.

Directory: Templates

Both the templates for the source code and the *.xml* file named in the *.vsz* file for the TwinCAT Module Class Wizard are located in corresponding subdirectories in the *Templates* directory.

This *.xml* file describes the procedure for getting from the template to a real project.

11.5.3 Transformations

Transformation description (XML file)

The configuration file describes (in XML) the transformation of the template files into the project folder. In the normal case these will be *.cpp / .h* and, if applicable, project files; however, all types of files can be handled.

The root node is a <ProjectFileGeneratorConfig> element. The useProjectInterface="true" attribute can be set directly at this node. It sets the processing procedure in the Visual Studio mode to generate projects (as opposed to TC-C++ modules).

Several <FileDescription> elements, each of which describes the transformation of a file, follow here. After these elements there is a possibility to define symbols that are available for the transformation in a <Symbols> element.

Transformation of the template files

A <FileDescription> element is structured as follows:

```
<FileDescription openFile="true">
<SourceFile>FileInTemplatesDirectory.cpp</SourceFile>
<TargetFile>[!output SYMBOLNAME].cpp</TargetFile>
<Filter>Source Files</Filter>
</FileDescription>
```

- The source file from the templates directory is specified as the <SourceFile>.
- The target file in the Project directory is specified as the <TargetFile>. A symbol is normally used by means of the [!output...] command.
- The attribute "copyOnly" can be used to specify whether the file should be transformed, i.e. whether the transformations described in the source file are executed. Otherwise the file is merely copied.
- The "openFile" attribute can be used to specify whether the file is to be opened after creation of the project in Visual Studio.
- Filter: A filter is created in the project.
  To do this the useProjectInterface="true" attribute must be set at the <ProjectFileGeneratorConfig>.

Transformation instructions

Commands that describe the transformations themselves are used in the template files.

The following commands are available:

- `[!output SYMBOLNAME]`
  This command replaces the command by the value of the symbol. A number of predefined symbols are available.
- `[!if SYMBOLNAME]`, `[!else]` and `[!endif]` describe a possibility to integrate corresponding text only in certain situations during the transformation.

Symbol name

Symbol names can be provided for the transformation instructions in 3 ways.
These are used by the commands described above in order to carry out replacements.

1. A series of predefined symbols directly in the configuration file:
   A list of <Symbols> is provided in the XML file. Symbols can be defined here: `<Symbols>`
   ```
   <Symbol>
   <Name>CustomerSymbol</Name>
   <Value>CustomerString</Value>
   </Symbol>
   </Symbols>
   ```
2. The generated target file names can be provided by adding the "symbolName" attribute:
   `<TargetFile symbolName="CustomerFileName">[!output SYMBOLNAME].txt</TargetFile>`
3. Important symbols are provided by the system itself.

Symbol Name (Projects)	Description
PROJECT_NAME	The project name from the Visual Studio dialog.
PROJECT_NAME_UPPERCASE	The project name in upper case letters.
WIN32_WINNT	0x0400
DRVID	Driver ID in the format: 0x03010000
PLATFORM_TOOLSET	Toolset version, e.g. v100
PLATFORM_TOOLSET_ELEMENT	Toolset version as an XML element, e.g. <PlatformToolset>v100</PlatformToolset>

Symbol Name (Projects)	Description
NEW_GUID_REGISTRY_FORMAT	Creates a new GUID in the format: {48583F97-206A-4C7C-9EF2-D5C8A31F7BDC}

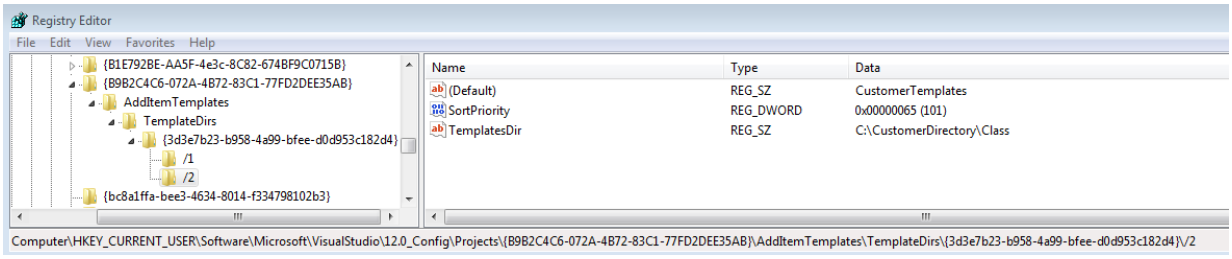
Symbol Name (Classes)	Description
PROJECT_NAME	The project name from the Visual Studio dialog.
HEADER_FILE_NAME	Entered by the user in the wizard dialog.
SOURCE_FILE_NAME	Entered by the user in the wizard dialog.
CLASS_NAME	Entered by the user in the wizard dialog.
CLASS_SHORT_NAME	Entered by the user in the wizard dialog.
CLASS_ID	A new GUID created by the wizard.
GROUP_NAME	C++
TMC_FILE_NAME	Used to identify the TMC file.
NEW_GUID_REGISTRY_FORMAT	Creates a new GUID in the format: {48583F97-206A-4C7C-9EF2-D5C8A31F7BDC}

11.5.4 Notes on handling

Template in customer-specific directory

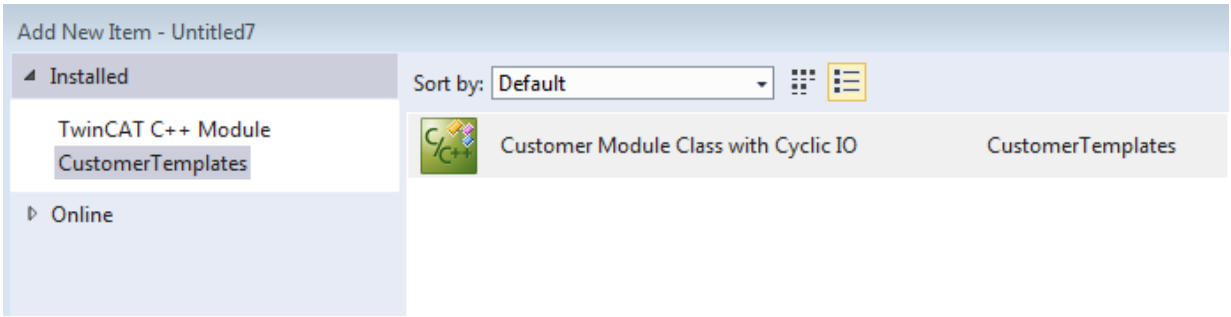
Templates can also be stored outside of the usual TwinCAT directory.

1. In the registry, expand the search path (in this case V12.0, i.e. for VS 2013) in which the node /2 is created:
   Registry Key:
   `HKEY_CURRENT_USER\Software\Microsoft\VisualStudio\12.0_Config\Projects\{B9B2C4C6-072A-4B72-83C1-77FD2DEE35AB}\AddItemTemplates\TemplateDirs\{3d3e7b23-b958-4a99-bfee-d0d953c182d4}\`



2. Increase the SortPriority.
3. Recommendation: in the directory, create a subdirectory called Class, which is entered in the registry, and a subdirectory called Templates in order to separate the *.vsz* / *.vsdir* / *.ico* files from the templates.
4. Adapt the paths within the files.

⇨ As a result, a dedicated order exists for the templates:



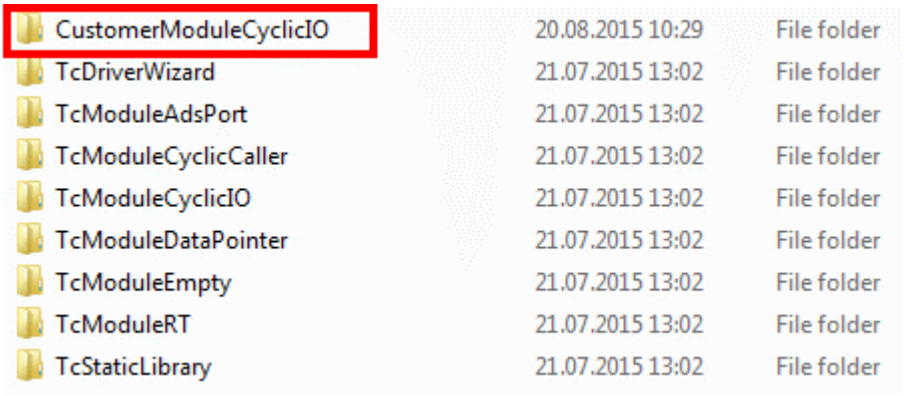
This directory or directory structure can, for example, now be versioned in the version management system and is also not affected by TwinCAT installations/updates.

Quick start

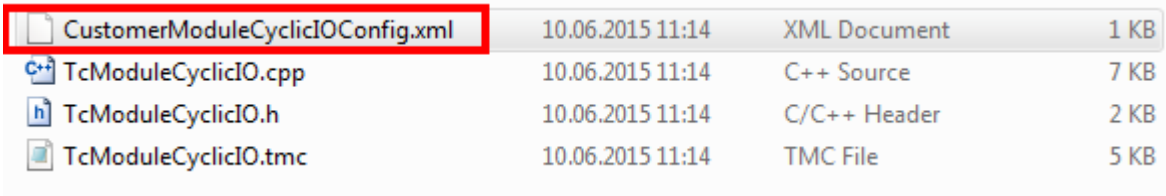
A general introduction to the wizard environment in MSDN is the entry point: https://msdn.microsoft.com/de-de/library/7k3w6w59%28v=VS.120%29.aspx.

This describes how a template is used for creating a customer-specific module with the TwinCAT C++ Module Wizard.

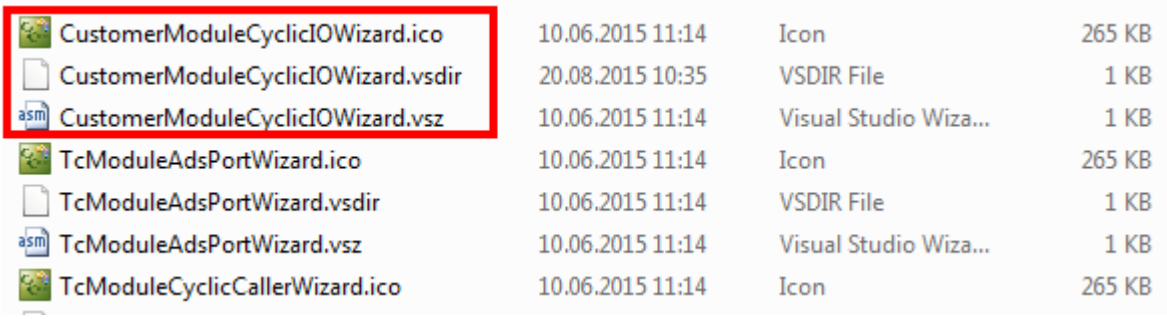
1. Take an existing module template as the copying template
   In **C:\TwinCAT\3.1\Components\Base\CppTemplate\Templates**



2. Rename the *.xml* file within the folder

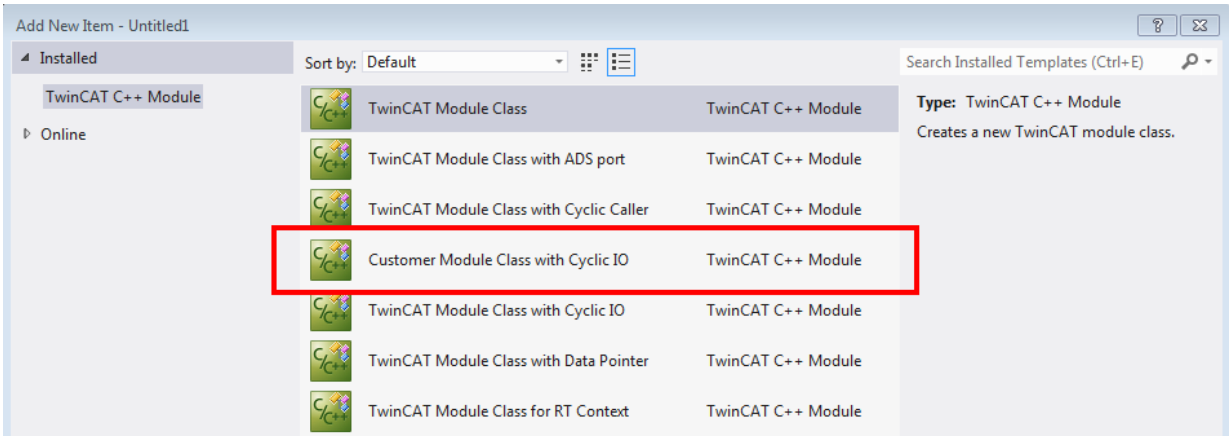


3. Copy the corresponding files *.ico* / *.vsdir* / *.vsz* also in the Class/



4. Now reference the copied *.vsz* file in the *.vsdir* file and adapt the description.
5. Enter the *.xml* file created in step 2 in the *.vsz* file.
6. You can now make changes to the source files in the *Template/CustomerModuleCyclicIO/* directory. The *.xml* takes care of replacements when creating a project from this template.

⇨ The TwinCAT Module Class Wizard now displays the new project for selection:



If the vsxproj, for example, are also to be provided in a changed form, it is recommended to adapt a copy of the TwinCAT C++ Project Wizard.

If necessary, the use of settings in *.props* files should also be considered so that settings can also be changed in existing projects created from a template – e.g. as a result of the *.props* files being updated by a version management system.

Alternative creation on the basis of an existing project

A viable way here is to create a finished project and transform it into a template afterwards.

1. Copy the cleaned project into the *Templates* folder.
2. Create a transformation description (XML file).
3. Prepare the source files and the project file by means of the replacements described.
4. Provide the *.ico* / *.vsdir* / *.vsz* files.

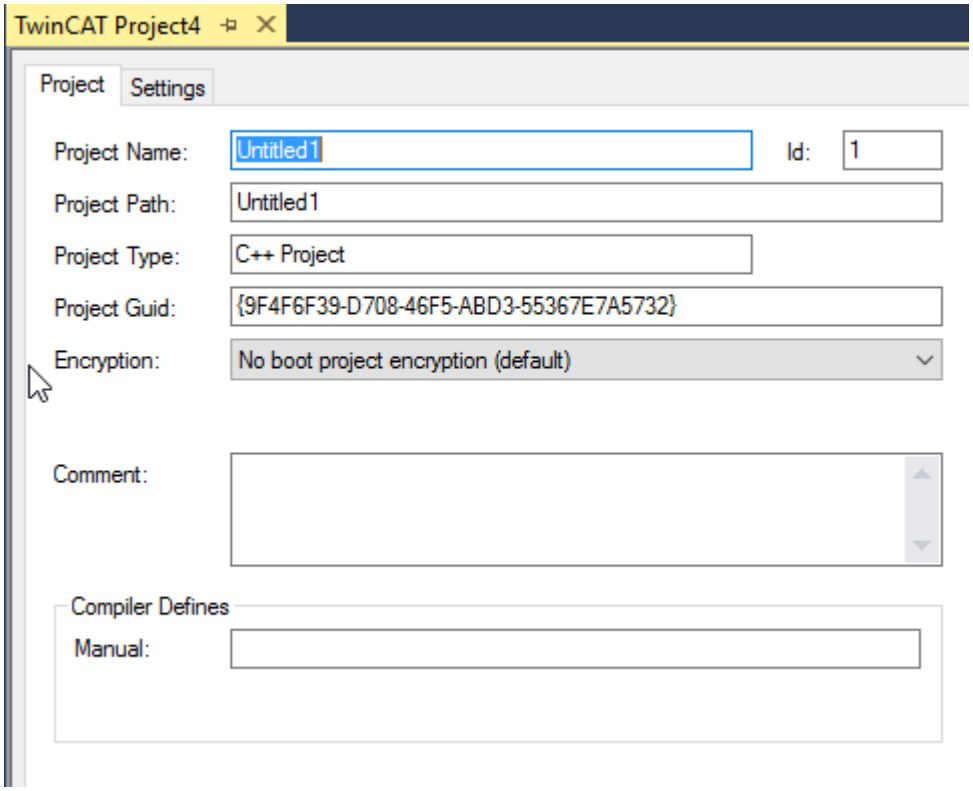
12 Programming Reference

TwinCAT offers a wide range of basic functions. They all can be very useful for TwinCAT C++ programmers and are documented here.

There is a wide range of C++ samples, which contain the valuable information on the handling of the modules and interfaces.

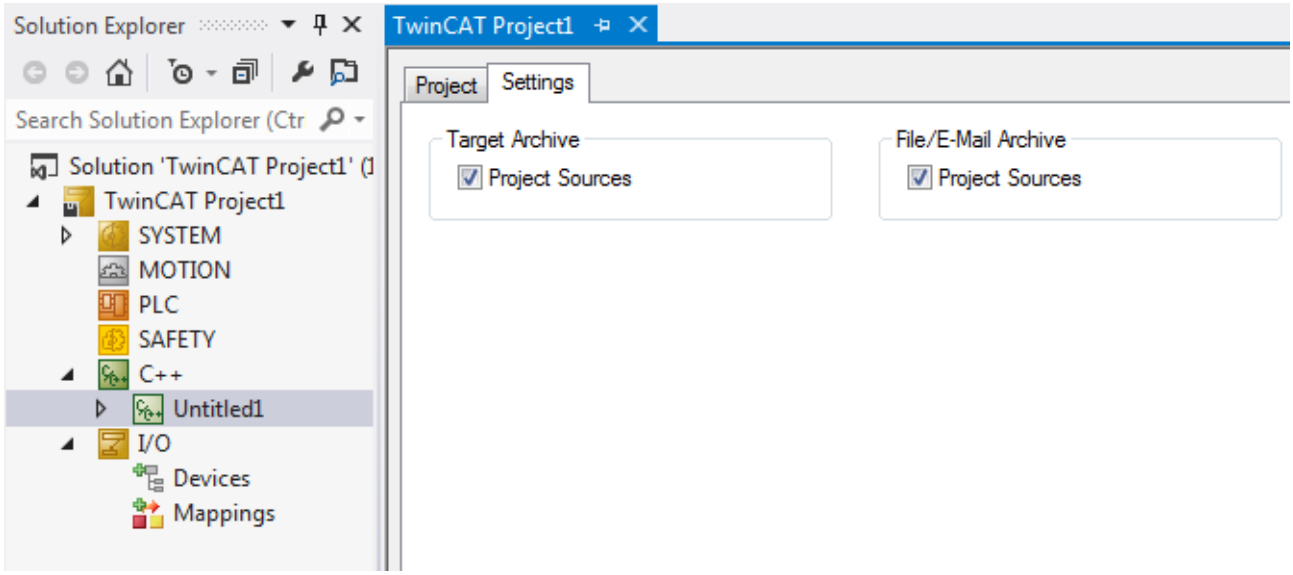
TwinCAT C++ project

A TwinCAT C++ project has several parameters that can be opened by double-clicking on the TwinCAT C++ project (project name here “Untitled1”).



Renaming is not possible here (see Renaming TwinCAT C++ projects [▶ 216])

The encryption of the binary module can be set here, a more detailed description of the requirements can be found in chapter Encrypting Modules [▶ 53].



The option whether the sources should be included can be set here for the two archive types, which are transferred to the target system or sent by email.

Accordingly, empty archives are created on deselection.

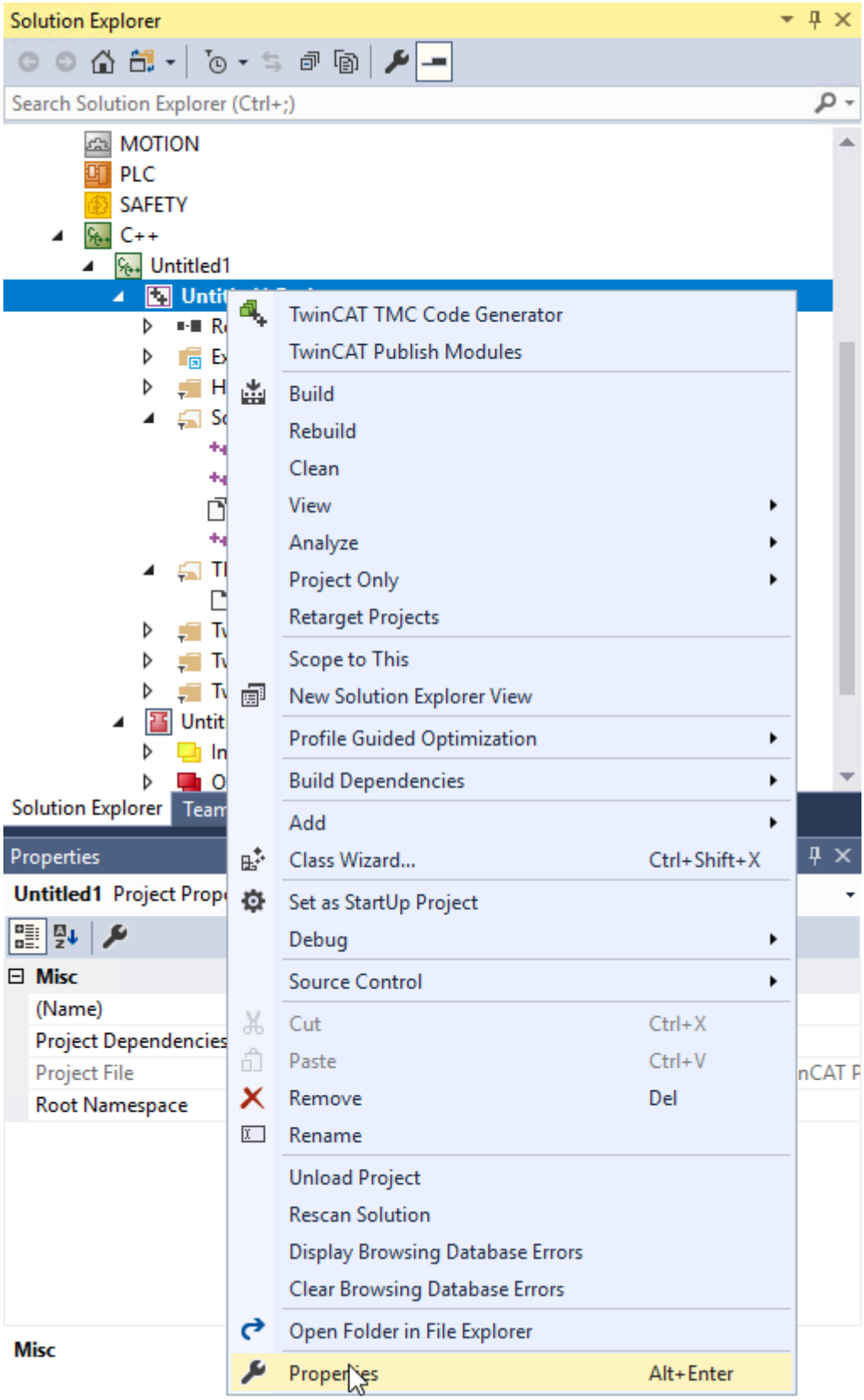
12.1 TwinCAT C++ Project properties

From TwinCAT 3.1 Build 4024.0

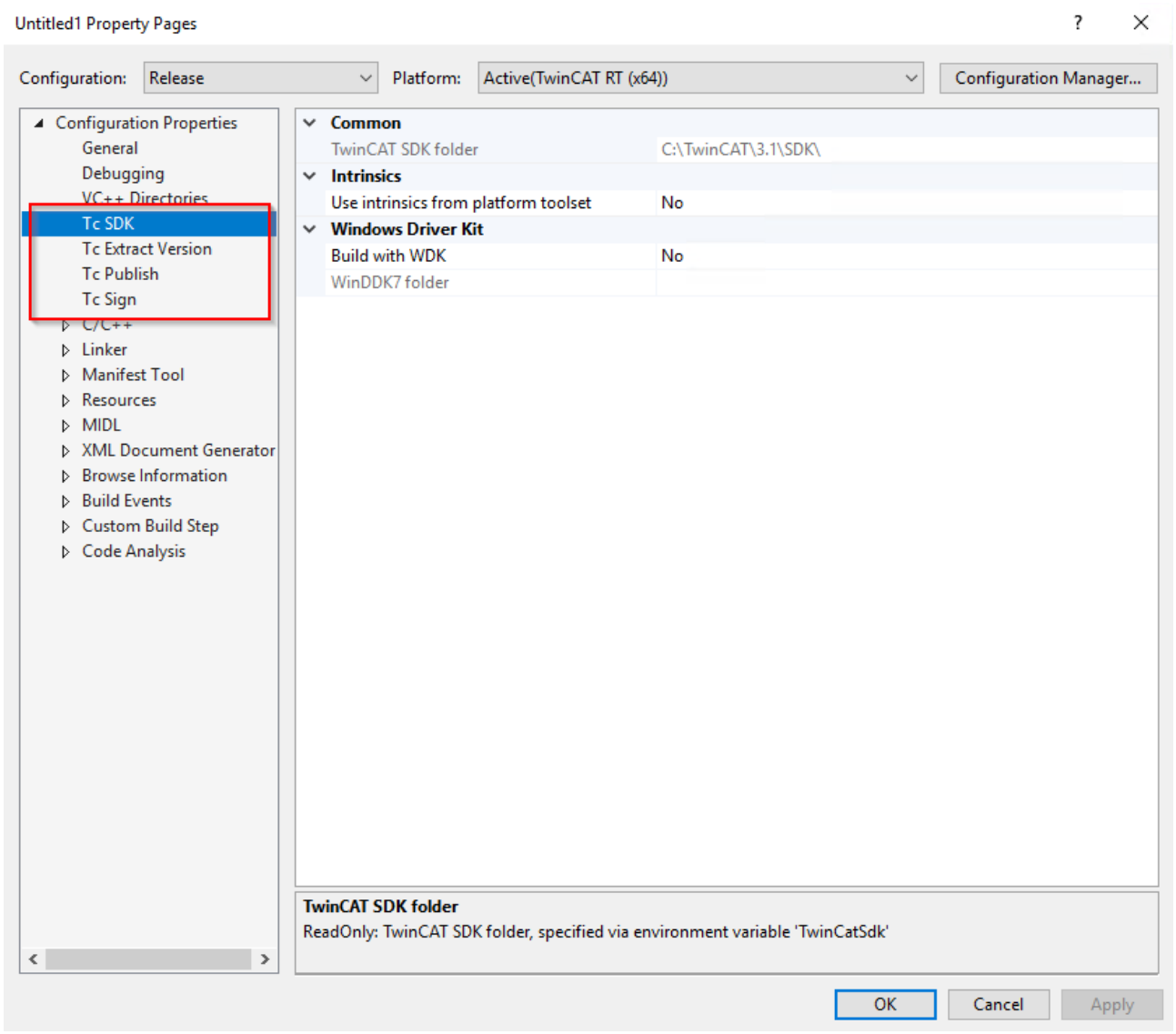
The functionality described here is available from TwinCAT 3.1. 4024.0.

Different settings can be made in the project properties for a TwinCAT C++ project.

The project properties are opened by right-clicking on the C++ project -> **Properties**.

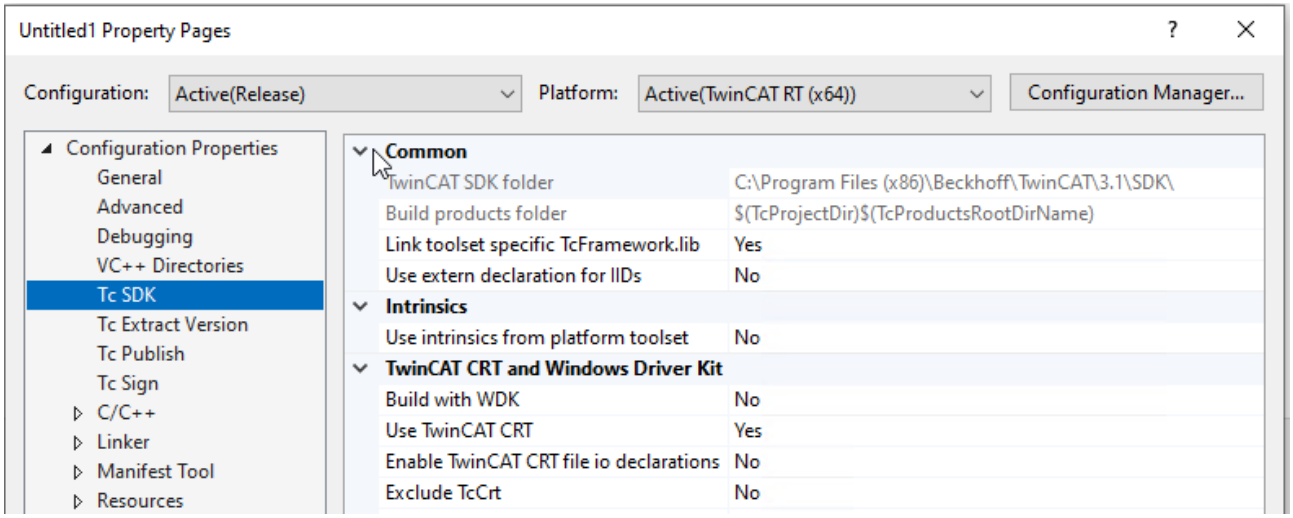


TwinCAT pages exist in addition to the Visual Studio C++ dialogs for the settings:



These are described on the subpages.

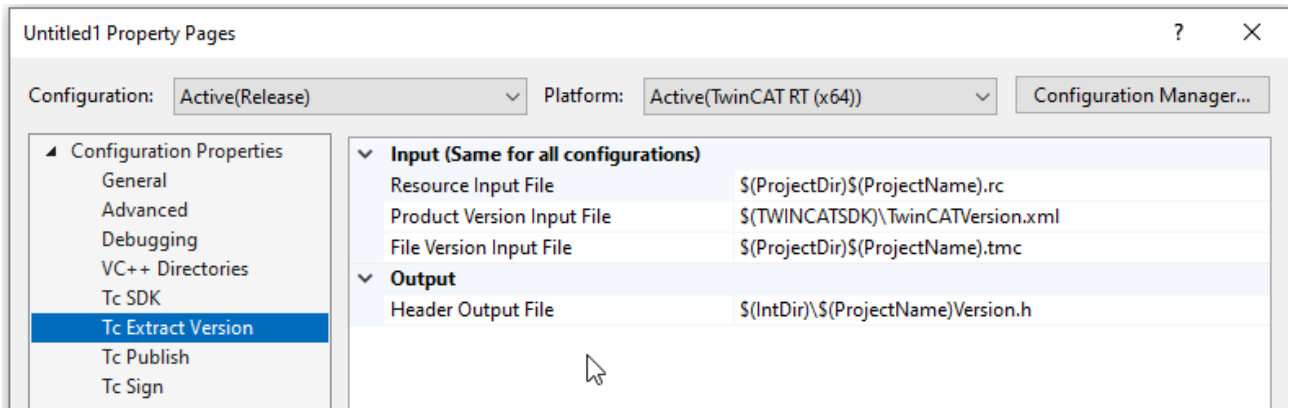
12.1.1 Tc SDK



Settings for the TwinCAT SDK

Category	Field	Description
Common	TwinCAT SDK folder	File folder that provides the TwinCAT SDK and shows the value of the environment variable TWINCATSDK.
Intrinsics	Use intrinsics from platform toolset	The intrinsics should be used.

12.1.2 Tc Extract Version

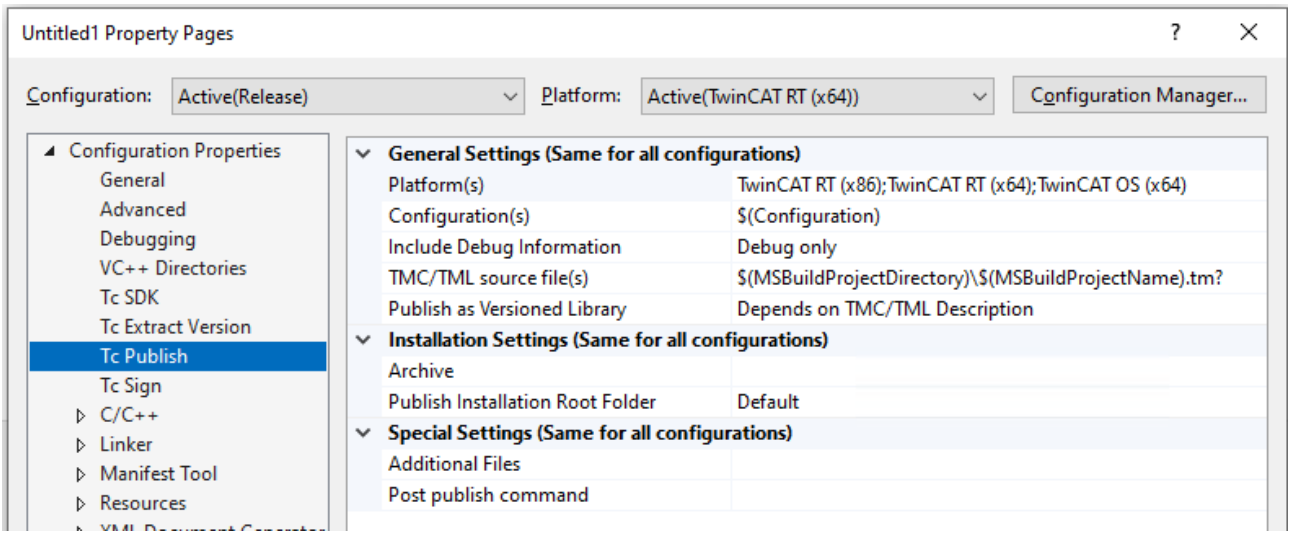


Version information from the project is provided in a header file and used for the build process.

If the *.rc* file contains version information, the header file is generated from it. With versioned C++ projects, the version information is read from the TMC file and the macros from the generated header file are used in the *.rc* file.

Category	Field	Description
Extract Version	Resource Input File	The .rc file to consider.
	Product Version Input File	TMC file containing the product version for versioned projects.
	File Version Input File	TMC file, which contains the file version for versioned projects.
	Header-Output File	Header File in which the information is provided.

12.1.3 Tc Publish



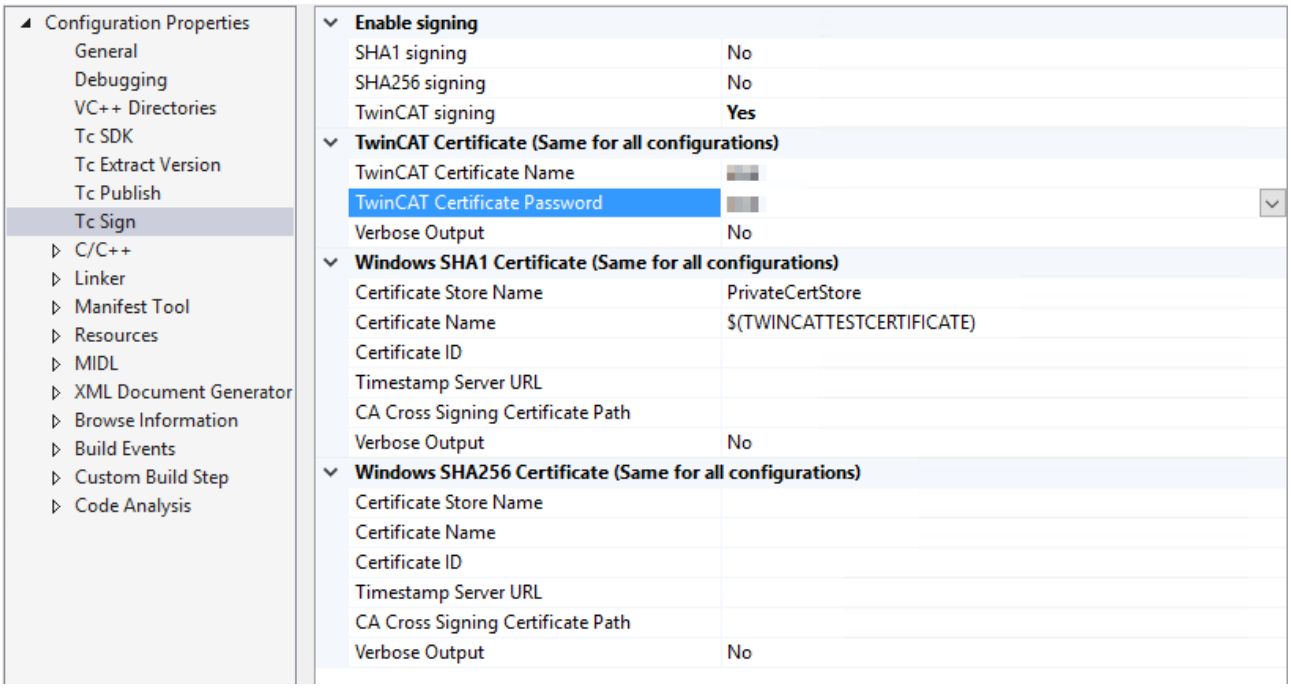
Information about publishing modules.

Category	Field	Description
General Settings	Platform(s)	Which platforms should be built in a Publish?
	Configuration(s)	Selection whether to build debug / release.
	Include Debug Information	For which configurations should the debug symbols (PDBs) be provided in the repository?
	TMC / TML source file(s)	TMC / TML files from the project that represent the starting point for the Publish process.
	Publish as Versioned Library	Should the publishing take place in the repository [▶ 49]?
Installation Settings	Archive	File path for an archive. Extensions .zip (for a ZIP archive) and .exe (for a self-extracting ZIP archive) are allowed. Both contain the content for a repository (versioned C++ projects) or CustomConfig/Modules (non-versioned C++ drivers) on another engineering system.
	Publish Installation Root Folder	No installation is performed on the local system with "None". The files are only available under TWINCATSDK/_products/TcPublish. An archive can be created to manually transfer these files to another system and install them there. If "Default" is selected, an installation into the repository (versioned C++ projects) or CustomConfig/Modules (non-versioned C++ drivers) is performed on the local system.
Special Settings	Additional Files	Adding additional files to the Publish process, which are stored in the "deploy" subdirectory during installation.
	Post publish command	Execute a command after the publish, e.g. to clean up.

12.1.4 Tc Sign

Can be omitted for Beckhoff RT Linux®

Signing can be omitted if only Beckhoff RT Linux® systems are used as the target system.



For Windows and TwinCAT/BSD® systems: TwinCAT modules must be signed [▶ 20], which can be configured here.

Category	Field	Description
Enable signing	SHA1 signing	Should an operating system signature, which is necessary for the operating system, be carried out?
	SHA256 signing	Should an operating system signature, which is necessary for the operating system, be carried out?
	TwinCAT signing	Should a TwinCAT user certificate be used for signing? This is necessary for the TwinCAT Loader [▶ 50].
TwinCAT Certificates These parameters are used for all configurations such as debug and release.	TwinCAT Certificate Name	Name of the certificate file (directory: C:\ProgramData\Beckhoff\TwinCAT\3.1\CustomConfig\Certificates). Alternatively, the environment variable TcSignTwinCatCertName can be set to the name of the certificate file.
	TwinCAT Certificate Password	Password that protects the TwinCAT user certificate (stored in plain text, leave blank if necessary). The TcSignTool [▶ 55] can be used to not store the password of the TwinCAT user certificate in the project, where it would also end up in versioning, for example.
	Verbose Output	Should extended information be output during the signature?
Windows Certificate (SHA1) Only included for compatibility reasons – please switch to TwinCAT Certificates immediately	Certificate Store Name	Name of the certificate store in the certificate manager of the operating system.
	Certificate Name	Name of the certificate in the certificate store.
	Certificate ID	ID of the certificate.
	Timestamp Server URL	URL of the timestamp server for use during the signature. This is provided by various CAs.
	CA Cross Signing Certificate Path	Path to Cross Signing certificate.
	Verbose Output	Should extended information be output during the signature?
Windows Certificate (SHA256) Only included for compatibility reasons – please switch to TwinCAT Certificates immediately	Certificate Store Name	Name of the certificate store in the certificate manager of the operating system.
	Certificate Name	Name of the certificate in the certificate store.
	Certificate ID	ID of the certificate.
	Timestamp Server URL	URL of the timestamp server for use during signature, provided by the CA.
	CA Cross Signing Certificate Path	Path to Cross Signing certificate.
	Verbose Output	Should extended information be output during the signature?

12.2 File Description

During the development of TwinCAT C++ modules, files in the file system can be handled directly. This is of interest, either to understand how the system works or for specific use cases such as manual file transfer, etc.

Here is a list of files related to C++ modules.

File	Description	Further information
Engineering / XAE		
*.sln	Visual Studio Solution file, hosts TwinCAT and non-TwinCAT projects	

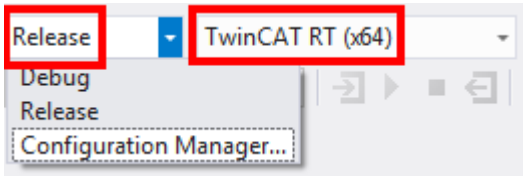
File	Description	Further information
*.tsproj	TwinCAT project, collection of all nested TwinCAT projects, such as TwinCAT C++ or TwinCAT PLC project	
_Config/	Folder contains further configuration files *(*.xti)* that belong to the TwinCAT project.	See menu Tools| Options| TwinCAT| XAE-Environment| File Settings
_Deployment/	Folder for compiled TwinCAT C++ drivers	
*.tmc	TwinCAT Module Class file (XML-based)	See TwinCAT Module Class Editor (TMC) [▶ 88]
*.rc	Resource file	See Setting version/ vendor information [▶ 214]
.vcxproj.	Visual Studio C++ project files	
*ClassFactory.cpp/.h	Class Factory for this TwinCAT driver	
*Ctrl.cpp/.h	Upload and remove drivers for TwinCAT UM platform	
*Driver.cpp/.h	Upload and remove drivers for TwinCAT RT platform	
*Interfaces.cpp/.h	Declaration of the TwinCAT COM interface classes	
*W32.cpp./.def/.idl		
*.cpp/.h	One C++/Header file per TwinCAT module in the driver. Insert user code here.	
Resource.h	Required by *.rc* file	
TcPch.cpp/.h	Used for creating precompiled headers	
%TC_INSTALLPATH% \Repository\<Vendor>\<PrjName>\<Version>\<Platform>*.tmx or *.tme	Compiled TwinCAT C++ project TwinCAT 3.1.4024.x: *C:\TwinCAT\3.1\Repository\C++ Module Vendor\Untitled1\0.0.0.1\TwinCAT RT *\Unititled1.tmx* From TwinCAT 3.1.4026.x: C:\ProgramData\Beckhoff\TwinCAT\3.1\Repository\C++ Module Vendor\Untitled1\0.0.0.1\TwinCAT RT *\Unititled1.tmx | tme	See Versioned C++ Projects [▶ 49]
%TC_INSTALLPATH% \CustomConfig\Modules*	Published TwinCAT C++ project normally TwinCAT 3.1.4024.x: *C:\TwinCAT\3.1\CustomConfig\Modules** From TwinCAT 3.1.4026.x: C:\ProgramFiles (x86)\Beckhoff \TwinCAT\3.1\Config\Modules*	See Export modules
Runtime / XAR		
%TC_BOOTPRJPATH% \CurrentConfig*	Current configuration setup *Windows:* TwinCAT 3.1.4024.x: *C:\TwinCAT\3.1\Boot* From TwinCAT 3.1.4026.x: Windows: C:\ProgramData\Beckhoff \TwinCAT\3.1\Boot *TwinCAT/BSD: /usr/local/etc/TwinCAT/3.1/Boot*	

File	Description	Further information
	Beckhoff RT Linux: /etc/TwinCAT/3.1/Boot	
%TC_DRIVERAUTOINSTALLPATH% *.sys/pdb	Compiled, platform-specific driver that is loaded via the operating system. *Windows:* TwinCAT 3.1.4024.x: *C:\TwinCAT\3.1\Driver\AutoInstall (loaded by the system)* From TwinCAT 3.1.4026.x *<not available> Please migrate to TMX based C++ projects*	
%TC_INSTALLPATH% \Boot\Repository\<Vendor>\<PrjName>\<Version>*.tmx	Compiled platform-specific driver *Windows:* TwinCAT 3.1.4024.x: *C:\TwinCAT\3.1\Boot\Repository\C++ Module Vendor\Untitled1\0.0.0.1\Untitled1.tmx* From TwinCAT 3.1.4026.x: C:\ProgramData\Beckhoff \TwinCAT\3.1\Boot\Repository\C++ Module Vendor\Untitled1\0.0.0.1\Untitled1.tmx *TwinCAT/BSD: /usr/local/etc/TwinCAT/3.1/Boot\/Repository\C++ Module Vendor\Untitled1/0.0.0.1/Untitled1.tmx* *Beckhoff RT Linux : /etc/TwinCAT/3.1/Boot\/Repository\C++ Module Vendor\Untitled1/0.0.0.1/Untitled1.tme*	
%TC_BOOTPRJPATH% \TM\OBJECTID.tmi	TwinCAT Module Instance file Describes variables of the driver File name is *ObjectID.tmi* Windows: TwinCAT 3.1.4024.x: *C:\TwinCAT\3.1\Boot\TMI\OTCID.tmi* From TwinCAT 3.1.4026.x: Windows: C:\ProgramData\Beckhoff \TwinCAT\3.1\Boot\TMI\OTCID.tmi *TwinCAT/BSD: /usr/local/etc/TwinCAT/3.1/Boot/TMI/OTCID.tmi* *Beckhoff RT Linux: /etc/TwinCAT/3.1/Boot/TMI/OTCID.tmi*	
Temporary files		
*.sdf	IntelliSense Database	
*.suo / *.v12.suo	User-specific and Visual Studio-specific files	
*.tsproj.bak	Automatically generated backup file from *tsproj*	
ipch/	Intermediate directory created for precompiled headers	

Compilation procedure

The procedure that initiates a Build or Rebuild on a TwinCAT C++ project in the TwinCAT Engineering XAE is described here. This is to be taken into account, for example, if company-specific environments and building processes are to be integrated.

The configurations that are built in the case of a Build or Rebuild depend on the current selection in Visual Studio:



The correct target architecture (in this case TwinCAT RT (x64)) is set appropriately by choosing the target system.

The **Configuration Manager** allows the dedicated setting of the build configuration.

When selecting **Build** or **Rebuild** (and thus also **Activate Configuration**), the following steps are performed:

1. The sources are located in the respective project directory.
2. The compilations are generated in an architecture-specific manner below the project directory _products\TwinCAT RT (x64)\Release\<ProjectName>
3. After the link process, the .tmx/.pdb file is then stored in the project directory.
4. Pressing the Activate Configuration button transfers .tmx or tme with .pdb from the engineering repository to the target system (this may be a local copy).

When Publish is selected on the project, it builds for the selected platforms and stores the result in the engineering repository: C:\ProgramData\Beckhoff\TwinCAT\3.1\Repository\<vendor>\<ProjectName>\<version>\TwinCAT RT (x64)\Release\, whereby the TMC file is also updated.

12.3 Online Change

From TwinCAT 3.1 Build 4024.0

The functionality described here is available from TwinCAT 3.1. 4024.0.

TwinCAT 3.1 supports the exchange of C++ modules at runtime, i.e. without interrupting the real-time program.

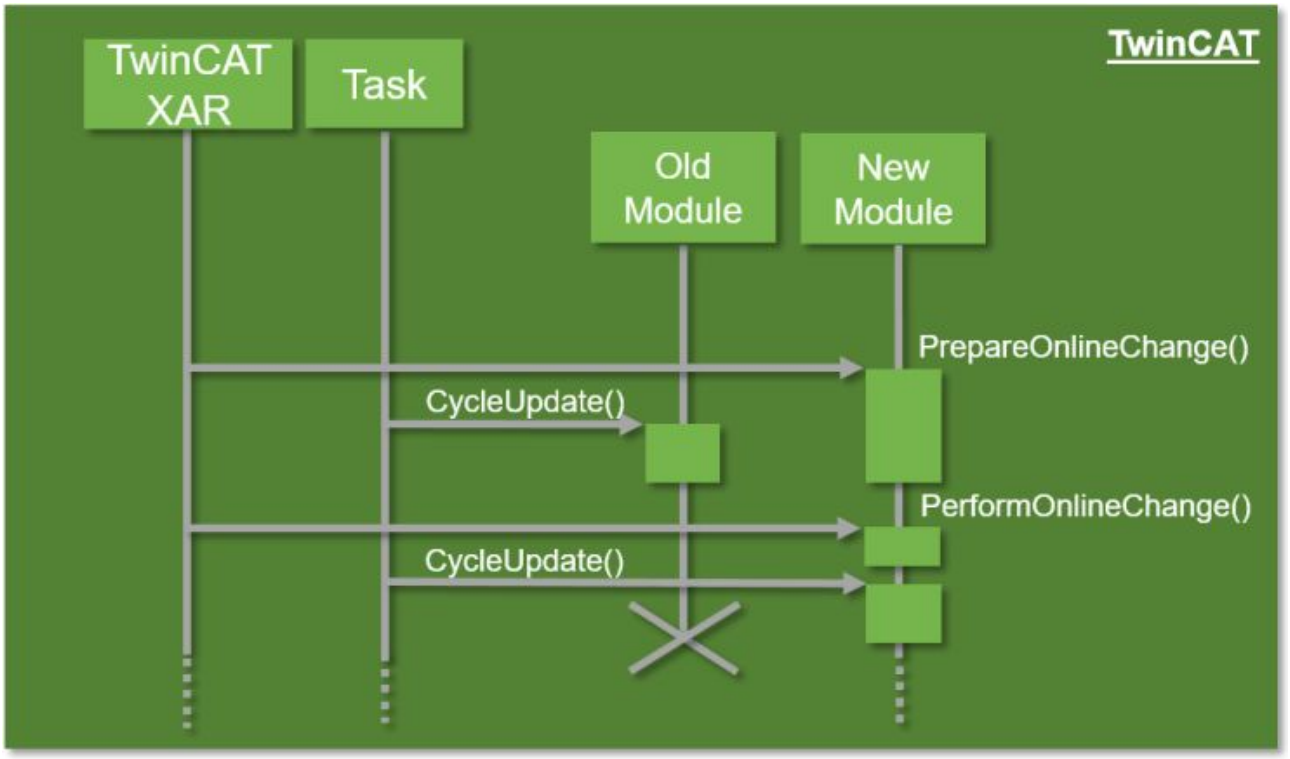
For this purpose, different versions [▶ 49] of a TwinCAT Executable (TMX) are stored on the target system,.

For all module instances from a TMX, a switchover between the versions can be initiated by the engineering.

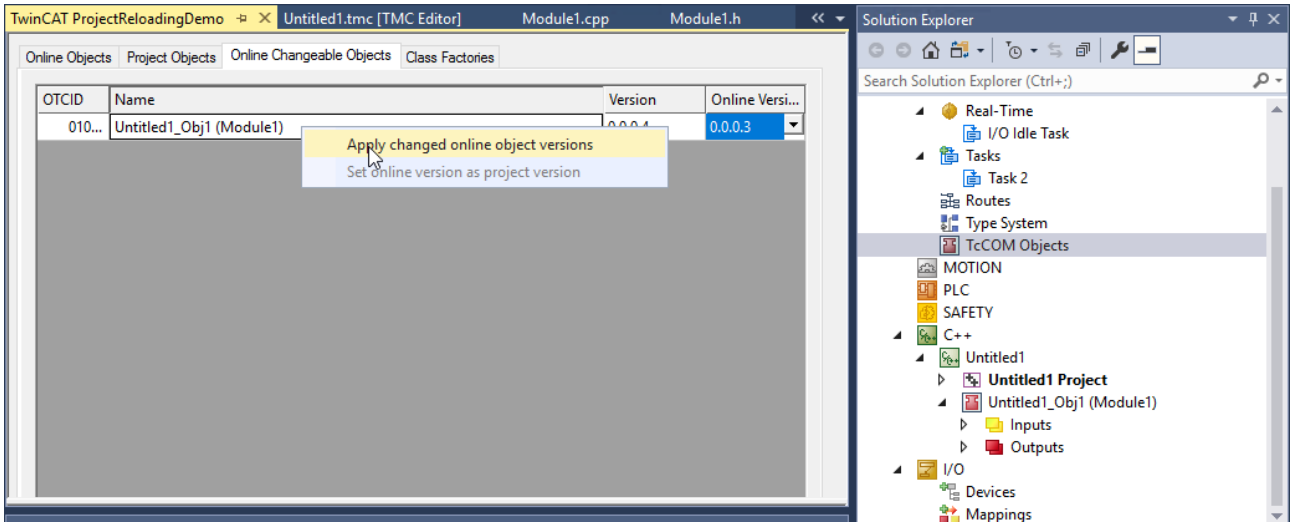
The procedure is roughly sketched:

✓ Online Change capable module in TMX

1. TwinCAT instantiates the new module. The old module is still called cyclically by the task.
2. TwinCAT calls ITcOnlineChange::PrepareOnlineChange() of the new module.
   This call can access the old module and accept data that does not change due to the cyclic calls of the module - for example parameter values.
3. TwinCAT calls ITcOnlineChange::PerformOnlineChange() of the new module.
   This call can access the old module and take over data which have changed cyclically before. This call is executed if no cyclic call is made by a task. The old module is **not** called again by the task, but the new module is called. The PerformOnlineChange() method should use as little computing time as possible so that this switchover can take place from one task cycle to the other.
4. After completion, the task will call the new module cyclically.



The Online Change can be carried out through this dialog in engineering.



When dealing with the Online Change, there are therefore some aspects to consider:

- The project must provide a revision control.
- The DataAreas for these modules are kept outside the TcCOM module and made available to the modules. This means that they do not need to consider the data or the mapping of the symbols in the DataAreas.
- The DataAreas of the module must not change.
- References to internal data structures must not be passed on. Access must always take place via interfaces that are retrieved via the TcQueryInterface, since these references are updated during an Online Change.

After a restart, TwinCAT will start the driver in the initial version of the modules.

12.4 Limitations for TwinCAT C++ modules

TwinCAT 3 C++ modules [▶ 35] are executed in Windows kernel mode. Developers must therefore be aware of some limitations:

- Win32 API [▶ 150] is not available in kernel mode
- Windows kernel mode API must not be used directly.
  TwinCAT SDK provides functions, which are supported.
- User mode libraries (DLL) cannot be used. (see Third Party Libraries [▶ 228])
- The storage capacity for dynamic allocation in a real-time context is limited by the router memory (this can be configured during engineering), see Memory allocation [▶ 150].
- A subset of the C++ runtime library functions (CRT) is supported
- C++ exceptions are not supported.
- Runtime Type Information (RTTI [▶ 150]) is not supported.
- Subset of STL is supported (see STL / Containers [▶ 208])
- Support for functions from math.h through TwinCAT implementation (see Mathematical Functions [▶ 206])

TwinCAT functions as replacement for Win32 API functions

The original Win32 API is not available in Windows kernel mode. For this reason a list of the common functions of the Win32 API and their equivalents for TwinCAT is provided here:

Win32API	TwinCAT functionality
WinSock	TF6311 TCP/UDP real-time
Message boxes	Tracing [▶ 209]
File I/O	See Interface ITcFileAccess [▶ 164], Interface ITcFileAccessAsync [▶ 171] and Sample19: Synchronous File Access [▶ 289], Sample20: FileIO-Write [▶ 290], Sample20a: FileIO-Cyclic Read / Write [▶ 290]
Synchronization	See Sample11a: Module communication: Method call C++ module to C++ module [▶ 284]
Visual C CRT	See RtlR0.h

RTTI dynamic_cast function in TwinCAT

TwinCAT has no support for dynamic_cast<>.

Instead, it may be possible to use the TCOM strategy. Define an ICustom interface, which is derived from ITcUnknown and contains the methods, which are called from a derived class. The base class CMyBase is derived from ITcUnknown and implements this interface. The class CMyDerived is derived from CMyBase and from ICustom. It overwrites the TcQueryInterface method, which can then be used instead of dynamic cast.

TcQueryInterface can also be used to display the IsType() function through evaluation of the return value.

See Interface ITcUnknown [▶ 187].

12.5 Memory allocation

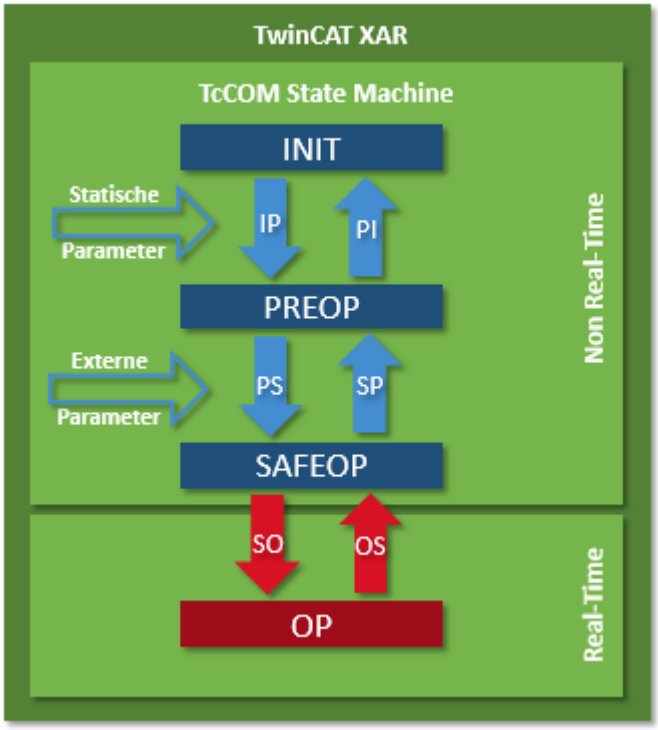
Generally we recommend reserving memory with the aid of member variables of the module class. This is done automatically for data areas defined in the TMC editor.
It is also possible to allocate and release memory dynamically.

- Operator new / delete
- TcMemAllocate / TcMemFree

This memory allocation can be used in the transitions [▶ 44] or in the OP state of the state machine.

If the memory allocation is made in a non-real-time context, the memory is allocated in the non-paged pool of the operating system (blue in the diagram). In the TwinCAT real-time context, the memory is allocated in the router memory (red in the diagram).

The memory can also be released in the transitions or the OP state; we recommend to always release the memory in the "symmetric" transition, e.g. allocation in PS, release in SP.



When using static variables, some special features [▶ 152] have to be considered.

In addition, some notes on the implementation of transitions can be provided:

- Transitions IP, PS and SP, PI are executed in a non-real-time context (e.g. Windows Kernel Mode).
- Transitions SO, OS, however, are executed in the TwinCAT RT Task Context (more precisely: TcCOM Server Task).
- Transitions must be implemented according to the "all-or-nothing" pattern. This means that all steps that are carried out successfully must be undone if a subsequent step fails.
- If memory is allocated in the OP state, it must be released in the OS transition at the latest.
- The OS, SP, PI transitions should be implemented in such a way that they are always successful. No error codes should be returned.
- Transitions should be implemented symmetrically, i.e. OS and SO should be inverse, ditto for PS <-> SP and IP <->PI.
- Transition SO should log in the module to the task as the last step (if an ITcCaller is present)
- Transition OS should log off the module from the task as a first step (if an ITcCaller is present)
- Semaphores
  - Can be created and deleted in a non-real-time context
  - Use (pend / post) is only permitted in a real-time context
- Several modules can wait for each other in the states. The InitSeq (default "PSO") is used to specify the states in which the module instances should wait for the overall system (and thus all other modules) to be switched on.

12.6 Static variables

Memory allocation between Windows and real-time context in conjunction with static variables

If global instances are used, be aware of the following:

- A maximum of 128 global instances may exist in total.
  It should be noted that global instances can also be used by TwinCAT's own modules.
- If the number is exceeded, the destructors are no longer called.
- From TwinCAT 3.1 4026, a message is displayed in the "Output" of the engineering if the limit is exceeded:
  Too many static objects detected on shutdown of class factory <ProjectName>.tmx. Max Number of static objects exceeded by N item(s). This may result in a memory leak.
- The program code can query the currently used or available global instances:
  ```
  #include "CrtInit.h"
  int overflow = cppGetNumberHandlerOverflow();
  int free = TcCrtGetFreeNumberAtexitHandler();
  ```
- Memory allocated in the real-time context must be released in the OS transition, for example, so that this does not happen via the destructor.
- When the constructor is executed, TwinCAT system resources cannot yet be accessed, because the connection via the Classfactory to the TcCOM object server has not yet been established. Below you will see how to deal with this.

This code includes three examples of global instances:

```
class MyClassA
{
public:
    MyClassA() {}
    ~MyClassA() {}
private:
    int v;
}

MyClassA A;

class MyClassB
{
    static MyClassA Value;
};

MyClassA& GetInstance()
{
    static MyClassA a;
    return a;
}
```

Singletons for the initialization of global instances

In order to delay the time, global or so-called singleton instances can also be created as local static objects. It is very important when using global class instances that the destructors are called late and at this point there is no longer a connection to the TwinCAT system, so that TwinCAT resources must be released beforehand.

Here is a sample code showing a singleton in TwinCAT:

```
#include "TcInterfaces.h"
class VersionProvider : public ITcVersionProvider
{
public:
       DECLARE_IUNKNOWN_NODELETE()
       VersionProvider()
       {
              m_Version = { 1, 2, 3, 0};
       }
       virtual HRESULT TCOMAPI GetVersion(T_Version& version)
       {
```

```
                version = m_Version;
                return S_OK;
        };
private:
        T_Version m_Version;
};
BEGIN_INTERFACE_MAP(VersionProvider)
        INTERFACE_ENTRY(IID_ITcVersionProvider, ITcVersionProvider)
END_INTERFACE_MAP()
class MySingleton
{
public:
        static MySingleton& Instance(ITcVersionProvider* ip = NULL, bool bRelease=false)
        {
                static MySingleton g_instance(ip);
                if (bRelease) g_instance.Release();
                return g_instance;
        }
        T_Version GetVersion()
        {
              T_Version v = {0, 0, 0, 0};
                if (m_spTcInst != NULL)
                {
                        m_spTcInst->GetVersion(v);
                }
                return v;
        }
private:
        MySingleton(ITcVersionProvider* ip = NULL)
                : m_spTcInst(ip)
        {
        }
        void Release()
        {
              m_spTcInst = NULL;
        }
        ITcVersionProviderPtr m_spTcInst;
};
```

Here the usage is shown:

```
// actual instance which is an example for a TwinCAT resource
VersionProvider vp;
// Initialize singleton
MySingleton::Instance(&vp);
// use singleton to access information provided by TwinCAT resource
T_Version v1 = MySingleton::Instance().GetVersion();
Assert::IsTrue( v1.major == 1 && v1.minor == 2 && v1.build == 3 &&  v1.revision == 0);
// Deinitialize singleton and release reference to TwinCAT resource
MySingleton::Instance(NULL, true);
// use singleton after deinitialization which has to implement some default behaviour for this case
T_Version v2 = MySingleton::Instance().GetVersion();
Assert::IsTrue( v2.major == 0 && v2.minor == 0 && v2.build == 0 &&  v2.revision == 0);
```

12.7 Multi-task data access synchronization

When one set of data is accessed by multiple tasks, the tasks may access the same data simultaneously, depending on the task/real-time configuration. If the data is written by at least one of the tasks, the data may have an inconsistent state during or after a change. To prevent this, all concurrent accesses must be synchronized so that only one task can access the shared data at a time.

If the same data is accessed by several tasks and the data is written for at least one of these accesses, all read and write accesses must be synchronized. This applies regardless of whether the tasks are run on a single or multiple CPU cores.

⚠ WARNING

Inconsistencies and other risks due to unsecured data access

If concurrent accesses are not synchronized, there is a risk that the data records will be inconsistent or invalid. Depending on how the data is used in the further course of the program, this can result in incorrect program behavior, undesired axis movement, or even a sudden program standstill. Damage to equipment and workpieces may occur, or people's health and lives may be endangered, depending on the controlled system.

General synchronization options between the TcCOM modules are already described in chapter Module-to-module communication [▶ 46]. The following mechanisms were described:

- Data exchange via the process image (IO mapping). During this process, the cycle transition, and thus TwinCAT, ensures data synchronicity between the sources and targets.
- Method calls via interfaces in which CriticalSections can be used.
- ADS, which transfers data in the form of a transport medium and thus ensures synchronicity of the data.
- Common data area to be protected by a CriticalSection.

Ideally, however, you should try to avoid the need for synchronization. Otherwise, the simplest way is usually the exchange via the process image, which copies the data between the cycles from the output process images to the input process images and thus ensures a consistent state.
If this is not sufficient and data has to be accessed from different contexts, one of the following options can be used.

It is important to differentiate between the task contexts that are to access the data. In the TwinCAT Runtime, a distinction is made between the Windows kernel mode thread contexts (Windows context for short) and the TwinCAT real-time task contexts (RT context). During initialization of TwinCAT modules, the IP, PS, SP and PI transitions are executed in the Windows context. The SO and OS transitions are executed in the RT context.

The SDK offers appropriate synchronization options for such scenarios. Nevertheless, the TcCOM modules should be designed such that they are independent of each other, at least in the Windows context, and thus do not require synchronization. The transitions in the RT context can use CriticalSections if required.

For the CriticalSection and semaphores it applies that in case of a lock the task waits for the release. In the meantime the core is released for other tasks. With CriticalSections the active task inherits the priority of the waiting task ("Priority Inheritance").

12.7.1 CriticalSections

Instances of CriticalSections are created through TwinCAT real-time. This instance can be initialized in the Windows context as well as in the RT context. The CriticalSection instance can only be used in the RT context.

TwinCAT real-time uses "priority inheritance" to prevent a task with lower priority from indirectly blocking a task with high priority.

CCriticalSectionInstance

The CCriticalSectionInstance class provides the interface to handle Critical Sections and has the required memory. To create a Critical Section the class requires the object ID of the TwinCAT real-time instance, which is available via OID_TCRTIME_CTRL, as well as a reference to the TwinCAT object server via a pointer to ITComObjectServer.

Methods:

- CCriticalSectionInstance(OTCID oid=0, ITComObjectServer* ipSrv=NULL);
  The default constructor that initializes the object ID of the Critical Section Provider. If the pointer is given to the object server, the Critical Section is also initialized.
- ~CCriticalSectionInstance();
  The destructor deletes the Cricital Section instance.
- void SetOidCriticalSection(OTCID oid);
  Sets the Critical Section Provider, which is given by TwinCAT real-time and whose object ID is available via OID_TCRTIME_CTRL.
- bool HasOidCriticalSection();
  Returns TRUE if the object ID is set to a value other than 0, otherwise FALSE. The object ID is not checked to verify whether it belongs to a Critical Section Provider.
- bool IsInitializedCriticalSection();
  Returns TRUE if the Critical Section was successfully initialized.

- HRESULT CreateCriticalSection(ITComObjectServer* ipSrv);
  If the Critical Section is initialized, it is deleted and a new Critical Section is initialized. Returns S_OK if successful. Error cases are indicated by the return of error codes:

ADS_E_INVALIDPARM	Invalid Critical Section Provider
ADS_E_NOINTERFACE	The object ID is set to a reference that is not a Critical Section Provider, i.e. ITcRTime is not implemented.
E_FAIL	Internal error from the Critical Section Provider.

- HRESULT CreateCriticalSection(OTCID oid, ITComObjectServer* ipSrv);
  Initializes the Critical Section. DeleteCriticalSection() must be called if this method is used again. Return values as in CreateCriticalSection(ITComObjectServer* ipSrv);
- HRESULT DeleteCriticalSection(); Critical Section is deleted. Always returns S_OK.
- HRESULT EnterCriticalSection(); blocked until the Critical Section is released.
- HRESULT LeaveCriticalSection();
  Leaves the Critical Section and thus releases it again.
  Returns MAKE_RTOS_HRESULT(OS_CS_ERR) if the caller is not the owner.

EnterCriticalSection() and LeaveCriticalSection() must be called in the RT context, otherwise the following return value is returned:

ADS_E_INVALIDCONTEXT	Return value if Critical Section is entered outside the RT context. The Critical Section is not entered.

If these methods are used without initializing the Critical Section, S_OK is returned without further action. All other methods can be used in the RT context as well as in the Windows context.

In the CCriticalSectionInstance class, the EnterCriticalSection() can be called multiple times by the same task. Leave() must then be called just as often.

CriticalSections allow nested calls and require an associated LeaveCriticalSection() call for each EnterCriticalSection() call. The Critical Section is released when the last LeaveCritcalSection() is called.

Sample:

The sample [▶ 284] shows how to use a CriticalSection to prevent concurrent access to a date via an interface method call.

Critical Section Concurrent

The CCriticalSectionInstanceConcurrent class allows concurrent access. This can be used if several tasks have read access to the data to be protected, but all other accesses must be prevented in the case of write access.

Methods:

- CCriticalSectionInstanceConcurrent(UINT concurrent, OTCID oid=0, ITComObjectServer* ipSrv=NULL); The parameter "concurrent" defines the number of tasks that can simultaneously enter the Critical Section via CsEnterCriticalSectionConcurrent(). If "concurrent" is 1, the Critical Section Concurrent variant works like a Critical Section.
  The value 0 is not allowed. It would prevent the Critical Section from being initialized.
- ~CCriticalSectionInstanceConcurrent(); The destructor implicitly deletes the Critical Section
- void SetOidCriticalSection(OTCID oid); Sets the Critical Section Provider, which is given by TwinCAT real-time and whose object ID is available via OID_TCRTIME_CTRL.
- bool HasOidCriticalSection(); Returns TRUE if the object ID is set to a value other than 0, otherwise FALSE. The object ID is not checked to verify whether it belongs to a Critical Section Provider.
- bool IsInitializedCriticalSection(); Returns TRUE if the Critical Section was successfully initialized.
- HRESULT CreateCriticalSection(ITComObjectServer* ipSrv); If the Critical Section is initialized, it is deleted and a new Critical Section is initialized. Returns S_OK if successful. Error cases are indicated by the return of error codes:

ADS_E_INVALIDPARM	Invalid Critical Section Provider
ADS_E_NOINTERFACE	The object ID is set to a reference that is not a Critical Section Provider, i.e. ITcRTime is not implemented.
E_FAIL	Internal error from the Critical Section Provider.

- HRESULT CreateCriticalSection(OTCID oid, ITComObjectServer* ipSrv);
  Initializes the Critical Section. DeleteCriticalSection() must be called if this method is used again. Return values as in CreateCriticalSection(ITComObjectServer* ipSrv);
- HRESULT DeleteCriticalSection(); Critical Section is deleted. Always returns S_OK.
- HRESULT EnterCriticalSection(); Blocked until the Critical Section is released and can be accessed exclusively.
- HRESULT LeaveCriticalSection(); Leaves the Critical Section and thus releases it again. Returns MAKE_RTOS_HRESULT(OS_CS_ERR) if the caller is not the owner.
- HRESULT EnterCriticalSectionConcurrent(); Enters the Critical Section in parallel with other tasks. The number of parallel accesses is defined by the parameter "concurrent" in the constructor. When this maximum number is reached, the call is blocked until one of the parallel accesses has ended.

EnterCriticalSection(), EnterCriticalSectionConcurrent() and LeaveCriticalSection() must be called in the RT context, otherwise the following return value is returned:

ADS_E_INVALIDCONTEXT	Return value if Critical Section is entered outside the RT context. The Critical Section is not entered.

If these methods are used without initializing the Critical Section, S_OK is returned without further action. All other methods can be used in the RT context as well as in the Windows context.

For the CCriticalSectionInstanceConcurrent class, it is not allowed for the EnterCriticalSection() method to be called multiple times by the same task if the concurrent parameter is greater than 1 during initialization.

12.7.2 Semaphores

Semaphores are used to synchronize access to limited resources. They can also be used to implement a notification event. Semaphores are provided by the TwinCAT real-time through the CSemaphoreInstance class.

CSemaphoreInstance

The CSemaphoreInstance class provides the interface for handling semaphores. To create a semaphore, the instance requires the object ID of the TwinCAT real-time instance, which is available via OID_TCRTIME_CTRL, as well as a reference to the TwinCAT object server via a pointer to ITComObjectServer.

Methods:

- HRESULT SemCreate(WORD nCntInit, OTCID oid, ITComObjectServer* ipSrv); Creates the semaphores with nCntInit as initial value of the available resources. Returns S_OK if successful and E_FAIL if the semaphore cannot be generated.
- HRESULT SemDelete(); SemDelete() deletes the semaphore. Returns S_OK.
- HRESULT SemPost() SemPost() increases the number of available resources. If a task is waiting for the semaphores, the task with the highest priority is released, i.e. it can continue its execution. Returns S_OK if successful. Possible error codes:

MAKE_RTOS_HRESULT(51)	Semaphore overflow. For example, if the internal memory is insufficient.

- HRESULT SemPend(OSTICKS nTimeout); Reduces the number of available resources; if no resources are available, the task is blocked at this point.
  SemPend() waits for a semaphore until it is available. A timeout can be specified in OSTICKS and is

therefore processor dependent. The ITcRTime interface in can be used to calculate it. In the sample TcSemaphoreSample a method TimeoutMsToTicks was used for this purpose. Special values for nTimeout:

RTIME_NOWAIT (-1)	Method immediately returns S_OK if the semaphore is available. Method returns a timeout if no resource is available.
RTIME_ENDLESSWAIT (0)	Method waits indefinitely for availability of the semaphores

The method returns S_OK if the semaphore was successfully obtained; otherwise:

MAKE_RTOS_HRESULT(10)	Indicates a timeout. If nTimeout is set as RTIME_NOWAIT, this semaphore is not available.

Sample:

Sample24: Semaphores [▶ 295]

12.7.3 FIFO template classes

The header file Fifo.h provides different FIFO Template classes. All classes recognize when their memory is full and will not accept new elements until an element is removed. The element types can be primitive data types ("Plain Old Datatypes", PODs). It is not possible to store element types with virtual tables in a FIFO buffer. Pointers are allowed (irrespective of type).

CFiFoListBase

The class CFifoListBase is the base implementation of a FIFO. This template class has a StorageType instance that is responsible for storing elements of one type. It uses a SyncType instance to maintain synchronized access to the storage and data it contains.

There are different protection options:

- CNoSync / SyncObects.h: No access protection within the FIFO. An external CriticalSection can be used.
- CSpinLock / TcSpinlock.h: Protection by a spinlock, as described above.

The Add(), Remove() and Clear() methods have corresponding _Unprotected_ options, which can be used with the Lock() and Unlock() methods to implement a synchronized sequence of Add() and Delete() operations.

Methods:

- CFiFoListBase(); The constructor.
- Lock() and Unlock(); Are passed to the SyncType synchronization instance.
- Clear(), Clear_Unprotected(); Are passed on to the StorageType instance.
- Add(), Add_Unprotected(); Add an element.
- AddBlock(); Add multiple elements. If a block is larger than the free memory, only the elements that match the free memory are added. The number of added elements is returned.
- Remove(), Remove_Unprotected(); Removes an element from the FIFO. Returns FALSE if no element was present, otherwise TRUE.
- RemoveBlock(); Removes multiple elements from the FIFO. If fewer elements are stored there, only the stored elements are removed. The number of removed elements is returned.
- GetEntryXBeforeHead(); Fetches the element before the xth element from the head without removing the element. For example, if 0 is used as parameter, the first element is fetched. Otherwise the parameter must be greater than 0. Returns FALSE if the element does not exist, otherwise TRUE.
- Count(); Returns the number of stored elements.
- Size(); Returns the size of the memory, i.e. the maximum number of elements the memory can hold.
- FreeCount(); Returns the number of elements added to the memory.
- GetNextEntry(); Iterates over the elements and starts at the head.

CFiFoList

The class CFiFoList is based on CFiFoListBase and uses CFiFoListStorageStatic as storage type in conjunction with the synchronization type CSyncDefault.
This implementation does not add methods and requires element types and memory size as template parameters.

CFiFoListDyn

The class CFiFoListDyn uses the dynamic StorageType CFifoListStorageDynamic in conjunction with the synchronization type CSyncDefault.
This implementation does not add methods and requires element types and memory size as template parameters.

Handover3Buffer

The class CHandover3Buffer is used to pass elements from one task to another. The memory contains 3 elements of a given type. The writing task can always write the element. The reading task can always read the element when an element has been written. It reads the element that was written fully. Elements are not removed after they are read. Access via indices for reading and writing to memory is synchronized by the InterlockedExchangeR0 function, so that it is accessible from the Windows and RT context.

Methods:

- CHandover3Buffer(); The constructor.
- Write(); Writes an element into memory. Returns FALSE if the memory has an invalid state. Otherwise TRUE.
- Read(); Reads an element from memory. Returns FALSE if the memory is empty or is in an invalid state. Otherwise TRUE.

CHandoverFifo

The CHandoverFifo class is also used to exchange elements between tasks. The size of the memory is always a power of two. It can be set via the template parameter DEPTH_EXP. The default value is 3, resulting in a size of 8 elements.

Methods:

- Insert()/Add(); Writes an element into memory via a pointer or reference. Returns FALSE if no memory is available, otherwise TRUE.
- Remove(); Removes and returns an element from memory via a pointer or reference. Returns FALSE if no element is available, otherwise TRUE.

Sample:
TcHandoverSample.tszip

12.8 Interfaces

Several interfaces are available for the interaction of the modules developed by the user with the TwinCAT 3 system. These are described (at API level) in detail on these pages.

Name	Description
ITcUnknown [▶ 187]	ITcUnknown defines the reference count as well as the querying of a reference to a more specific interface.
ITComObject [▶ 177]	The ITComObject interface is implemented by every TwinCAT module.
ITcCyclic [▶ 161]	The interface is implemented by TwinCAT modules that are called once per task cycle.
ITcCyclicCaller [▶ 162]	Interface for logging the ItcCyclic interface of a module onto and off from a TwinCAT task.
ITcFileAccess [▶ 164]	Interface for accessing the file system
ITcFileAccessAsync [▶ 171]	Asynchronous access to file operations.

Name	Description
ITcPostCyclic [▶ 182]	The interface is implemented by TwinCAT modules that are called once per task cycle following the output update.
ITcPostCyclicCaller [▶ 159]	Interface for logging the ITcPostCyclic interface of a module onto and off from a TwinCAT task.
ITcIoCyclic [▶ 172]	This interface is implemented by TwinCAT modules that are called during the input update and output update within a task cycle.
ITcIoCyclicCaller [▶ 174]	Interface for logging the ITcIoCyclic interface of a module onto and off from a TwinCAT task.
ITcRTimeTask [▶ 183]	Query of extended TwinCAT task information.
ITcTask [▶ 183]	Query of the timestamp and task-specific information of a TwinCAT task.
ITcTaskNotification [▶ 187]	Executes a callback if the cycle time was exceeded during the previous cycle.

TwinCAT SDK

TwinCAT SDK contains a number of functions, which can be found in *C:\TwinCAT\3.1\sdk\Include*.

- The TcCOM framework is provided here (in particular TcInterfaces.h and TcServices.h).
- Tasks and data area access is provided via TcIoInterfaces.h.
- SDK functions are the mathematical functions [▶ 206].
- Subset of STL [▶ 208].
- TwinCAT runtime RtIR0.h
- Methods for ADS communication [▶ 189]
- Classes / functions with names beginning with "Os" must not be used in a real-time context.

12.8.1 Interface ITcPostCyclicCaller

Interface for logging in or out of the ITcPostCyclic interface of a module in a TwinCAT task.

Syntax

```
TCOM_DECL_INTERFACE("03000026-0000-0000-e000-000000000064", ITcCyclicCaller)
struct __declspec(novtable) ITcPostCyclicCaller : public ITcUnknown Ca
```

Required include: `TcIoInterfaces.h`

Methods

Name	Description
AddPostModule [▶ 159]	Login module that implements the ITcPostCyclic interface.
RemovePostModule [▶ 160]	Log off the previously logged in ITcPostCyclic interface of a module.

The ITcPostCyclicCaller interface is implemented by TwinCAT tasks. A module uses this interface to login its ITcPostCyclic interface to a task, usually as the last initialization step in the SafeOP to OP transition. After login, the PostCycleUpdate() method of the module instance is called. The interface is also used to log off the module so that it is no longer called by the task.

12.8.1.1 Method ITcPostCyclicCaller:AddPostModule

Logs the ITcPostCyclic interface of a module to a cyclic caller, e.g. a TwinCAT task.

Syntax

```
virtual HRESULT TCOMAPI
AddPostModule(STcPostCyclicEntry* pEntry, ITcPostCyclic* ipMod, ULONG_PTR
context=0, ULONG sortOrder=0)=0;
```

Return value

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [▶ 161]. Extended messages refer in particular to the column HRESULT in ADS Return Codes [▶ 323].

The error ADSERR_DEVICE_INVALIDSTATE is returned if the cyclic caller, i.e. the TwinCAT task is not in the OP state.

Parameter

Name	Type	Description
pEntry	STcPostCyclicEntry	[in] Pointer to a list item that is inserted into the internal list of the cyclic caller; see also description.
ipMod	ITcPostCyclic	[in] Interface pointer used by the cyclic caller.
context	ULONG_PTR	[optional] A context value that is transferred to the ITcPostCyclic::PostCyclicUpdate() method with each call.
sortOrder	ULONG	[optional] The sort order can be used for controlling the order of execution if various module instances are executed by the same cyclic caller.

Description

A TwinCAT module class uses a Smart Pointer to refer to the cyclic caller of type ITcPostCyclicCallerPtr. The object ID of the task is stored in this Smart Pointer and a reference to the task can be obtained via the TwinCAT object server. In addition, the Smart Pointer class already contains a list item. Therefore the Smart Pointer can be used as the first parameter for the AddPostModule method.

The following code sample illustrates the login of the ITcPostCyclicCaller interface.

```
RESULT hr =
S_OK;

if ( m_spPostCyclicCaller.HasOID() ) {

if ( SUCCEEDED_DBG(hr =
m_spSrv->TcQuerySmartObjectInterface(m_spPostCyclicCaller)) )
{

if ( FAILED(hr =
m_spPostCyclicCaller->AddPostModule(m_spPostCyclicCaller,
THIS_CAST(ITcPostCyclic))) ) {

m_spPostCyclicCaller = NULL;

}

}

}
```

12.8.1.2 Method ITcPostCyclicCaller:RemovePostModule

Remove a module instance from being called by a cyclic caller.

Syntax

```
virtual HRESULT TCOMAPI
RemovePostModule(STcPostCyclicEntry* pEntry)=0;
```

Return value

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [▶ 161]. Extended messages refer in particular to the column HRESULT in ADS Return Codes [▶ 323].

The method returns E_FAIL if the entry is not in the internal list.

Parameter

Name	Type	Description
pEntry	STcPostCyclicEntry	Refers to the list item to be removed from the internal list of the cyclic caller.

Comparable with the AddPostModule() method, the smart pointer is used for the cyclic caller as a list item if the module instance is to be removed from the cyclic caller.

Declaration and use of the Smart Pointer:

ITcPostCyclicCallerInfoPtr m_spPostCyclicCaller;

```
if (
m_spPostCyclicCaller ) {
m_spPostCyclicCaller->RemovePostModule(m_spPostCyclicCaller);

}

m_spPostCyclicCaller = NULL;
```

12.8.2 Return values

ITc interfaces methods generally return an HRESULT.

The following return values can be returned in the case of ITc interfaces:

Name	HRESULT
S_OK	0x0000 0000
S_FALSE	0x0000 0001
E_NOTIMPL	0x8000 4001
E_NOINTERFACE	0x8000 4002
E_POINTER	0x8000 4003
E_ABORT	0x8000 4004
E_FAIL	0x8000 4005
E_UNEXPECTED	0x8000 FFFF
E_ACCESSDENIED	0x8007 0005
E_HANDLE	0x8007 0006
E_OUTOFMEMORY	0x8007 000E
E_INVALIDARG	0x8007 0057

In addition, there is a possibility for ADS Return Codes [▶ 323] to be returned as HRESULT. These are also available as macros in the SDK, where they are known, for example, as ADS_E_BUSY for the ADS Error Code ADSERR_DEVICE_BUSY.

12.8.3 Interface ITcCyclic

The ITcCyclic interface is implemented by TwinCAT modules that are called once per task cycle.

Syntax

```
TCOM_DECL_INTERFACE("03000010-0000-0000-e000-000000000064", ITcCyclic)
struct __declspec(novtable) ITcCyclic : public ITcUnknown
```

Required include: `TcIoInterfaces.h`

Methods

Name	Description
CycleUpdate [▶ 162]	Is called once per task cycle if the interface is logged on to a cyclic caller.

The ITcCyclic interface is implemented by TwinCAT modules. This interface is passed to the ITcCyclicCaller::AddModule() method when a module logs itself on to a task, usually as the last initialization step during the transition from SafeOP to OP. After login, the CycleUpdate() method of the module instance is called.

12.8.3.1 Method ITcCyclic: CycleUpdate

The CycleUpdate method is normally called by a TwinCAT task after the interface has been logged in.

Syntax

```
HRESULT TCOMAPI CycleUpdate(ITcTask* ipTask, ITcUnknown* ipCaller, ULONG_PTR context)
```

Return value

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [▶ 161]. Extended messages refer in particular to the column HRESULT in ADS Return Codes [▶ 323].

At present, the return value is ignored by the TwinCAT tasks.

Parameter

Name	Type	Description
ipTask	ITcTask	Refers to the current task context.
ipCaller	ITcUnknown	Refers to the calling instance.
Context	ULONG_PTR	Context contains the value that was passed to the ITcCyclicCaller::AddModule() method.

Description

In a task cycle, the CycleUpdate() method is called after InputUpdate() has been called for all registered module instances. As a result, this method must be used to implement cyclic processing.

12.8.4 Interface ITcCyclicCaller

Interface for logging in or out of the ITcCyclic interface of a module in a TwinCAT task.

Syntax

```
TCOM_DECL_INTERFACE("0300001E-0000-0000-e000-000000000064", ITcCyclicCaller)
struct __declspec(novtable) ITcCyclicCaller : public ITcUnknown
```

Required include: `TcIoInterfaces.h`

Methods

Name	Description
AddModule [▶ 163]	Login module that implements the ITcCyclic interface.
RemoveModule [▶ 164]	Log off the previously logged in ITcCyclic interface of a module.

The ITcCyclicCaller interface is implemented by TwinCAT tasks. A module uses this interface to login its ITcCyclic interface to a task, usually as the last initialization step during the transition from SafeOP to OP. After login, the CycleUpdate() method of the module instance is called. The interface is also used to log off the module so that it is no longer called by the task.

12.8.4.1 Method ITcCyclicCaller:AddModule

Logs the ITcCyclic interface of a module to a cyclic caller, e.g. a TwinCAT task.

Syntax

```
virtual HRESULT TCOMAPI
AddModule(STcCyclicEntry* pEntry, ITcCyclic* ipMod, ULONG_PTR
context=0, ULONG sortOrder=0)=0;
```

Return value

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [▶ 161]. Extended messages refer in particular to the column HRESULT in ADS Return Codes [▶ 323].

The error ADSERR_DEVICE_INVALIDSTATE is returned if the cyclic caller, i.e. the TwinCAT task, is not in the OP state.

Parameter

Name	Type	Description
pEntry	STcCyclicEntry	[in] Pointer to a list item that is inserted into the internal list of the cyclic caller; see also description [▶ 163].
ipMod	ITcCyclic	[in] Interface pointer used by the cyclic caller.
context	ULONG_PTR	[optional] A context value that is passed to the ITcCyclic::CycleUpdate() method each time it is called.
sortOrder	ULONG	[optional] The sort order can be used for controlling the order of execution if various module instances are executed by the same cyclic caller.

Description

A TwinCAT module class normally uses a Smart Pointer to refer to the cyclic caller type ITcCyclicCallerPtr. The object ID of the task is stored in this Smart Pointer and a reference to the task can be obtained via the TwinCAT object server. In addition, the Smart Pointer class already contains a list item. Therefore the Smart Pointer can be used as the first parameter for the AddModule method.

The following sample code shows the login of the ITcCyclicCaller interface.

```
RESULT hr =
S_OK;

if ( m_spCyclicCaller.HasOID() ) {

if ( SUCCEEDED_DBG(hr =
m_spSrv->TcQuerySmartObjectInterface(m_spCyclicCaller)) )
{

if ( FAILED(hr =
m_spCyclicCaller->AddModule(m_spCyclicCaller,
THIS_CAST(ITcCyclic))) ) {

m_spCyclicCaller = NULL;

}

}

}
```

12.8.4.2 Method ITcCyclicCaller:RemoveModule

Remove a module instance from being called by a cyclic caller.

Syntax

```
virtual HRESULT TCOMAPI
RemoveModule(STcCyclicEntry* pEntry)=0;
```

Return value

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [▶ 161]. Extended messages refer in particular to the column HRESULT in ADS Return Codes [▶ 323].

Parameter

Name	Type	Description
pEntry	STcCyclicEntry	Refers to the list item to be removed from the internal list of the cyclic caller.

The method returns E_FAIL if the entry is not in the internal list.

Comparable with the AddModule() method, the smart pointer is used for the cyclic caller as a list item if the module instance is to be removed from the cyclic caller.

Declaration and use of the Smart Pointer:

ITcCyclicCallerInfoPtr m_spCyclicCaller;

```
if (
m_spCyclicCaller ) {
m_spCyclicCaller->RemoveModule(m_spCyclicCaller);
}
m_spCyclicCaller = NULL;
```

12.8.5 Interface ITcFileAccess

Interface for accessing the file system from TwinCAT C++ modules

Syntax

```
TCOM_DECL_INTERFACE("742A7429-DA6D-4C1D-80D8-398D8C1F1747", ITcFileAccess) __declspec(novtable)
ITcFileAccess: public ITcUnknown
```

Required include: `TcFileAccessInterfaces.h`

Methods

Name	Description
FileOpen [▶ 165]	Opens a file.
FileClose [▶ 166]	Closes a file.
FileRead [▶ 166]	Reads from a file.
FileWrite [▶ 166]	Writes to a file.
FileSeek [▶ 167]	Specifies the position within a file for further actions.
FileTell [▶ 167]	Queries the currently set position within a file.
FileRename [▶ 167]	Renames a file.
FileDelete [▶ 168]	Deletes a file.
FileGetStatus [▶ 168]	Gets the status of a file.
FileFindFirst [▶ 169]	Searches for a file, first iteration.
FileFindNext [▶ 169]	Searches for a file, next iteration.

Name	Description
FileFindClose [▶ 170]	Closes a file search.
MkDir [▶ 170]	Creates a directory.
RmDir [▶ 171]	Deletes a directory.

The ITcFileAccess interface is used to access files in file systems.
Because the methods provided cause blockages, this interface should not be used in CycleUpdate() / real-time context. The derived interface ITcFileAccessAsync [▶ 171] adds a Check() method that can be used instead.

See Sample20a: FileIO-Cyclic Read / Write [▶ 290].

The interface is implemented via the CID_TcFileAccess module class.

12.8.5.1 Method ITcFileAccess:FileOpen

Opens a file.

Syntax

```
virtual HRESULT TCOMAPI FileOpen(PCCH szFileName, TcFileAccessMode AccessMode, PTcFileHandle
phFile);
```

Return value

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [▶ 161]. Extended messages refer in particular to the column HRESULT in ADS Return Codes [▶ 323].

A particularly interesting error code is ADSERR_DEVICE_TIMEOUT if the timeout (5 seconds) has elapsed.

Parameter

Name	Type	Description
szFileName	PCCH	[in] the name of the file to be opened.
AccessMode	TcFileAccessMode	[in] method of accessing the file; see TcFileAccessServices.h.
phFile	TcFileHandle	[out] returned file handle.

AccessModes can be used as follows:

```
typedef enum TcFileAccessMode
{
amRead = 0x00000001,
amWrite = 0x00000002,
amAppend = 0x00000004,
amPlus = 0x00000008,
amBinary = 0x00000010,
amReadBinary = 0x00000011,
amWriteBinary = 0x00000012,
amText = 0x00000020,
amReadText = 0x00000021,
amWriteText = 0x00000022,
amEnsureDirectory = 0x00000040,
amReadBinaryED = 0x00000051,
amWriteBinaryED = 0x00000052,
amReadTextED = 0x00000061,
amWriteTextED = 0x00000062,
amEncryption = 0x00000080,
amReadBinEnc = 0x00000091,
amWriteBinEnc = 0x00000092,
amReadBinEncED = 0x000000d1,
amWriteBinEncED = 0x000000d2,
} TcFileAccessMode, *PTcFileAccessMode;
```

12.8.5.2 Method ITcFileAccess:FileClose

Closes a file.

Syntax

```
virtual HRESULT TCOMAPI FileClose(PTcFileHandle phFile);
```

Return value

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [▶ 161]. Extended messages refer in particular to the column HRESULT in ADS Return Codes [▶ 323].

A particularly interesting error code is ADSERR_DEVICE_TIMEOUT if the timeout (5 seconds) has elapsed.

Parameter

Name	Type	Description
phFile	TcFileHandle	[out] returned file handle of the file to be closed

12.8.5.3 Method ITcFileAccess:FileRead

Reads data from a file.

Syntax

```
virtual HRESULT TCOMAPI
FileRead(TcFileHandle hFile, PVOID pData, UINT cbData, PUINT pcbRead);
```

Return value

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [▶ 161]. Extended messages refer in particular to the column HRESULT in ADS Return Codes [▶ 323].

A particularly interesting error code is ADSERR_DEVICE_TIMEOUT if the timeout (5 seconds) has elapsed.

Parameter

Name	Type	Description
hFile	TcFileHandle	[in] refers to the previously opened file.
pData	PVOID	[out] storage location of the data to be read.
cbData	PVOID	[in] maximum size of the data to be read (size of the memory behind pData).
pcbRead	PUINT	[out] size of the data read.

12.8.5.4 Method ITcFileAccess:FileWrite

Write data to a file.

Syntax

```
virtual HRESULT TCOMAPI
FileWrite(TcFileHandle hFile, PCVOID pData, UINT cbData, PUINT pcbWrite);
```

Return value

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [▶ 161]. Extended messages refer in particular to the column HRESULT in ADS Return Codes [▶ 323].

A particularly interesting error code is ADSERR_DEVICE_TIMEOUT if the timeout (5 seconds) has elapsed.

Parameter

Name	Type	Description
hFile	TcFileHandle	[in] refers to the previously opened file.
pData	PVOID	[in] storage location of the data to be written.
cbData	PVOID	[in] size of the data to be written (size of the memory behind pData).
pcbRead	PUINT	[out] size of the written data.

12.8.5.5 Method ITcFileAccess:FileSeek

Specifies the position within a file for further actions.

Syntax

```
virtual HRESULT TCOMAPI FileSeek(TcFileHandle hFile, UINT uiPos);
```

Return value

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [▶ 161]. Extended messages refer in particular to the column HRESULT in ADS Return Codes [▶ 323].

A particularly interesting error code is ADSERR_DEVICE_TIMEOUT if the timeout (5 seconds) has elapsed.

Parameter

Name	Type	Description
hFile	TcFileHandle	[in] refers to the previously opened file.
uiPos	UINT	[in] position at which setting is to take place.

12.8.5.6 Method ITcFileAccess:FileTell

Queries the currently set position within a file.

Syntax

```
virtual HRESULT TCOMAPI FileTell(TcFileHandle hFile, PUINT puiPos);
```

Return value

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [▶ 161]. Extended messages refer in particular to the column HRESULT in ADS Return Codes [▶ 323].

A particularly interesting error code is ADSERR_DEVICE_TIMEOUT if the timeout (5 seconds) has elapsed.

Parameter

Name	Type	Description
hFile	TcFileHandle	[in] refers to the previously opened file.
puiPos	PUINT	[out] storage location of the position to be returned.

12.8.5.7 Method ITcFileAccess:FileRename

Renames a file.

Syntax

```
virtual HRESULT TCOMAPI FileRename(PCCH szOldName, PCCH szNewName);
```

Return value

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [▶ 161]. Extended messages refer in particular to the column HRESULT in ADS Return Codes [▶ 323].

A particularly interesting error code is ADSERR_DEVICE_TIMEOUT if the timeout (5 seconds) has elapsed.

Parameter

szOldName	PCCH	[in] the file name to be changed.
szNewName	PCCH	[in] the new filename.

12.8.5.8 Method ITcFileAccess:FileDelete

Deletes a file.

Syntax

```
virtual HRESULT TCOMAPI FileDelete(PCCH szFileName);
```

Return value

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [▶ 161]. Extended messages refer in particular to the column HRESULT in ADS Return Codes [▶ 323].

A particularly interesting error code is ADSERR_DEVICE_TIMEOUT if the timeout (5 seconds) has elapsed.

Parameter

Name	Type	Description
szFileName	PCCH	[in] name of the file to be deleted.

12.8.5.9 Method ITcFileAccess:FileGetStatus

Queries the status of a file.

Syntax

```
virtual HRESULT TCOMAPI FileGetStatus(PCCH szFileName, PTcFileStatus pFileStatus));
```

Return value

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [▶ 161]. Extended messages refer in particular to the column HRESULT in ADS Return Codes [▶ 323].

A particularly interesting error code is ADSERR_DEVICE_TIMEOUT if the timeout (5 seconds) has elapsed.

Parameter

Name	Type	Description
szFileName	PCCH	[in] the name of the file in question.
pFileStatus	PTcFileStatus	[out] the status of the file, cf. TcFileAccessServices.h .

Description

This method queries status information regarding a given file name.

This includes the following information:

```
typedef struct TcFileStatus
{
union
{
ULONGLONG ulFileSize;
struct
{
ULONG ulFileSizeLow;
ULONG ulFileSizeHigh;
};
};
ULONGLONG ulCreateTime;
ULONGLONG ulModifiedTime;
ULONGLONG ulReadTime;
DWORD dwAttribute;
DWORD wReserved0;
} TcFileStatus, *PTcFileStatus;
```

12.8.5.10 Method ITcFileAccess:FileFindFirst

Option to scroll through the files in a directory.

Syntax

```
virtual HRESULT TCOMAPI FileFindFirst (PCCH szFileName, PTcFileFindData pFileFindData ,
PTcFileFindHandle phFileFind);
```

Return value

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [▶ 161]. Extended messages refer in particular to the column HRESULT in ADS Return Codes [▶ 323].

A particularly interesting error code is ADSERR_DEVICE_TIMEOUT if the timeout (5 seconds) has elapsed.

Parameter

Name	Type	Description
szFileName	PCCH	[in] directory or path and name of the file sought. The file name can contain placeholders such as asterisk (*) or question mark (?).
pFileFindData	PTcFileFindData	[out] the description of the first file, cf. TcFileAccessServices.h
phFileFind	PTcFileFindHandle	[out] handle for searching further with FileFindNext.

Description

This method starts by searching for files in a given directory. The method grants access to PTcFileFindData of the first file found, with the following information:

```
typedef struct TcFileFindData
{
TcFileHandle hFile;
DWORD dwFileAttributes;
ULONGLONG ui64CreationTime;
ULONGLONG ui64LastAccessTime;
ULONGLONG ui64LastWriteTime;
DWORD dwFileSizeHigh;
DWORD dwFileSizeLow;
DWORD dwReserved1;
DWORD dwReserved2;
CHAR cFileName[260];
CHAR cAlternateFileName[14];
WORD wReserved0;
} TcFileFindData, *PTcFileFindData;
```

12.8.5.11 Method ITcFileAccess:FileFindNext

Continue scanning the files in a directory.

Syntax

```
virtual HRESULT TCOMAPI FileFindNext (TcFileFindHandle hFileFind, PTcFileFindData pFileFindData);
```

Return value

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [▶ 161]. Extended messages refer in particular to the column HRESULT in ADS Return Codes [▶ 323].

A particularly interesting error code is ADSERR_DEVICE_TIMEOUT if the timeout (5 seconds) has elapsed.

Parameter

Name	Type	Description
hFileFind	PTcFileFindHandle	[in] handle for searching further with FileFindNext.
pFileFindData	PTcFileFindData	[out] the description of the next file. Compare with TcFileAccessServices.h

Description

This method searches for the next file in a directory. The method grants access to PTcFileFindData of the file found, with the following information:

```
typedef struct TcFileFindData
{
TcFileHandle hFile;
DWORD dwFileAttributes;
ULONGLONG ui64CreationTime;
ULONGLONG ui64LastAccessTime;
ULONGLONG ui64LastWriteTime;
DWORD dwFileSizeHigh;
DWORD dwFileSizeLow;
DWORD dwReserved1;
DWORD dwReserved2;
CHAR cFileName[260];
CHAR cAlternateFileName[14];
WORD wReserved0;
} TcFileFindData, *PTcFileFindData;
```

12.8.5.12 Method ITcFileAccess:FileFindClose

Close the search for files in a directory.

Syntax

```
virtual HRESULT TCOMAPI FileFindClose (TcFileFindHandle hFileFind);
```

Return value

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [▶ 161]. Extended messages refer in particular to the column HRESULT in ADS Return Codes [▶ 323].

A particularly interesting error code is ADSERR_DEVICE_TIMEOUT if the timeout (5 seconds) has elapsed.

Parameter

Name	Type	Description
hFileFind	PTcFileFindHandle	[in] handle to exit the search.

12.8.5.13 Method ITcFileAccess:MkDir

Create a directory in a file system.

Syntax

```
virtual HRESULT TCOMAPI MkDir(PCCH szDir);
```

Return value

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [▶ 161]. Extended messages refer in particular to the column HRESULT in ADS Return Codes [▶ 323].

A particularly interesting error code is ADSERR_DEVICE_TIMEOUT if the timeout (5 seconds) has elapsed.

Parameter

Name	Type	Description
szDir	PCCH	[in] directory to be created.

12.8.5.14 Method ITcFileAccess:RmDir

Remove a directory from the file system.

Syntax

```
virtual HRESULT TCOMAPI RmDir(PCCH szDir);
```

Return value

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [▶ 161]. Extended messages refer in particular to the column HRESULT in ADS Return Codes [▶ 323].

A particularly interesting error code is ADSERR_DEVICE_TIMEOUT if the timeout (5 seconds) has elapsed.

Parameter

Name	Type	Description
szDir	PCCH	[in] directory to be deleted.

12.8.6 Interface ITcFileAccessAsync

Asynchronous access to file operations. This interface extends ITcFileAccess [▶ 164].

Syntax

```
TCOM_DECL_INTERFACE("C04AC244-C126-466E-982E-93EC571F2277", ITcFileAccessAsync) struct
__declspec(novtable) ITcFileAccessAsync: public ITcFileAccess
```

Required include: `TcFileAccessInterfaces.h`

Properties

Name	Description
PID_TcFileAccessAsyncSegmentSize	Size of the segments transferred to system service.
PID_TcFileAccessAsyncTimeoutMs	Sets the timeout in [ms].
PID_TcFileAccessAsyncNetId(Str)	NetId of the system service to be contacted.

Methods

Name	Description
Check [▶ 172]	Check the status of the previously called file operation.

Interface can be obtained from module instance with class ID CID_TcFileAccessAsync.
When using the asynchronous interface, the interface methods inherited from the synchronous variant return ADS_E_PENDING if a query has been successfully submitted but not yet completed. If the call is received while the previous request was still being processed, the error code ADS_E_BUSY is returned.

Description of the module parameters:

- PID_TcFileAccessAsyncAdsProvider: Object ID of a task that provides the ADS interface.
- PID_TcFileAccessAsyncNetId / PID_TcFileAccessAsyncNetIdStr: AmsNetId of the system service used for file access. The "Str" variant takes the AmsNetId as string. Please use one.
- PID_TcFileAccessAsyncTimeoutMs: Timeout for a file access.
- PID_TcFileAccessAsyncSegmentSize: The read and write access to file is fragmented with this segment size.

See Sample20a: FileIO-Cyclic Read / Write [▶ 290].

12.8.6.1 Method ITcFileAccessAsync::Check()

Check the status of the previously called file operation.

Syntax

```
virtual HRESULT TCOMAPI Check();
```

Return value

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [▶ 161]. Extended messages refer in particular to the column HRESULT in ADS Return Codes [▶ 323].

Particularly interesting error codes are

- ADSERR_DEVICE_TIMEOUT if the timeout (5 seconds) has elapsed.
- ADSERR_DEVICE_PENDING if the file operation is not completed.

Parameter

none

12.8.7 Interface ITcIoCyclic

This interface is implemented by TwinCAT modules that are called during the input update and output update within a task cycle.

Syntax

```
TCOM_DECL_INTERFACE("03000011-0000-0000-e000-000000000064", ITcIoCyclic)
struct __declspec(novtable) ITcIoCyclic : public ITcUnknown
```

Required include: `TcIoInterfaces.h`

Methods

Name	Description
InputUpdate [▶ 173]	Is called at the beginning of a task cycle if the interface is logged on to a cyclic I/O caller.
OutputUpdate [▶ 173]	Is called at the end of a task cycle if the interface is logged on to a cyclic I/O caller.

ITcIoCyclic can be used to implement a TwinCAT module that acts as a fieldbus driver or I/O filter module.

This interface is passed to the method ITcIoCyclicCaller::AddIoDriver when a module logs on to a task, usually as the last initialization step in the transition from SafeOP to OP. After login, the InputUpdate() and OutputUpdate() methods of the module instance are each called once per task cycle.

12.8.7.1 Method ITcIoCyclic:InputUpdate

Is called at the beginning of a task cycle if the interface is logged on to a cyclic I/O caller.

Syntax

```
virtual HRESULT TCOMAPI InputUpdate(ITcTask* ipTask,
ITcUnknown* ipCaller, DWORD dwStateIn, ULONG_PTR context = 0)=0;
```

Return value

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [▶ 161]. Extended messages refer in particular to the column HRESULT in ADS Return Codes [▶ 323].

Parameter

Name	Type	Description
ipTask	ITcTask	Refers to the current task context.
ipCaller	ITcUnknown	Refers to the calling instance.
dwStateIn	DWORD	Future extensions reserved; at present this value is always 0.
context	ULONG_PTR	Context contains the value that was passed to the ITcCyclicCaller::AddIoDriver() method.

Description

In a task cycle the InputUpdate() method is first called for all registered module instances. Therefore this method must be used for updating the data areas of the Input-Source type of the module.

12.8.7.2 Method ITcIoCyclic:OutputUpdate

Is called at the end of a task cycle if the interface is logged on to a cyclic I/O caller.

Syntax

```
virtual HRESULT TCOMAPI OutputUpdate(ITcTask* ipTask, ITcUnknown* ipCaller,
PDWORD pdwStateOut = NULL, ULONG_PTR context = 0)=0;
```

Return value

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [▶ 161]. Extended messages refer in particular to the column HRESULT in ADS Return Codes [▶ 323].

Parameter

Name	Type	Description
ipTask	ITcTask	Refers to the current task context.
ipCaller	ITcUnknown	Refers to the calling instance.
pdwStateOut	DWORD	[out] reserved for future extensions, currently the return value is ignored.
context	ULONG_PTR	Context contains the value that was passed to the ITcCyclicCaller::AddIoDriver() method.

Description

In a task cycle the OutputUpdate() method is called for all registered module instances. Therefore this method must be used for updating the data areas of the type Output-Destination of the module.

12.8.8 Interface ITcIoCyclicCaller

Interface for logging in or out of the ITcIoCyclic interface of a module in a TwinCAT task.

Syntax

```
TCOM_DECL_INTERFACE("0300001F-0000-0000-e000-000000000064", ITcIoCyclicCaller)
struct __declspec(novtable) ITcIoCyclicCaller : public ITcUnknown
```

Required include: `TcIoInterfaces.h`

Methods

Name	Description
AddIoDriver [▶ 174]	Login module that implements the ITcIoCyclic interface.
RemoveIoDriver [▶ 175]	Log off the previously logged in ITcIoCyclic interface of a module.

The ITcIoCyclicCaller interface is implemented by TwinCAT tasks. A module uses this interface to login its ITcIoCyclic interface to a task, usually as the last initialization step in the SafeOP to OP transition. After login, the CycleUpdate() method of the module instance is called. The interface is also used to log off the module so that it is no longer called by the task.

12.8.8.1 Method ITcIoCyclicCaller:AddIoDriver

Logs the ITcIoCyclic interface of a module to a cyclic I/O caller, e.g. a TwinCAT task.

Syntax

```
virtual HRESULT TCOMAPI AddIoDriver(STcIoCyclicEntry*
pEntry, ITcIoCyclic* ipDrv, ULONG_PTR context=0, ULONG sortOrder=0)=0;
```

Return value

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [▶ 161]. Extended messages refer in particular to the column HRESULT in ADS Return Codes [▶ 323].

Parameter

Name	Type	Description
pEntry	STcIoCyclicEntry	Pointer to a list item that is inserted into the internal list of the cyclic I/O caller; see also description [▶ 163].
ipDrv	ITcIoCyclic	[in] interface pointer used by the cyclic I/O caller.
context	ULONG_PTR	[optional] a context value that is transferred to the ITcIoCyclic::InputUpdate() and ITcIoCyclic::OutputUpdate methods with each call.
sortOrder	ULONG	[optional] The sort order can be used for controlling the order of execution if various module instances are executed by the same cyclic caller.

Description

A TwinCAT module class normally uses a Smart Pointer to refer to the cyclic I/O caller of type ITcIoCyclicCallerPtr. The object ID of the cyclic I/O caller is stored in this Smart Pointer and a reference can be obtained via the TwinCAT object server. In addition, the Smart Pointer class already contains a list item. Therefore the Smart Pointer can be used as the first parameter for the AddIoDriver method.

The following code sample illustrates the login of the ITcIoCyclicCaller interface.

```
HRESULT hr = S_OK;
if ( m_spIoCyclicCaller.HasOID() )
{
if ( SUCCEEDED_DBG(hr = m_spSrv->TcQuerySmartObjectInterface(m_spIoCyclicCaller))
)
{
if ( FAILED(hr = m_spIoCyclicCaller->AddIoDriver(m_spIoCyclicCaller,
THIS_CAST(ITcIoCyclic))) )
{
m_spIoCyclicCaller = NULL;
}
}
}
```

12.8.8.2 Method ITcIoCyclicCaller:RemoveIoDriver

Remove a module instance from being called by a cyclic I/O caller.

Syntax

```
virtual HRESULT TCOMAPI
RemoveIoDriver(STcIoCyclicEntry* pEntry)=0;
```

Return value

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [▶ 161]. Extended messages refer in particular to the column HRESULT in ADS Return Codes [▶ 323].

The method returns E_FAIL if the entry is not in the internal list.

Parameter

Name	Type	Description
pEntry	STcIoCyclicEntry	Refers to the list item to be removed from the internal list of the cyclic I/O caller.

Comparable with the AddIoDriver() method, the smart pointer is used for the cyclic I/O caller as a list item if the module instance is to be removed from the cyclic I/O caller.

Declaration and use of the Smart Pointer:

```
ITcIoCyclicCallerInfoPtr
m_spIoCyclicCaller;
if ( m_spIoCyclicCaller )
{
m_spIoCyclicCaller->RemoveIoDriver(m_spIoCyclicCaller);
}
m_spCyclicCaller = NULL;
```

12.8.9 ITComOnlineChange interface

The ITComOnlineChange interface is used to perform Online Changes of modules.

Syntax

```
TCOM_DECL_INTERFACE ("D28A8CD2-5477-4B75-AF0F-998841AF9E44", ITComOnlineChange)
```

Methods

Name	Description
PrepareOnlineChange [▶ 176]	This method is called by TwinCAT to prepare for the Online Change.
PerformOnlineChange [▶ 176]	This method is called by TwinCAT to perform the Online Change.

The implementation of this interface is necessary for a module to be capable of Online Change. Furthermore such a module must be created in a versioned C++ project.

- Here [▶ 148] is a general description of the procedure.
- This procedure can be followed for existing modules: Online Change [▶ 148].

12.8.9.1 Method ITComOnlineChange:PrepareOnlineChange

This method is called by TwinCAT to prepare for the Online Change.

It runs asynchronously in the background, which must be taken into account when accessing the existing object.
The preparation should include all operations that can already be performed.

Syntax

```
virtual HRESULT TCOMAPI PrepareOnlineChange(ITComObject* ipOldObj, TmcInstData* pOldInfo) = 0;
```

Return value

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [▶ 161]. Extended messages refer in particular to the column HRESULT in ADS Return Codes [▶ 323].

Parameter

Name	Type	Description
ipOldObj	ITComObject*	Reference to the existing object to be exchanged.
pOldInfo	TmcInstData*	Reference to information of the existing object.

Via ipOldObj, the data of the existing object is made available for transfer so that it can be applied.

For example:

```
ULONG nData = sizeof(m_Parameter);
PVOID pData = &m_Parameter;
ipOldObj->TcGetObjPara(PID_Module1Parameter, nData, pData);
```

12.8.9.2 Method ITComOnlineChange:PerformOnlineChange

This method is called by TwinCAT to perform the Online Change.

It is called blocking. It should therefore only take a short time.

Syntax

```
virtual HRESULT TCOMAPI PerformOnlineChange(ITComObject* ipOldObj, TmcInstData* pOldInfo) = 0;
```

Return value

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [▶ 161]. Extended messages refer in particular to the column HRESULT in ADS Return Codes [▶ 323].

Parameter

Name	Type	Description
ipOldObj	ITComObject*	Reference to the existing object to be exchanged.
pOldInfo	TmcInstData*	Reference to information of the existing object.

Via ipOldObj, the data of the existing object is made available for transfer so that it can be applied.

For example:

```
ULONG nData = sizeof(m_Parameter);
PVOID pData = &m_Parameter;
ipOldObj->TcGetObjPara(PID_Module1Parameter, nData, pData);
```

12.8.10 ITComObject interface

The ITComObject interface is implemented by every TwinCAT module. It makes basic functionalities available.

Syntax

```
TCOM_DECL_INTERFACE("00000012-0000-0000-e000-000000000064", ITComObject)
struct __declspec(novtable) ITComObject: public ITcUnknown
```

Methods

Name	Description
TcGetObjectId(OTCID& objId) [▶ 177]	Saves the object ID using the given OTCID reference.
TcSetObjectId [▶ 178]	Sets the object ID of the object to the given OTCID.
TcGetObjectName [▶ 178]	Saves the object names in the buffer with the given length.
TcSetObjectName [▶ 178]	Sets the object name of the object to given CHAR*.
TcSetObjState [▶ 179]	Initializes a transition to a predefined state.
TcGetObjState [▶ 179]	Queries the current state of the object.
TcGetObjPara [▶ 179]	Queries an object parameter identified with its PTCID.
TcSetObjPara [▶ 180]	Sets an object parameter identified with its PTCID.
TcGetParentObjId [▶ 180]	Saves the parent object ID with the help of the given OTCID reference.
TcSetParentObjId [▶ 180]	Sets the parent object ID to the given OTCID.

The ITComObject interface is implemented by every TwinCAT module. It makes functionalities available regarding the state machine and information from/to the TwinCAT system.

12.8.10.1 ITcComObject:TcGetObjectId method

The method saves the object ID with the help of the given OTCID reference.

Syntax

```
HRESULT TcGetObjectId( OTCID& objId )
```

Return value

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [▶ 161]. Extended messages refer in particular to the column HRESULT in ADS Return Codes [▶ 323].

Parameter

Name	Type	Description
objId	OTCID&	Reference to OTCID value.

12.8.10.2 ITcComObject:TcSetObjectId method

The TcSetObjectId method sets the object ID of the object to the given OTCID.

Syntax

```
HRESULT TcSetObjectId( OTCID objId )
```

Return value

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [▶ 161]. Extended messages refer in particular to the column HRESULT in ADS Return Codes [▶ 323].

At present, the return value is ignored by the TwinCAT tasks.

Parameter

Name	Type	Description
objId	OTCID	The OTCID to be set - indicates the success of the ID change.

12.8.10.3 ITcComObject:TcGetObjectName method

The TcGetObjectName method saves the object names in the buffer with the given length.

Syntax

```
HRESULT TcGetObjectName( CHAR* objName, ULONG nameLen );
```

Return value

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [▶ 161]. Extended messages refer in particular to the column HRESULT in ADS Return Codes [▶ 323].

Parameter

Name	Type	Description
objName	CHAR*	The name to be set.
nameLen	ULONG	The maximum length to be written.

12.8.10.4 ITcComObject:TcSetObjectName method

The TcSetObjectName method sets the object name of the object to the given CHAR*.

Syntax

```
HRESULT TcSetObjectName( CHAR* objName )
```

Return value

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [▶ 161]. Extended messages refer in particular to the column HRESULT in ADS Return Codes [▶ 323].

Parameter

Name	Type	Description
objName	CHAR*	The name to be set of the object.

12.8.10.5 ITcComObject:TcSetObjState method

The TcSetObjState method initializes a transition to the given state.

Syntax

```
HRESULT TcSetObjState(TCOM_STATE state, ITComObjectServer* ipSrv, PTComInitDataHdr pInitData);
```

Return value

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [▶ 161]. Extended messages refer in particular to the column HRESULT in ADS Return Codes [▶ 323].

Parameter

Name	Type	Description
state	TCOM_STATE	Displays the new state.
ipSrv	ITComObjectServer*	ObjServer that handles the object.
pInitData	PTComInitDataHdr	Points to a list of parameters (optional); see the IMPLEMENT_ITCOMOBJECT_EVALUATE_INITDATA macro as an example of how the list can be iterated.

12.8.10.6 ITcComObject:TcGetObjState method

The TcGetObjState method queries the current state of the object.

Syntax

```
HRESULT TcGetObjState(TCOM_STATE* pState)
```

Return value

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [▶ 161]. Extended messages refer in particular to the column HRESULT in ADS Return Codes [▶ 323].

Parameter

Name	Type	Description
pState	TCOM_STATE*	Pointer to the state.

12.8.10.7 ITcComObject:TcGetObjPara method

The TcGetObjPara method queries an object parameter identified by means of its PTCID.

Syntax

```
HRESULT TcGetObjPara(PTCID pid, ULONG& nData, PVOID& pData, PTCGP pgp=0)
```

Return value

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [▶ 161]. Extended messages refer in particular to the column HRESULT in ADS Return Codes [▶ 323].

Parameter

Name	Type	Description
pid	PTCID	Parameter-ID of the object parameter.
nData	ULONG&	Max. length of data.
pData	PVOID&	Pointer to the data.
pgp	PTCGP	Reserved for future expansion, pass NULL.

12.8.10.8 ITcComObject:TcSetObjPara method

The TcSetObjPara method sets an object parameter identified by means of its PTCID.

Syntax

```
HRESULT TcSetObjPara(PTCID pid, ULONG nData, PVOID pData, PTCGP pgp=0)
```

Return value

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [▶ 161]. Extended messages refer in particular to the column HRESULT in ADS Return Codes [▶ 323].

Parameter

Name	Type	Description
pid	PTCID	Parameter-ID of the object parameter.
nData	ULONG	Max. length of data.
pData	PVOID	Pointer to the data.
pgp	PTCGP	Reserved for future expansion, pass NULL.

12.8.10.9 ITcComObject:TcGetParentObjId method

The TcGetParentObjId method saves the parent object ID with the help of the given OTCID reference.

Syntax

```
HRESULT TcGetParentObjId( OTCID& objId )
```

Return value

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [▶ 161]. Extended messages refer in particular to the column HRESULT in ADS Return Codes [▶ 323].

Parameter

Name	Type	Description
objId	OTCID&	Reference to OTCID value.

12.8.10.10 ITcComObject:TcSetParentObjId method

The TcSetParentObjId method sets the parent object ID with the help of the given OTCID reference.

Syntax

```
HRESULT TcSetParentObjId( OTCID objId )
```

Return value

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [▶ 161]. Extended messages refer in particular to the column HRESULT in ADS Return Codes [▶ 323].

At present, the return value is ignored by the TwinCAT tasks.

Parameter

Name	Type	Description
objId	OTCID	Reference to OTCID value.

12.8.11 ITComObject interface (C++ convenience)

The ITComObject interface is implemented by every TwinCAT module. It makes basic functionalities available.

TwinCAT C++ provides additional functions, which are not directly defined through the interface.

Syntax

Required include: `TcInterfaces.h`

Methods

Name	Description
OTCID TcGetObjectId [▶ 181]	Queries the object ID.
TcTryToReleaseOpState [▶ 181]	Releases resources; must be implemented.

Further methods exist, which are not itemized here.

This functionality is provided as standard by the module wizards.

12.8.11.1 TcGetObjectId method

The method queries the object ID.

Syntax

```
OTCID TcGetObjectId(void)
```

Return value

Name	Description
OTCID	Returns the OTCID of the object.

12.8.11.2 TcTryToReleaseOpState method

The method TcTryToReleaseOpState releases resources, e.g. data pointers, in order to prepare for exiting the OP state.

Syntax

```
BOOL TcTryToReleaseOpState(void)
```

Return value

TRUE or FALSE is returned.

Description

The TcTryToReleaseOpState method must be implemented to remove possible mutual dependencies between module instances.

It is only called if another module holds a reference to this module.

12.8.12 Interface ITcPostCyclic

The interface is implemented by TwinCAT modules that are called once per task cycle following the output update (comparable to the attribute TcCallAfterOutputUpdate of the PLC).

Syntax

```
TCOM_DECL_INTERFACE("03000025-0000-0000-e000-000000000064", ITcPostCyclic)
struct __declspec(novtable) ITcPostCyclic : public ITcUnknown
```

Required include: `TcIoInterfaces.h`

Methods

Name	Description
PostCycleUpdate [▶ 182]	Is called once per task cycle after the output update if the interface has been logged on to a cyclic caller.

The ITcPostCyclic interface is implemented by TwinCAT modules. This interface is passed to the ITcCyclicCaller::AddPostModule() method when a module logs itself on to a task, usually as the last initialization step during the transition from SafeOP to OP. After login, the PostCycleUpdate() method of the module instance is called.

12.8.12.1 Method ITcPostCyclic:PostCyclicUpdate

The PostCyclicUpdate method normally called by a TwinCAT task after the output update, after the interface has been logged in.

Syntax

```
HRESULT TCOMAPI PostCycleUpdate(ITcTask* ipTask, ITcUnknown* ipCaller, ULONG_PTR context)
```

Return value

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [▶ 161]. Extended messages refer in particular to the column HRESULT in ADS Return Codes [▶ 323].

At present, the return value is ignored by the TwinCAT tasks.

Parameter

Name	Type	Description
ipTask	ITcTask	Refers to the current task context.
ipCaller	ITcUnknown	Refers to the calling instance.
Context	ULONG_PTR	Context contains the value that was passed to the method ITcPostCyclicCaller::AddPostModule().

Description

Within a task cycle, the PostCycleUpdate() method is called after OutputUpdate() has been called for all registered module instances. As a result, this method must be used to implement such cyclic processing.

12.8.13 Interface ITcRTimeTask

Query of extended TwinCAT task information.

Syntax

TCOM_DECL_INTERFACE("02000003-0000-0000-e000-000000000064", ITcRTimeTask)
struct __declspec(novtable) ITcRTimeTask : public ITcTask

Required include: `TcRtInterfaces.h`

Methods

Name	Description
GetCpuAccount [▶ 183]	Query of the CPU account of a TwinCAT task.

TwinCAT task information can be queried and used with this interface.

See Sample30: Timing Measurement [▶ 300]

12.8.13.1 Method ITcRTimeTask::GetCpuAccount()

Query of the CPU account of a TwinCAT task.

Syntax

```
virtual HRESULT TCOMAPI GetCpuAccount(PULONG pAccount)=0;
```

Return value

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [▶ 161]. Extended messages refer in particular to the column HRESULT in ADS Return Codes [▶ 323].

E_POINTER if the parameter pAccount = NULL.

Parameter

Name	Type	Description
pAccount	PULONG	[out] TwinCAT Task CPU account is stored in this parameter.

The GetCpuAccount() method can be used to query the current computing time used for the task.

Code snippet showing the use of GetCpuAccount(), e.g. within an ITcCyclic::CycleUpdate() method:

```
// CPU account in 100 ns interval
ITcRTimeTaskPtr spRTimeTask = ipTask;
ULONG nCpuAccountForComputeSomething = 0;
if (spRTimeTask != NULL)
{
ULONG nStart = 0;
hr = FAILED(hr) ? hr : spRTimeTask->GetCpuAccount(&nStart);

ComputeSomething();

ULONG nStop = 0;
hr = FAILED(hr) ? hr : spRTimeTask->GetCpuAccount(&nStop);

nCpuAccountForComputeSomething = nStop - nStart;
}
```

12.8.14 Interface ITcTask

Query of the timestamp and task-specific information of a TwinCAT task.

Syntax

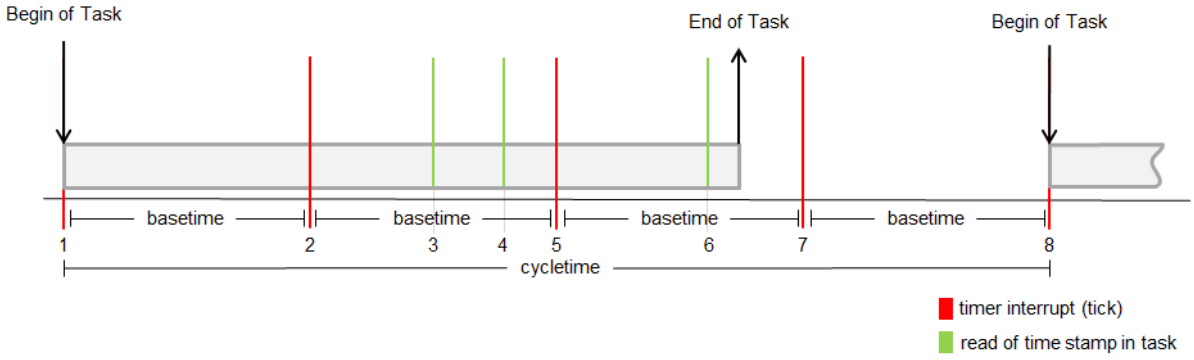
```
TCOM_DECL_INTERFACE("02000002-0000-0000-e000-000000000064", ITcTask)
struct __declspec(novtable) ITcTask : public ITcUnknown
```

Required include: `TcRtInterfaces.h`

Methods

Name	Description
GetCycleCounter [▶ 186]	Querying the number of task cycles since task start.
GetCycleTime [▶ 186]	Querying the task cycle time in nanoseconds, i.e. time between "begin of task" and next "begin of task".
GetPriority [▶ 184]	Querying the task priority.
GetCurrentSysTime [▶ 185]	Querying the time when the task cycle starts at intervals of 100 nanoseconds since January 1, 1601 (UTC).
GetCurrentDcTime [▶ 185]	Querying the distributed clock time when the task cycle starts in nanoseconds since January 1, 2000.
GetCurPentiumTime [▶ 186]	Querying the time when the method is called at intervals of 100 nanoseconds since January 1, 1601 (UTC).

With the ITcTask interface the time can be measured in real-time context.



Reference	Function (ITcTask)	Unit	Zerotime	read time stamps at		
				3	4	6
Distributed Clock master (EtherCAT, Sercos,...)	GetCurrentSysTime	100ns	01.01.1601	1	1	1
	GetCurrentDcTime	1ns	01.01.2000	1	1	1
Processor Clock	GetCurPentiumTime	100ns	01.01.1601	3	4	6

12.8.14.1 Method ITcTask:GetPriority

Querying the task priority.

Syntax

```
virtual HRESULT TCOMAPI GetPriority(PULONG
pPriority)=0;
```

Return value

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [▶ 161]. Extended messages refer in particular to the column HRESULT in ADS Return Codes [▶ 323].

E_POINTER if the parameter pPriority = NULL.

Parameter

Name	Type	Description
pPriority	PULONG	[out] Priority value of the task is stored in this parameter.

Sample30: Timing Measurement [▶ 300] illustrates the use of this method.

12.8.14.2 Method ITcTask:GetCurrentSysTime

Querying the time when the task cycle starts at intervals of 100 nanoseconds since January 1, 1601 (UTC).

Syntax

```
virtual HRESULT TCOMAPI GetCurrentSysTime(PLONGLONG
pSysTime)=0;
```

Return value

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [▶ 161]. Extended messages refer in particular to the column HRESULT in ADS Return Codes [▶ 323].

E_POINTER if the parameter pSysTime = NULL.

Parameter

Name	Type	Description
pSysTime	PLONGLONG	[out] This parameter stores the current system time at the beginning of the task cycle.

Sample30: Timing Measurement [▶ 300] illustrates the use of this method.

12.8.14.3 Method ITcTask:GetCurrentDcTime

Querying the distributed clock time when the task cycle starts in nanoseconds since January 1, 2000.

Syntax

```
virtual HRESULT TCOMAPI GetCurrentDcTime(PLONGLONG
pDcTime)=0;
```

Return value

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [▶ 161]. Extended messages refer in particular to the column HRESULT in ADS Return Codes [▶ 323].

E_POINTER if the parameter pDcTime = NULL.

Parameter

Name	Type	Description
pDcTime	PLONGLONG	[out] This parameter stores the distributed clock time at the beginning of the task cycle.

Sample30: Timing Measurement [▶ 300] illustrates the use of this method.

12.8.14.4 Method ITcTask:GetCurPentiumTime

Querying the time when the method is called at intervals of 100 nanoseconds since January 1, 1601 (UTC).

Syntax

```
virtual HRESULT TCOMAPI GetCurPentiumTime(PLONGLONG
pCurTime)=0;
```

Return value

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [▶ 161]. Extended messages refer in particular to the column HRESULT in ADS Return Codes [▶ 323].

E_POINTER if the parameter pCurTime = NULL.

Parameter

Name	Type	Description
pCurTime	PLONGLONG	[out] This parameter stores the current time (UTC) in 100 nanosecond intervals since January 1, 1601.

Sample30: Timing Measurement [▶ 300] illustrates the use of this method.

12.8.14.5 Method ITcTask:GetCycleCounter

Querying the number of task cycles since task start.

Syntax

```
virtual HRESULT TCOMAPI GetCycleCounter(PULONGLONG
pCnt)=0;
```

Return value

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [▶ 161]. Extended messages refer in particular to the column HRESULT in ADS Return Codes [▶ 323].

E_POINTER if the parameter pCnt = NULL.

Parameter

Name	Type	Description
pCnt	PULONGLONG	[out] The number of task cycles since the task was started is stored in this parameter.

Sample30: Timing Measurement [▶ 300] illustrates the use of this method.

12.8.14.6 Method ITcTask:GetCycleTime

Querying the task cycle time in nanoseconds, i.e. time between "begin of task" and next "begin of task".

Syntax

```
virtual HRESULT TCOMAPI GetCycleTime(PULONG
pCycleTimeNS)=0;
```

Return value

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [▶ 161]. Extended messages refer in particular to the column HRESULT in ADS Return Codes [▶ 323].

E_POINTER if the parameter pCnt = NULL.

Parameter

Name	Type	Description
pCycleTimeNS	PULONG	[out] The configured task cycle time is stored in nanoseconds in this parameter.

Sample30: Timing Measurement [▶ 300] illustrates the use of this method.

12.8.15 Interface ITcTaskNotification

Executes a callback if the cycle time was exceeded during the previous cycle.
This interface provides comparable functions such as PLC PlcTaskSystemInfo->CycleTimeExceeded.

Syntax

```
TCOM_DECL_INTERFACE("9CDE7C78-32A0-4375-827E-924B31021FCD", ITcTaskNotification) struct
__declspec(novtable) ITcTaskNotification: public ITcUnknown
```

Required include: `TcRtInterfaces.h`

Methods

Name	Description
NotifyCycleTimeExceeded [▶ 187]	Called if the cycle time was exceeded.

Note that the callback does not take place during the calculations, but at the end of the cycle. Therefore, this method does not offer any mechanism for immediately stopping the calculations.

12.8.15.1 Method ITcTaskNotification::NotifyCycleTimeExceeded()

Called when the cycle time has previously expired, i.e., not immediately upon timeout, but afterwards.

Syntax

```
virtual HRESULT TCOMAPI NotifyCycleTimeExceeded ();
```

Return value

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [▶ 161]. Extended messages refer in particular to the column HRESULT in ADS Return Codes [▶ 323].

Parameter

Name	Type	Description
ipTask	ITcTask	Refers to the current task context.
context	ULONG_PTR	Context

12.8.16 Interface ITcUnknown

ITcUnknown defines the reference count as well as the querying of a reference to a more specific interface.

Syntax

```
TCOM_DECL_INTERFACE("00000001-0000-0000-e000-000000000064", ITcUnknown)
```

Declared in: TcInterfaces.h

Required include: -

Methods

Name	Description
TcAddRef [▶ 188]	Increments the reference counter.
TcQueryInterface [▶ 188]	Query of the reference to an implemented interface via the IID.
TcRelease [▶ 189]	Decrements the reference counter.

Every TcCOM interface is derived directly or indirectly from ITcUnknown. As a result, every TcCOM module class implements ITcUnknown because it is derived from ITComObject.

The standard implementation of ITcUnknown ensures that the object is deleted after the last reference is released. For this reason, an interface pointer must not be dereferenced after calling TcRelease().

12.8.16.1 Method ITcUnknown:TcAddRef

Increments the reference counter and returns the new value.

Syntax

```
ULONG TcAddRef( )
```

Return value

Resulting reference count value.

12.8.16.2 Method ITcUnknown:TcQueryInterface

Query of an interface pointer with regard to an interface that is given by interface ID (IID).

Syntax

```
HRESULT TcQueryInterface(RITCID iid, PPVOID pipItf )
```

Return value

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [▶ 161]. Extended messages refer in particular to the column HRESULT in ADS Return Codes [▶ 323].

If the demanded interface is not available, the method returns ADSERR_DEVICE_NOINTERFACE.

Parameter

Name	Type	Description
iid	RITCID	Interface IID.
pipItf	PPVOID	Pointer to interface pointer. Is set when the requested interface type is available from the corresponding instance.

Description

Query of the reference to an implemented interface via the IID. It is recommended to use Smart Pointers to initialize and hold interface pointers.

Variant 1:

```
HRESULT GetTraceLevel(ITcUnkown* ip, TcTraceLevel& tl)
{
HRESULT hr = S_OK;
if (ip != NULL)
{
ITComObjectPtr spObj;
hr = ip->TcQueryInterface(spObj.GetIID(), &spObj);
if (SUCCEEDED(hr))
{
hr = spObj->TcGetObjPara(PID_TcTraceLevel, &tl, sizeof(tl));
}
return hr;
}
}
```

The interface ID associated with the Smart Pointer can be used as a parameter in TcQueryInterface. The "&" operator will return the pointer to the internal interface pointer member variable of the Smart Pointer. Variant 1 assumes that the interface pointer is initialized when TcQueryInterface indicates success. If the range remains, then the destructor of the Smart Pointer spObj releases the reference.

Variant 2:

```
HRESULT GetTraceLevel(ITcUnkown* ip, TcTraceLevel& tl)
{
HRESULT hr = S_OK;
ITComObjectPtr spObj = ip;
if (spObj != NULL)
{
spObj->TcGetObjParam(PID_TcTraceLevel, &tl);
}
else
{
hr = ADS_E_NOINTERFACE;
}
return hr;
}
```

If the interface pointer ip is assigned to the Smart Pointer spObj, then the TcQueryInterface method is called implicitly with IID_ITComObject on the instance to which ip refers. This results in a shorter code, but the original return code of TcQueryInterface is lost.

12.8.16.3 Method ITcUnknown:TcRelease

This method decrements the reference counter.

Syntax

```
ULONG TcRelease( )
```

Return value

Resulting reference count value.

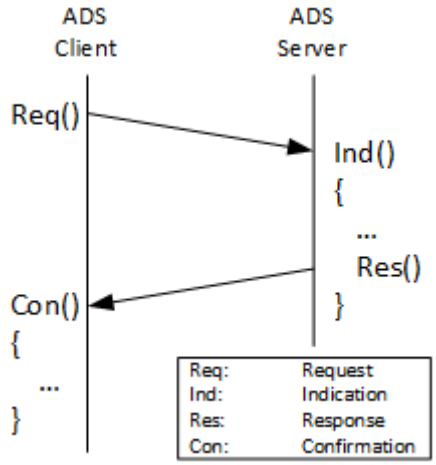
Description

Decrements the reference counter and returns the new value.

If the reference counter is 0, the object deletes itself.

12.9 ADS Communication

ADS based on client-server principle. An ADS query calls the corresponding indication methods on the server side. The ADS response calls the corresponding confirmation method on the client side.



ADS client-server sequence

In this section both the outgoing and incoming ADS communication is described for TwinCAT 3 C++ modules.

ADS command set	Description
AdsReadDeviceInfo [▶ 190]	The general device information can be read with this command.
AdsRead [▶ 192]	ADS read command for retrieving data from an ADS device.
AdsWrite [▶ 194]	ADS write command for transferring data to an ADS device.
AdsReadState [▶ 198]	ADS command to query the state of an ADS device.
AdsWriteControl [▶ 200]	ADS control command to change the state of an ADS device.
AdsAddDeviceNotification [▶ 201]	Observe variable. The client is informed in case of an event.
AdsDelDeviceNotification [▶ 203]	Removes the variable that was previously linked.
AdsDeviceNotification [▶ 205]	Used to transfer the device notification event.
AdsReadWrite [▶ 196]	ADS read/write command. Data is transmitted to an ADS device (write) and its response data read with one call.

The ADS Return Codes [▶ 323] apply to the entire ADS communication.

To get started, take a look at Sample07: Receiving ADS Notifications [▶ 252].

12.9.1 AdsReadDeviceInfo

12.9.1.1 AdsReadDeviceInfoReq

The method AdsDeviceInfoReq enables the transfer of an ADS DeviceInfo command for reading the identification and version number of an ADS server.
AdsReadDeviceInfoCon is called on receipt of the response.

Syntax

```
int AdsReadDeviceInfoReq( AmsAddr& rAddr, ULONG invokeId );
```

Return value

Type: int

Error code - see AdsStatuscodes [▶ 323].

Parameter

Name	Type	Description
rAddr	AmsAddr&	[in] Structure with NetId and port number of the ADS server.
invokeId	ULONG	[in] Handle of the command, which was sent. The InvokeId is specified by the source device and is used for the identification of the commands.

12.9.1.2 AdsReadDeviceInfoInd

The AdsDeviceInfoInd method refers to an ADS DeviceInfo command for reading the identification and version number of an ADS server. AdsReadDeviceInfoRes [▶ 191] must then be called.

Syntax

```
void AdsReadDeviceInfoInd( AmsAddr& rAddr, ULONG invokeId );
```

Return value

void

Parameter

Name	Type	Description
rAddr	AmsAddr&	[in] Structure with NetId and port number of the ADS server.
invokeId	ULONG	[in] Handle of the command, which was sent. The InvokeId is specified by the source device and is used for the identification of the commands.

12.9.1.3 AdsReadDeviceInfoRes

The method AdsReadDeviceInfoRes sends an ADS Read Device Info. AdsReadDeviceInfoCon [▶ 192] forms the counterpart and is subsequently called.

Syntax

```
int AdsReadDeviceInfoRes( AmsAddr& rAddr, ULONG invokeId, ULONG nResult, CHAR
name[ADS_FIXEDNAMESIZE], AdsVersion version );
```

Return value

Type: int

Error code - see AdsStatuscodes [▶ 323].

Parameter

Name	Type	Description
rAddr	AmsAddr&	[in] Structure with NetId and port number of the responding ADS server.
invokeId	ULONG	[in] Handle of the command, which was sent. The InvokeId is specified by the source device and is used for the identification of the commands.
nResult	ULONG	[in] contains the result of the ADS write command; see AdsStatuscodes [▶ 323].
name	char[ADS_FIXEDNAMESIZE	[in] contains the name of the device.
version	AdsVersion	[in] structure of build (int), revision (byte) and version (byte) of the device.

12.9.1.4 AdsReadDeviceInfoCon

The AdsReadDeviceInfoCon method enables the receipt of an ADS read confirmation of device information. The receiving module must provide this method. The counterpart AdsReadDeviceInfoReq [▶ 190]must be called beforehand.

Syntax

```
void AdsReadDeviceInfoCon( AmsAddr& rAddr, ULONG invokeId, ULONG nResult,
CHAR name[ADS_FIXEDNAMESIZE], AdsVersion version );
```

Return value

void

Parameter

Name	Type	Description
rAddr	AmsAddr&	[in] Structure with NetId and port number of the responding ADS server.
invokeId	ULONG	[in] Handle of the command, which was sent. The InvokeId is specified by the source device and is used for the identification of the commands.
nResult	ULONG	[in] contains the result of the ADS write command; see AdsStatuscodes [▶ 323].
name	char[ADS_FIXEDNAMESIZE]	[in] contains the name of the device.
version	AdsVersion	[in] structure of build (int), revision (byte) and version (byte) of the device.

12.9.2 AdsRead

12.9.2.1 AdsReadReq

The AdsReadReq method enables the sending of an ADS read command for the data transmission from an ADS device.
AdsReadCon [▶ 194] is called on receipt of the response.

Syntax

```
int AdsReadReq( AmsAddr& rAddr, ULONG invokeId, ULONG indexGroup, ULONG indexOffset, ULONG
cbLength );
```

Return value

Type: int

Error code - see AdsStatuscodes [▶ 323].

Parameter

Name	Type	Description
rAddr	AmsAddr&	[in] Structure with NetId and port number of the ADS server.
invokeId	ULONG	[in] Handle of the command, which was sent. The InvokeId is specified by the source device and is used for the identification of the commands.
indexGroup	ULONG	[in] Contains the index group number (32-bit, unsigned) of the requested ADS service.
indexOffset	ULONG	[in] Contains the index offset number (32-bit, unsigned) of the requested ADS service.

Name	Type	Description
cbLength	ULONG	[in] contains the length in bytes of the data to be read (pData).

12.9.2.2 AdsReadInd

The AdsReadInd method enables the receipt of an ADS read request. The AdsReadRes [▶ 193] must be called to send the result.

Syntax

```
void AdsReadInd( AmsAddr& rAddr, ULONG invokeId, ULONG indexGroup, ULONG indexOffset, ULONG
cbLength );
```

Return value

Type: int

ADS Return Code - see AdsStatuscodes [▶ 323].

Parameter

Name	Type	Description
rAddr	AmsAddr&	[in] Structure with NetId and port number of the responding ADS server.
invokeId	ULONG	[in] Handle of the command, which was sent. The InvokeId is specified by the source device and is used for the identification of the commands.
indexGroup	ULONG	[in] Contains the index group number (32-bit, unsigned) of the requested ADS service.
indexOffset	ULONG	[in] Contains the index offset number (32-bit, unsigned) of the requested ADS service.
cbLength	ULONG	[in] contains the length in bytes of the data to be read (pData).

12.9.2.3 AdsReadRes

The method AdsReadRes enables the sending of an ADS read response. AdsReadCon [▶ 194] forms the counterpart and is subsequently called.

Syntax

```
int AdsReadRes( AmsAddr& rAddr, ULONG invokeId, ULONG nResult, ULONG cbLength, PVOID pData );
```

Return value

Type: int

ADS Return Code - see AdsStatuscodes [▶ 323].

Parameter

Name	Type	Description
rAddr	AmsAddr&	[in] Structure with NetId and port number of the responding ADS server.
invokeId	ULONG	[in] Handle of the command, which was sent. The InvokeId is specified by the source device and is used for the identification of the commands.
nResult	ULONG	[in] contains the result of the ADS read command; see AdsStatuscodes [▶ 323].
cbLength	ULONG	[in] contains the length in bytes of the data read (pData).
pData	PVOID	[in] pointer to the data buffer in which the data are located.

12.9.2.4 AdsReadCon

The method AdsReadCon enables the receipt of an ADS read confirmation. The receiving module must provide this method.
The counterpart is AdsReadReq [▶ 192] and must have been called beforehand.

Syntax

```
void AdsReadCon( AmsAddr& rAddr, ULONG invokeId, ULONG nResult, ULONG cbLength, PVOID pData );
```

Return value

void

Parameter

Name	Type	Description
rAddr	AmsAddr&	[in] Structure with NetId and port number of the responding ADS server.
invokeId	ULONG	[in] Handle of the command, which was sent. The InvokeId is specified by the source device and is used for the identification of the commands.
nResult	ULONG	[in] contains the result of the ADS read command; see AdsStatuscodes [▶ 323].
cbLength	ULONG	[in] contains the length in bytes of the data read (pData).
pData	PVOID	[in] pointer to the data buffer in which the data are located.

12.9.3 AdsWrite

12.9.3.1 AdsWriteReq

The method AdsWriteReq enables the sending of an ADS write command for transferring data to an ADS device.
AdsWriteCon [▶ 195] is called on receipt of the response.

Syntax

```
int AdsWriteReq( AmsAddr& rAddr, ULONG invokeId, ULONG indexGroup, ULONG indexOffset, ULONG
cbLength, PVOID pData );
```

Return value

Type: int

Error code - see AdsStatuscodes [▶ 323].

Parameter

Name	Type	Description
rAddr	AmsAddr&	[in] Structure with NetId and port number of the ADS server.
invokeId	ULONG	[in] Handle of the command, which was sent. The InvokeId is specified by the source device and is used for the identification of the commands.
indexGroup	ULONG	[in] contains the index group number (32-bit, unsigned) of the requested ADS service.
indexOffset	ULONG	[in] contains the index offset number (32-bit, unsigned) of the requested ADS service.
cbLength	ULONG	[in] contains the length in bytes of the data to be written (pData).
pData	PVOID	[in] pointer to the data buffer in which the written data are located.

12.9.3.2 AdsWriteInd

The AdsWriteInd method specifies an ADS write command to transfer data to an ADS device.
The AdsWriteRes [▶ 195] must be called to confirm the operation.

Syntax

```
void AdsWriteInd( AmsAddr& rAddr, ULONG invokeId, ULONG indexGroup, ULONG indexOffset, ULONG
cbLength, PVOID pData );
```

Return value

void

Error code - see AdsStatuscodes [▶ 323].

Parameter

Name	Type	Description
rAddr	AmsAddr&	[in] Structure with NetId and port number of the ADS server.
invokeId	ULONG	[in] Handle of the command, which was sent. The InvokeId is specified by the source device and is used for the identification of the commands.
indexGroup	ULONG	[in] Contains the index group number (32-bit, unsigned) of the requested ADS service.
indexOffset	ULONG	[in] Contains the index offset number (32-bit, unsigned) of the requested ADS service.
cbLength	ULONG	[in] Contains the length in bytes of the data to be written (pData).
pData	PVOID	[in] pointer to the data buffer in which the written data are located.

12.9.3.3 AdsWriteRes

The AdsWriteRes method sends an ADS write response. AdsWriteCon [▶ 195] forms the counterpart and is subsequently called.

Syntax

```
int AdsWriteRes( AmsAddr& rAddr, ULONG invokeId, ULONG nResult );
```

Return value

Type: int

Parameter

Name	Type	Description
rAddr	AmsAddr&	[in] Structure with NetId and port number of the responding ADS server.
invokeId	ULONG	[in] Handle of the command, which was sent. The InvokeId is specified by the source device and is used for the identification of the commands.
nResult	ULONG	[in] Contains the result of the ADS write command; see AdsStatuscodes [▶ 323].

ADS Return Code - see AdsStatuscodes [▶ 323].

12.9.3.4 AdsWriteCon

The AdsWriteCon method enables the receipt of an ADS write confirmation. The receiving module must provide this method.
AdsWriteReq [▶ 194] is the counterpart and must have been called beforehand.

Syntax

```
void AdsWriteCon( AmsAddr& rAddr, ULONG invokeId, ULONG nResult );
```

Return value

void

Parameter

Name	Type	Description
rAddr	AmsAddr&	[in] Structure with NetId and port number of the responding ADS server.
invokeId	ULONG	[in] Handle of the command, which was sent. The InvokeId is specified by the source device and is used for the identification of the commands.
nResult	ULONG	[in] contains the result of the ADS write command; see AdsStatuscodes [▶ 323].

12.9.4 AdsReadWrite

12.9.4.1 AdsReadWriteReq

The AdsReadWriteReq method enables an ADS read/write command to be sent for data transmission to and from an ADS device. The AdsReadWriteCon [▶ 197] is called when the response is received.

Syntax

```
int AdsReadWriteReq( AmsAddr& rAddr, ULONG invokeId, ULONG indexGroup, ULONG indexOffset, ULONG
cbReadLength, ULONG cbWriteLength, PVOID pData );
```

Return value

Type: int

Error code - see AdsStatuscodes [▶ 323].

Parameter

Name	Type	Description
rAddr	AmsAddr&	[in] Structure with NetId and port number of the ADS server.
invokeId	ULONG	[in] Handle of the command, which was sent. The InvokeId is specified by the source device and is used for the identification of the commands.
indexGroup	ULONG	[in] contains the index group number (32-bit, unsigned) of the requested ADS service.
indexOffset	ULONG	[in] contains the index offset number (32-bit, unsigned) of the requested ADS service.
cbReadLength	ULONG	[in] contains the length in bytes of the data to be read (pData).
cbWriteLength	ULONG	[in] contains the length in bytes of the data to be written (pData).
pData	PVOID	[in] pointer to the data buffer in which the written data are located.

12.9.4.2 AdsReadWriteInd

The AdsReadWriteInd method specifies an ADS read/write command to transfer data to an ADS device. The AdsReadWriteRes [▶ 199] must be called in order to send the result.

Syntax

```
void AdsReadWriteInd( AmsAddr& rAddr, ULONG invokeId, ULONG indexGroup,
ULONG indexOffset, ULONG cbReadLength, ULONG cbWriteLength, PVOID pData );
```

Return value

void

Parameter

Name	Type	Description
rAddr	AmsAddr&	[in] Structure with NetId and port number of the ADS server.
invokeId	ULONG	[in] Handle of the command, which was sent. The InvokeId is specified by the source device and is used for the identification of the commands.
indexGroup	ULONG	[in] contains the index group number (32-bit, unsigned) of the requested ADS service.
indexOffset	ULONG	[in] contains the index offset number (32-bit, unsigned) of the requested ADS service.
cbReadLength	ULONG	[in] contains the length in bytes of the data to be read (pData).
cbWriteLength	ULONG	[in] contains the length in bytes of the data to be written (pData).
pData	PVOID	[in] pointer to the data buffer in which the written data are located.

12.9.4.3 AdsReadWriteRes

The AdsReadWriteRes method enables the receipt of an ADS read/write confirmation. AdsReadWriteCon [▶ 197] forms the counterpart and is subsequently called.

Syntax

```
int AdsReadWriteRes( AmsAddr& rAddr, ULONG invokeId, ULONG nResult, ULONG cbLength, PVOID pData );
```

Return value

Type: int

ADS Return Code - see AdsStatuscodes [▶ 323].

Parameter

Name	Type	Description
rAddr	AmsAddr&	[in] Structure with NetId and port number of the responding ADS server.
invokeId	ULONG	[in] Handle of the command, which was sent. The InvokeId is specified by the source device and is used for the identification of the commands.
nResult	ULONG	[in] contains the result of the ADS write command; see AdsStatuscodes [▶ 323].
cbLength	ULONG	[in] contains the length in bytes of the data read (pData).
pData	PVOID	[in] pointer to the data buffer in which the data are located.

12.9.4.4 AdsReadWriteCon

The AdsReadWriteCon method enables the receipt of an ADS read/write confirmation. The receiving module must provide this method.
AdsReadWriteReq [▶ 196] is the counterpart and must be called beforehand.

Syntax

```
void AdsReadWriteCon( AmsAddr& rAddr, ULONG invokeId, ULONG nResult, ULONG cbLength, PVOID pData );
```

Return value

void

Parameter

Name	Type	Description
rAddr	AmsAddr&	[in] Structure with NetId and port number of the responding ADS server.
invokeId	ULONG	[in] Handle of the command, which was sent. The InvokeId is specified by the source device and is used for the identification of the commands.
nResult	ULONG	[in] contains the result of the ADS write command; see AdsStatuscodes [▶ 323].
cbLength	ULONG	[in] contains the length in bytes of the data read (pData).
pData	PVOID	[in] pointer to the data buffer in which the data are located.

12.9.5 AdsReadState

12.9.5.1 AdsReadStateReq

The AdsReadStateReq method enables the transmission of an ADS state read command for reading the ADS and device state from an ADS server. When the response is received, AdsReadStateCon [▶ 199] is called.

Syntax

```
int AdsReadStateReq(AmsAddr& rAddr, ULONG invokeId);
```

Return value

Type: int

Error code - see AdsStatuscodes [▶ 323].

Parameter

Name	Type	Description
rAddr	AmsAddr	[in] Structure with NetId and port number of the ADS server.
invokeId	ULONG	[in] Handle of the command, which was sent. The InvokeId is specified by the source device and is used for the identification of the commands.

12.9.5.2 AdsReadStateInd

The AdsReadStateInd method displays an ADS state read command for reading the ADS state and the device state from an ADS device. The AdsReadStateRes [▶ 199] must be called to send the result.

Syntax

```
void AdsReadStateInd( AmsAddr& rAddr, ULONG invokeId );
```

Return value

void

Parameter

Name	Type	Description
rAddr	AmsAddr	[in] Structure with NetId and port number of the ADS server.
invokeId	ULONG	[in] Handle of the command, which was sent. The InvokeId is specified by the source device and is used for the identification of the commands.

12.9.5.3 AdsReadStateRes

The method AdsWriteRes enables the sending of an ADS state read response. AdsReadStateCon [▶ 199] forms the counterpart and is subsequently called.

Syntax

```
int AdsReadStateRes( AmsAddr& rAddr, ULONG invokeId, ULONG nResult, USHORT adsState, USHORT
deviceState );
```

Return value

Type: int

Error code - see AdsStatuscodes [▶ 323].

Parameter

Name	Type	Description
rAddr	AmsAddr&	[in] Structure with NetId and port number of the responding ADS server.
invokeId	ULONG	[in] Handle of the command, which was sent. The InvokeId is specified by the source device and is used for the identification of the commands.
nResult	ULONG	[in] contains the result of the ADS write command; see AdsStatuscodes [▶ 323].
adsState	USHORT	[in] contains the ADS state of the device.
deviceState	USHORT	[in] contains the device state of the device.

12.9.5.4 AdsReadStateCon

The AdsWriteCon method enables the receipt of an ADS state read confirmation. The receiving module must provide this method.
AdsReadStateReq [▶ 198] is the counterpart and must be called beforehand.

Syntax

```
void AdsReadStateCon( AmsAddr& rAddr, ULONG invokeId, ULONG nResult, USHORT adsState, USHORT
deviceState );
```

Return value

void

Parameter

Name	Type	Description
rAddr	AmsAddr&	[in] Structure with NetId and port number of the responding ADS server.
invokeId	ULONG	[in] Handle of the command, which was sent. The InvokeId is specified by the source device and is used for the identification of the commands.
nResult	ULONG	[in] contains the result of the ADS write command; see AdsStatuscodes [▶ 323].
adsState	USHORT	[in] contains the ADS state of the device.
deviceState	USHORT	[in] contains the device state of the device.

12.9.6 AdsWriteControl

12.9.6.1 AdsWriteControlReq

The AdsWriteControlReq method can be used to send an ADS write control command to change the ADS and device state of an ADS server. AdsWriteControlCon [▶ 201] is called when the response is received.

Syntax

```
int AdsWriteControlReq( AmsAddr& rAddr, ULONG invokeId, USHORT adsState,
USHORT deviceState, ULONG cbLength, PVOID pData );
```

Return value

Type: int

Error code - see AdsStatuscodes [▶ 323].

Parameter

Name	Type	Description
rAddr	AmsAddr&	[in] Structure with NetId and port number of the ADS server.
invokeId	ULONG	[in] Handle of the command, which was sent. The InvokeId is specified by the source device and is used for the identification of the commands.
adsState	USHORT	[in] new ADS state (see enum nAdsState in Ads.h).
deviceState	USHORT	[in] new device state.
cbLength	ULONG	[in] contains the length in bytes of the data (pData).
pData	PVOID	[in] pointer to the data buffer in which the written data are located.

12.9.6.2 AdsWriteControlInd

The AdsWriteControlInd method can be used to send an ADS write control command to change the ADS and device state of an ADS device. The AdsWriteControlRes [▶ 201] must be called to confirm the process.

Syntax

```
void AdsWriteControlInd( AmsAddr& rAddr, ULONG invokeId, USHORT adsState, USHORT deviceState, ULONG
cbLength, PVOID pDeviceData );
```

Return value

void

Parameter

Name	Type	Description
rAddr	AmsAddr&	[in] Structure with NetId and port number of the ADS server.
invokeId	ULONG	[in] Handle of the command, which was sent. The InvokeId is specified by the source device and is used for the identification of the commands.
adsState	USHORT	[in] contains the index group number (32-bit, unsigned) of the requested ADS service.
deviceState	USHORT	[in] contains the index offset number (32-bit, unsigned) of the requested ADS service.
cbLength	ULONG	[in] contains the length in bytes of the data (pData).
pData	PVOID	[in] pointer to the data buffer in which the written data are located.

12.9.6.3 AdsWriteControlRes

The AdsWriteControlRes method enables the sending of an ADS write control response. AdsWriteControlCon [▶ 201] forms the counterpart and is subsequently called.

Syntax

```
int AdsWriteControlRes( AmsAddr& rAddr, ULONG invokeId, ULONG nResult );
```

Return value

Type: int

ADS Return Code - see AdsStatuscodes [▶ 323].

Parameter

Name	Type	Description
rAddr	AmsAddr&	[in] Structure with NetId and port number of the responding ADS server.
invokeId	ULONG	[in] Handle of the command, which was sent. The InvokeId is specified by the source device and is used for the identification of the commands.
nResult	ULONG	[in] contains the result of the ADS write command; see AdsStatuscodes [▶ 323].

12.9.6.4 AdsWriteControlCon

The AdsWriteCon method enables the receipt of an ADS write control confirmation. The receiving module must provide this method.
AdsWriteControlReq [▶ 200] is the counterpart and must be called beforehand.

Syntax

```
void AdsWriteControlCon( AmsAddr& rAddr, ULONG invokeId, ULONG nResult );
```

Return value

void

Parameter

Name	Type	Description
rAddr	AmsAddr&	[in] Structure with NetId and port number of the responding ADS server.
invokeId	ULONG	[in] Handle of the command, which was sent. The InvokeId is specified by the source device and is used for the identification of the commands.
nResult	ULONG	[in] contains the result of the ADS write command; see AdsStatuscodes [▶ 323].

12.9.7 AdsAddDeviceNotification

12.9.7.1 AdsAddDeviceNotificationReq

The AdsAddDeviceNotificationReq method enables the sending an ADS add device notification command to add a device notification to an ADS device. AdsAddDeviceNotificationCon [▶ 203] is called when the response is received.

Syntax

```
int AdsAddDeviceNotificationReq( AmsAddr& rAddr, ULONG invokeId, ULONG indexGroup, ULONG
indexOffset,
AdsNotificationAttrib noteAttrib);
```

Return value

Type: int

Error code - see AdsStatuscodes [▶ 323].

Parameter

Name	Type	Description
rAddr	AmsAddr&	[in] Structure with NetId and port number of the ADS server.
invokeId	ULONG	[in] Handle of the command, which was sent. The InvokeId is specified by the source device and is used for the identification of the commands.
indexGroup	ULONG	[in] contains the index group number (32-bit, unsigned) of the requested ADS service.
indexOffset	ULONG	[in] contains the index offset number (32-bit, unsigned) of the requested ADS service.
noteAttrib	AdsNotificationAttrib	[in] contains the specification of the notification parameters (cbLength, TransMode, MaxDelay).

12.9.7.2 AdsAddDeviceNotificationInd

The AdsAddDeviceNotificationInd method should enable the sending of AdsDeviceNotification [▶ 205]. The AdsAddDeviceNotificationRes [▶ 202] must be called to confirm the process.

Syntax

```
void AdsAddDeviceNotificationInd( AmsAddr& rAddr, ULONG invokeId, ULONG indexGroup, ULONG
indexOffset, AdsNotificationAttrib noteAttrib );
```

Return value

void

Parameter

Name	Type	Description
rAddr	AmsAddr&	[in] Structure with NetId and port number of the responding ADS server.
invokeId	ULONG	[in] Handle of the command, which was sent. The InvokeId is specified by the source device and is used for the identification of the commands.
indexGroup	ULONG	[in] contains the index group number (32-bit, unsigned) of the requested ADS service.
indexOffset	ULONG	[in] contains the index offset number (32-bit, unsigned) of the requested ADS service.
noteAttrib	AdsNotificationAttrib	[in] contains the specification of the notification parameters (cbLength, TransMode, MaxDelay).

12.9.7.3 AdsAddDeviceNotificationRes

The AdsAddDeviceNotificationRes method enables the sending of an ADS device add notification response. AdsAddDeviceNotificationCon [▶ 203] forms the counterpart and is subsequently called.

Syntax

```
void AdsAddDeviceNotificationCon( AmsAddr& rAddr, ULONG invokeId, ULONG nResult, ULONG handle );
```

Return value

void

Parameter

Name	Type	Description
rAddr	AmsAddr&	[in] Structure with NetId and port number of the responding ADS server.
invokeId	ULONG	[in] Handle of the command, which was sent. The InvokeId is specified by the source device and is used for the identification of the commands.
nResult	ULONG	[in] contains the result of the ADS write command; see AdsStatuscodes.
Handle	ULONG	[in] handle to generated device notification.

12.9.7.4 AdsAddDeviceNotificationCon

The AdsAddDeviceNotificationCon method confirms an ADS device addition notification request. AdsAddDeviceNotificationReq [▶ 201] forms the counterpart and must be called beforehand.

Syntax

```
void AdsAddDeviceNotificationCon( AmsAddr& rAddr, ULONG invokeId, ULONG nResult, ULONG handle );
```

Return value

void

Parameter

Name	Type	Description
rAddr	AmsAddr&	[in] Structure with NetId and port number of the responding ADS server.
invokeId	ULONG	[in] Handle of the command, which was sent. The InvokeId is specified by the source device and is used for the identification of the commands.
nResult	ULONG	[in] contains the result of the ADS write command; see AdsStatuscodes.
Handle	ULONG	[in] handle to generated device notification.

12.9.8 AdsDelDeviceNotification

12.9.8.1 AdsDelDeviceNotificationReq

The AdsDelDeviceNotificationReq method enables the sending of an ADS device delete notification command to remove a device notification from an ADS device. The AdsDelDeviceNotificationCon [▶ 204] is called when the response is received.

Syntax

```
int AdsDelDeviceNotificationReq( AmsAddr& rAddr, ULONG invokeId, ULONG hNotification );
```

Return value

Type: int

Error code - see AdsStatuscodes [▶ 323].

Parameter

Name	Type	Description
rAddr	AmsAddr&	[in] Structure with NetId and port number of the ADS server.

Name	Type	Description
invokeId	ULONG	[in] Handle of the command, which was sent. The InvokeId is specified by the source device and is used for the identification of the commands.
hNotification	ULONG	[in] contains the handle of the notification to be removed.

12.9.8.2 AdsDelDeviceNotificationInd

The AdsAddDeviceNotificationCon method enables the receipt of an ADS device deletion notification confirmation. The receiving module must provide this method. The AdsDelDeviceNotificationRes [▶ 204] must be called to confirm the process.

Syntax

```
void AdsDelDeviceNotificationCon( AmsAddr& rAddr, ULONG invokeId, ULONG nResult );
```

Return value

void

Parameter

Name	Type	Description
rAddr	AmsAddr&	[in] Structure with NetId and port number of the responding ADS server.
invokeId	ULONG	[in] Handle of the command, which was sent. The InvokeId is specified by the source device and is used for the identification of the commands.
nResult	ULONG	[in] contains the result of the ADS write command; see AdsStatuscodes [▶ 323].

12.9.8.3 AdsDelDeviceNotificationRes

The AdsAddDeviceNotificationRes method enables the receipt of an ADS device deletion notification. AdsDelDeviceNotificationCon [▶ 204] forms the counterpart and is subsequently called.

Syntax

```
int AdsDelDeviceNotificationRes( AmsAddr& rAddr, ULONG invokeId, ULONG nResult );
```

Return value

Int

Returns the result of the ADS command, see AdsStatuscodes [▶ 323].

Parameter

Name	Type	Description
rAddr	AmsAddr&	[in] Structure with NetId and port number of the responding ADS server.
invokeId	ULONG	[in] Handle of the command, which was sent. The InvokeId is specified by the source device and is used for the identification of the commands.
nResult	ULONG	[in] contains the result of the ADS command; see AdsStatuscodes [▶ 323].

12.9.8.4 AdsDelDeviceNotificationCon

The AdsAddDeviceNotificationCon method enables the receipt of an ADS device deletion notification confirmation. The receiving module must provide this method. AdsDelDeviceNotificationReq [▶ 203] forms the counterpart and must be called beforehand.

Syntax

```
void AdsDelDeviceNotificationCon( AmsAddr& rAddr, ULONG invokeId, ULONG nResult );
```

Return value

void

Parameter

Name	Type	Description
rAddr	AmsAddr&	[in] Structure with NetId and port number of the responding ADS server.
invokeId	ULONG	[in] handle of the transmitted command; the InvokeId is specified by the source device and serves to identify the commands.
nResult	ULONG	[in] contains the result of the ADS write command; see AdsStatuscodes [▶ 323].

12.9.9 AdsDeviceNotification

12.9.9.1 AdsDeviceNotificationReq

The AdsAddDeviceNotificationReq method enables the sending of an ADS device notification to inform an ADS device. The AdsDeviceNotificationInd [▶ 205] is called on the counterpart.

Syntax

```
int AdsDeviceNotificationReq( AmsAddr& rAddr, ULONG invokeId, ULONG cbLength, AdsNotificationStream
notifications[] );
```

Return value

Type: int

ADS Return Code - see AdsStatuscodes [▶ 323].

Parameter

Name	Type	Description
rAddr	AmsAddr&	[in] Structure with NetId and port number of the ADS server.
invokeId	ULONG	[in] Handle of the command, which was sent. The InvokeId is specified by the source device and is used for the identification of the commands.
nResult	ULONG	[in] contains the result of the device notification display.
notifications[]	AdsNotificationStream	[in] contains information about the device notification(s).

12.9.9.2 AdsDeviceNotificationInd

The AdsDeviceNotificationInd method enables the receipt from an ADS device notification display. The receiving module must provide this method. Receipt is not acknowledged.
AdsDeviceNotificationCon [▶ 206] must be called on the AdsDeviceNotificationReq [▶ 205] side to check the transmission.

Syntax

```
void AdsDeviceNotificationInd( AmsAddr& rAddr, ULONG invokeId, ULONG cbLength,
AdsNotificationStream* pNotifications );
```

Return value

void

Parameter

Name	Type	Description
rAddr	AmsAddr&	[in] Structure with NetId and port number of the responding ADS server.
invokeId	ULONG	[in] Handle of the command, which was sent. The InvokeId is specified by the source device and is used for the identification of the commands.
cbLength	ULONG	[in] contains the length of pNotifications.
pNotifications	AdsNotificationStream*	[in] Pointer to the notifications. This array consists of AdsStampHeader with notification handle and data via AdsNotificationSample.

12.9.9.3 AdsDeviceNotificationCon

The sender can use the AdsAddDeviceNotificationCon method to check the transmission of an ADS device notification.
AdsDeviceNotificationReq [▶ 205] must be called beforehand.

Syntax

```
void AdsDeviceNotificationCon( AmsAddr& rAddr, ULONG invokeId, ULONG nResult );
```

Return value

void

Parameter

Name	Type	Description
rAddr	AmsAddr&	[in] Structure with NetId and port number of the ADS server.
invokeId	ULONG	[in] Handle of the command, which was sent. The InvokeId is specified by the source device and is used for the identification of the commands.
nResult	ULONG	[in] contains the result of the device notification display.

12.10 Mathematical Functions

TwinCAT has implemented its own mathematical functions because the math.h implementation from Microsoft is not real-time capable.
These functions are declared in TcMath.h, a part of TwinCAT SDK. For x64 the operations are executed via SSE; on x86 systems the FPU is used.

TwinCAT 3.1 4018 or earlier

TwinCAT 3.1 4018 provides an fpu87.h with the same methods. This continues to exist and redirects to TcMath.h.

Methods provided

Name	Description
sqr_	Calculates the square.
sqrt_	Calculates the square root.
sin_	Calculates the sine.
cos_	Calculates the cosine.
tan_	Calculates the tangent.
atan_	Calculates the angle whose tangent is the specified value.

Name	Description
atan2_	Calculates the angle whose tangent is the quotient of two specified values.
asin_	Calculates the angle whose sine is the specified value.
acos_	Calculates the angle whose cosine is the specified value.
exp_	Calculates e to the specified power.
log_	Calculates the logarithm of a specified value.
log10_	Calculates the base 10 logarithm of a specified value.
fabs_	Calculates the absolute value.
fmod_	Calculates the remainder.
ceil_	Calculates the smallest integer that is greater than or equal to the specified number.
floor_	Calculates the largest integer that is smaller than or equal to the specified number.
pow_	Calculates a specified number to the specified power.
sincos_	Calculates the sine and cosine of x.
fmodabs_	Calculates the absolute value that meets the Euclidean definition of the mod operation.
round_	Calculates a value and rounds to the nearest integer.
round_digits_	Calculates a rounded value with a specified number of decimal places.
coubic_	Calculates the cubic value.
ldexp_	Calculates a real number (double) from mantissa and exponent.
ldexpf_	Calculates a real number (float) from mantissa and exponent.
sinh_	Calculates the hyperbolic sine of the specified angle.
cosh_	Calculates the hyperbolic cosine of the specified angle.
tanh_	Calculates the hyperbolic tangent of the specified angle.
finite_	Determines whether the specified value is finite.
isnan_	Determines whether the specified value is not a number (NAN).
rands_	Calculates a pseudo random number between 0 and 32767. The parameter holdrand is set randomly and changed with every call.

- The functions have the extension "_" (underscore), which identifies them as TwinCAT implementation.
- Most are analog math.h, designed by Microsoft, only for the data type double.

See also

MSDN documentation of analog math.h functions.

12.11 Time Functions

TwinCAT provides functions for time conversion, they are declared in TcTimeConversion.h, which is part of TwinCAT SDK.

Methods provided

Name	Description
TcDayOfWeek(WORD day, WORD month, WORD year)	Determines the day of the week. Input: day (0..30) and month (1..12) Return: 0 is Sunday, 6 is Saturday
TcIsLeapYear	Determines whether the given year is a leap year.
TcDaysInYear	Determines the number of days in a given year.
TcDaysInMonth	Determines the number of days in a given month.
TcSystemTimeToFileTime(const SYSTEMTIME* lpSystemTime, FILETIME *lpFileTime);	Converts the given system time into a file time.
TcFileTimeToSystemTime(const FILETIME *lpFileTime, SYSTEMTIME* lpSystemTime);	Converts the given file time into a system time.
TcSystemTimeToFileTime(const SYSTEMTIME* lpSystemTime, ULONGLONG& ul64FileTime);	Converts the given system time into a file time (ULONGLONG format).
TcFileTimeToSystemTime(const ULONGLONG& ul64FileTime, SYSTEMTIME* lpSystemTime);	Converts the given file time (ULONGLONG format) into a system time.
TcIsISO8601TimeFormat(PCCH sDT)	Checks whether a PCCH follows the time format ISO8601.
TcDecodeDateTime(PCCH sDT)	Converts a ULONG as DateTime from the PCCH into ISO8601 format.
TcDecodeDcTime(PCCH sDT)	Converts a LONGLONG as DcTime from the PCCH into ISO8601 format.
TcDecodeFileTime(PCCH sFT)	Converts a LONGLONG as FileTime from the PCCH into ISO8601 format.
TcEncodeDateTime(ULONG value, PCHAR p, UINT len)	Converts a string (p, len) into ISO8601 format on the basis of the ULONG value in DateTime format. Minimum length for p is 24 bytes.
TcEncodeDcTime(LONGLONG value, PCHAR p, UINT len)	Converts a string (p, len) into ISO8601 format on the basis of the LONGLONG in DcTime format. Minimum length for p is 32 bytes.
TcEncodeFileTime(LONGLONG value, PCHAR p, UINT len)	Converts a string (p, len) into ISO8601 format on the basis of the LONGLONG in FileTime format. Minimum length for p is 32 bytes.
TcDcTimeToFileTime(LONGLONG dcTime)	Converts a LONGLONG as FileTime from the LONGLONG into DcTime.
TcFileTimeToDcTime(LONGLONG fileTime);	Converts a LONGLONG as DcTime from the LONGLONG into FileTime.
TcDcTimeToDateTime(LONGLONG dcTime)	Converts a ULONG as DateTime from the LONGLONG into DcTime.
TcDateTimeToDcTime(ULONG dateTime)	Converts a ULONG as DcTime from the LONGLONG into DateTime.
TcFileTimeToDateTime(LONGLONG fileTime)	Converts a ULONG as DateTime from the LONGLONG into FileTime.
TcDateTimeToFileTime(ULONG dateTime)	Converts a LONGLONG as FileTime from the ULONG into DateTime.

- Further information on different time sources is described here: https://infosys.beckhoff.com/content/1031/ethercatsystem/2469114379.html

12.12 STL / Containers

TwinCAT 3 C++ supports STL with regard to

- Array
- List
- Map, Unordered_Map
- Set, Unordered_Set
- Stack
- String
- Vector, Inplace_Vector
- String, wstring, u8string, u16string, u32string
- Algorithms (such as binary_search)
  - See *%TWINCAT3DIR%\Sdk\Include\Stl\Stl\algorithm* for a specific list of supported algorithms.
- Further non STL-defined extensions can be found in *%TWINCAT3DIR%\SDK\Include\Stl\StlExtend*

Restrictions

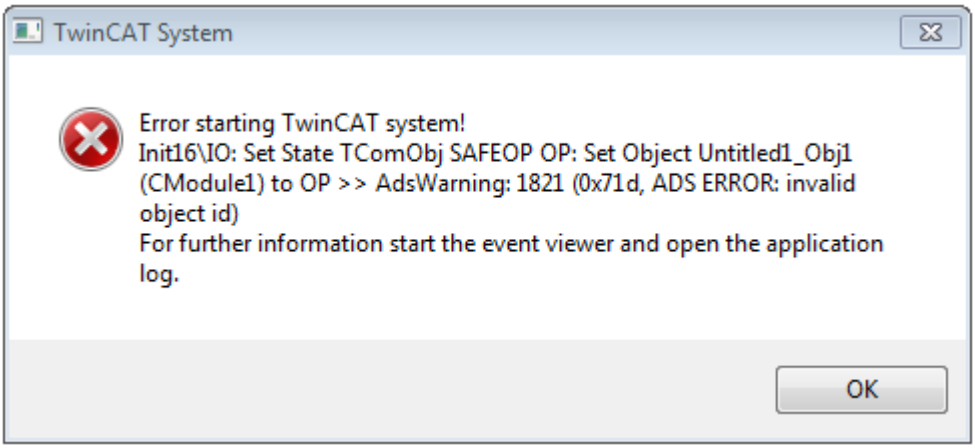
- Class templates do not exist for all data types.
- Some header files should not be used directly.

More detailed documentation on memory management, which uses STL, can be found in chapter Memory allocation [▶ 150].

12.13 Error messages - understanding

In TwinCAT you receive very detailed information about errors that occur.

For instance, this error message means:



- The error occurred during the transition from SAFE OP to OP.
- The affected object is "Untitled1_Obj1" (CModule1).
- The error code 1821 / 0x71d indicates that the object ID is invalid.

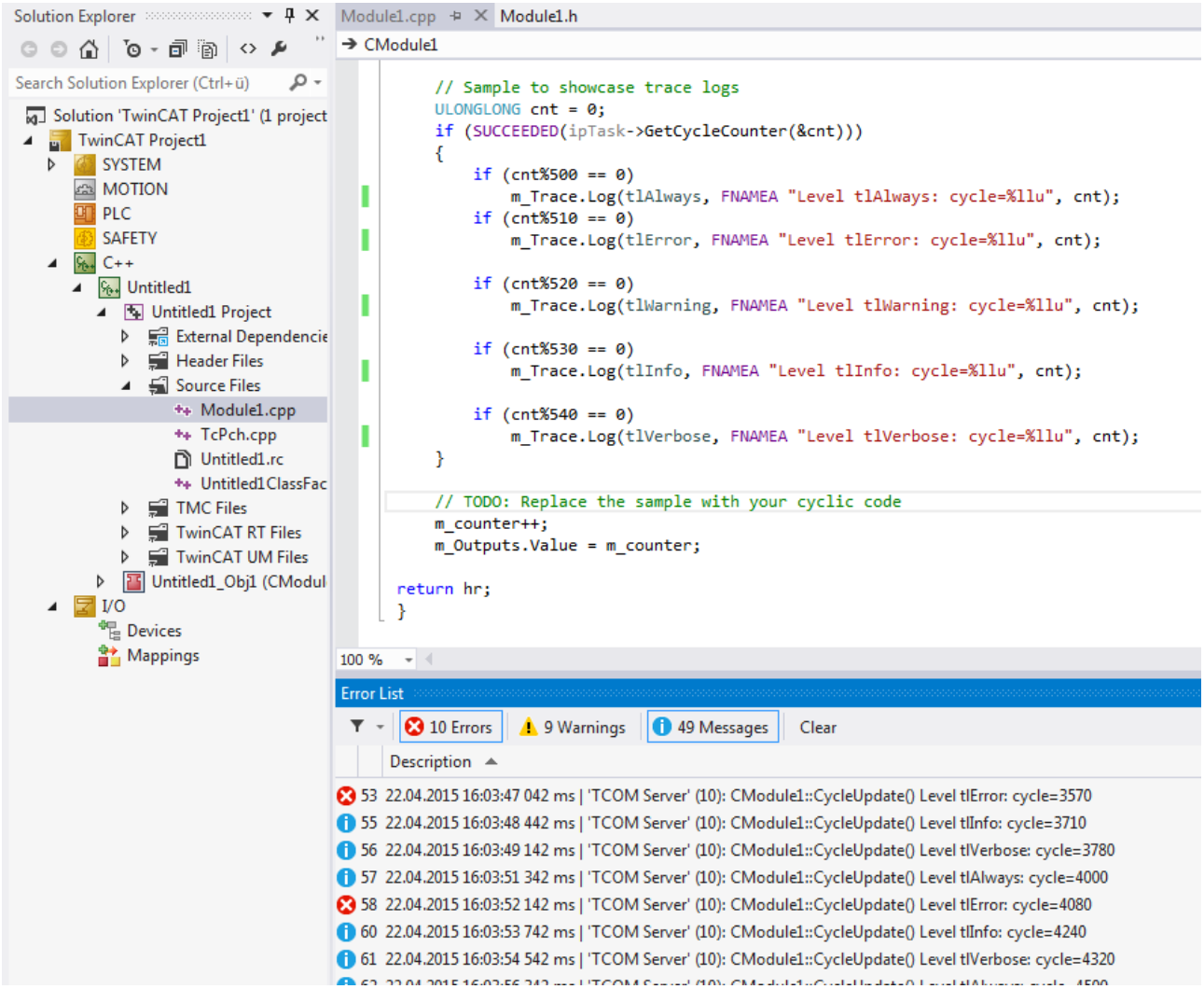
Therefore, you should examine the method "SetObjStateSP()", which is responsible for this transition, in more detail. In the case of the generated standard code, you can see that the addition of the module takes place there.

The reason for this error is that no task has been assigned to this module, so the module cannot have a task in which it is executed.

12.14 Module messages for the Engineering (logging / tracing)

Overview

TwinCAT 3 C++ offers the option of sending messages from a C++ module to the TwinCAT 3 Engineering as tracing or logging.



Syntax

The syntax for recording messages is as follows:

```
m_Trace.Log(TLEVEL, FNMACRO"A message", …);
```

With these properties:

- `TLEVEL` categorizes a message into one of five levels.
  The recording of the higher level always includes the recording of the lower levels: i.e. a message classified at level "tlWarning" will occur with level "tlAlways", "tlError" and "tlWarning" - it will NOT record the "tlInfo" and "tlVerbose" messages.

Level 0	tlAlways
Level 1	tlError
Level 2	tlWarning
Level 3	tlInfo
Level 4	tlVerbose

- `FNMACRO` can be used to place the function name before the message to be printed
  - FENTERA: Used when entering a function; prints the function name followed by ">>>".
  - FNAMEA: Used within a function; prints the function name.
  - FLEAVEA: Used when exiting a function; prints the function name followed by "<<<".
- The `%q` format specifier is used to output pointers and other variables of platform-specific size.

Sample

```
HRESULT CModule1::CycleUpdate(ITcTask* ipTask, ITcUnknown* ipCaller, ULONG_PTR context)
{
    HRESULT hr = S_OK;

    // Sample to showcase trace logs
    ULONGLONG cnt = 0;
    if (SUCCEEDED(ipTask->GetCycleCounter(&cnt)))
    {
        if (cnt%500 == 0)
            m_Trace.Log(tlAlways, FENTERA "Level tlAlways: cycle= %llu", cnt);

        if (cnt%510 == 0)
            m_Trace.Log(tlError, FENTERA "Level tlError: cycle=%llu", cnt);

        if (cnt%520 == 0)
            m_Trace.Log(tlWarning, FENTERA "Level tlWarning: cycle=%lld", cnt);

        if (cnt%530 == 0)
            m_Trace.Log(tlInfo, FENTERA "Level tlInfo: cycle=%llu", cnt);

        if (cnt%540 == 0)
            m_Trace.Log(tlVerbose, FENTERA "Level tlVerbose: cycle=%llu", cnt);
    }

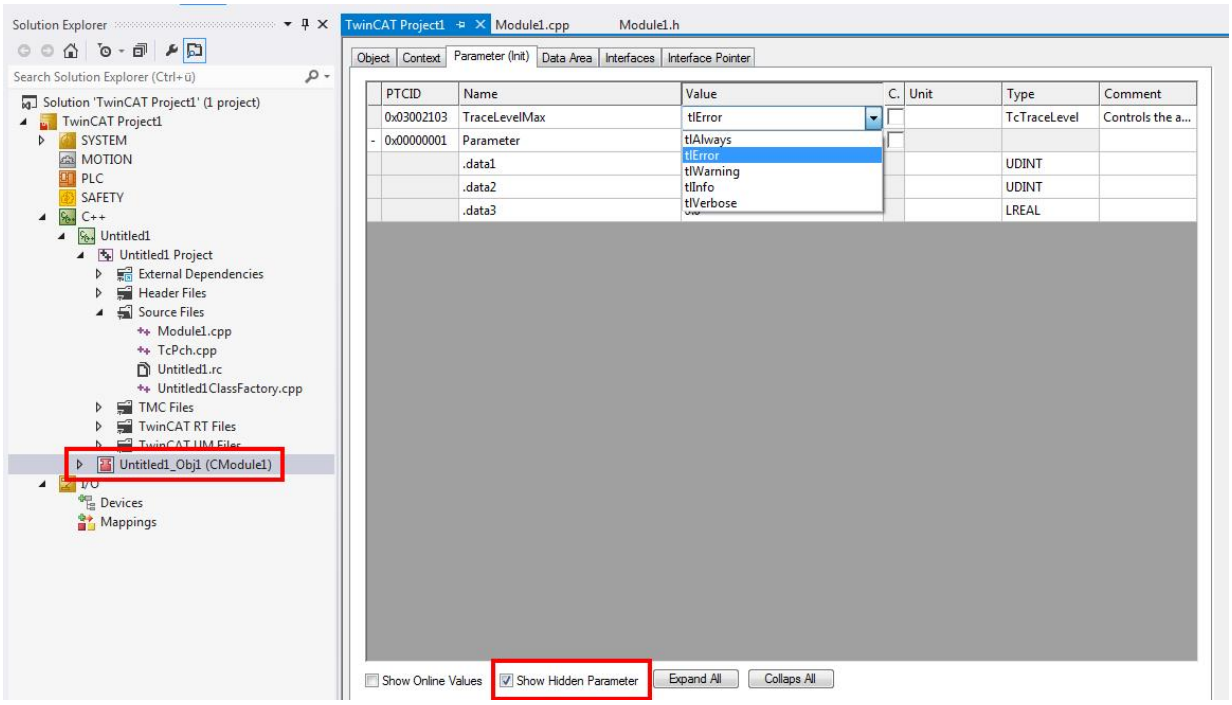
    // TODO: Replace the sample with your cyclic code
    m_counter++;
    m_Outputs.Value = m_counter;

    return hr;
}
```

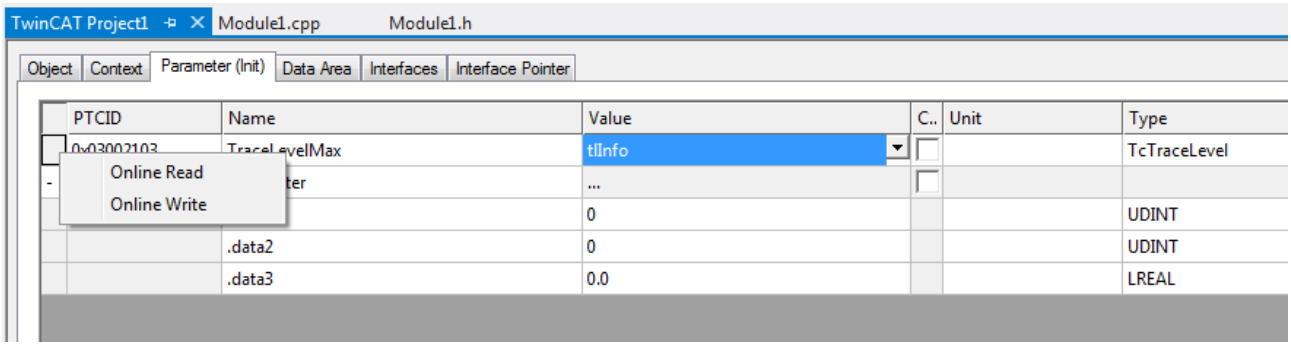
Use tracking level

The tracking level can be preconfigured at the level of the module instance.

1. Navigate to the instance of the module in the solution tree.
2. Select the **Parameter (Init)** tab on the right.
3. Make sure that you activate **Show Hidden Parameter**.
4. Select the tracking level.
5. To test everything, select the highest level **tlVerbose**.



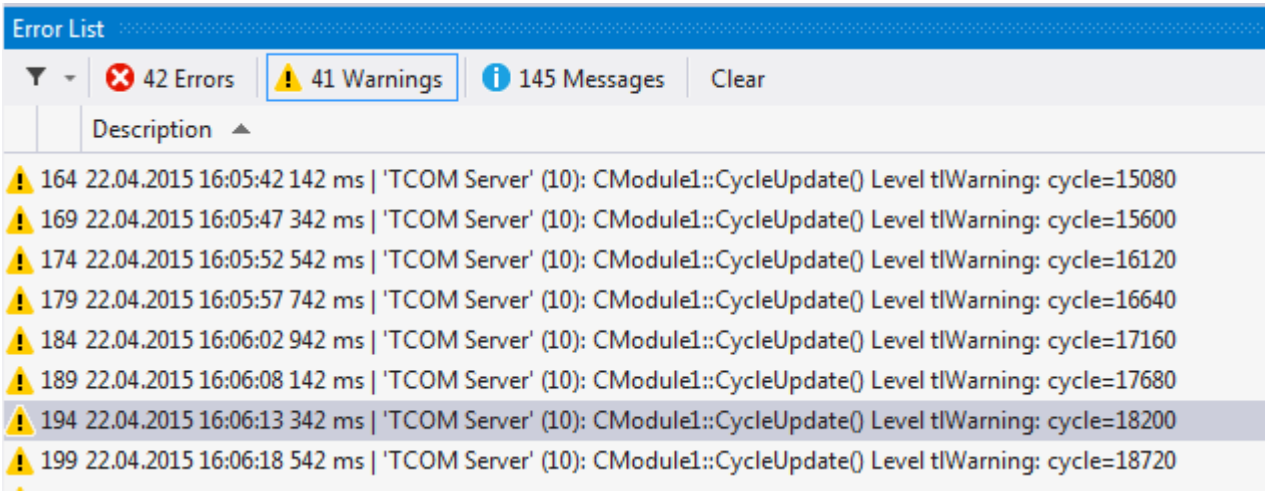
Alternatively, you can change the tracking level at runtime by going to the instance, selecting a level at **Value** for TraceLevelMax parameters, right-clicking in front of the first column, and selecting **Online Write**.



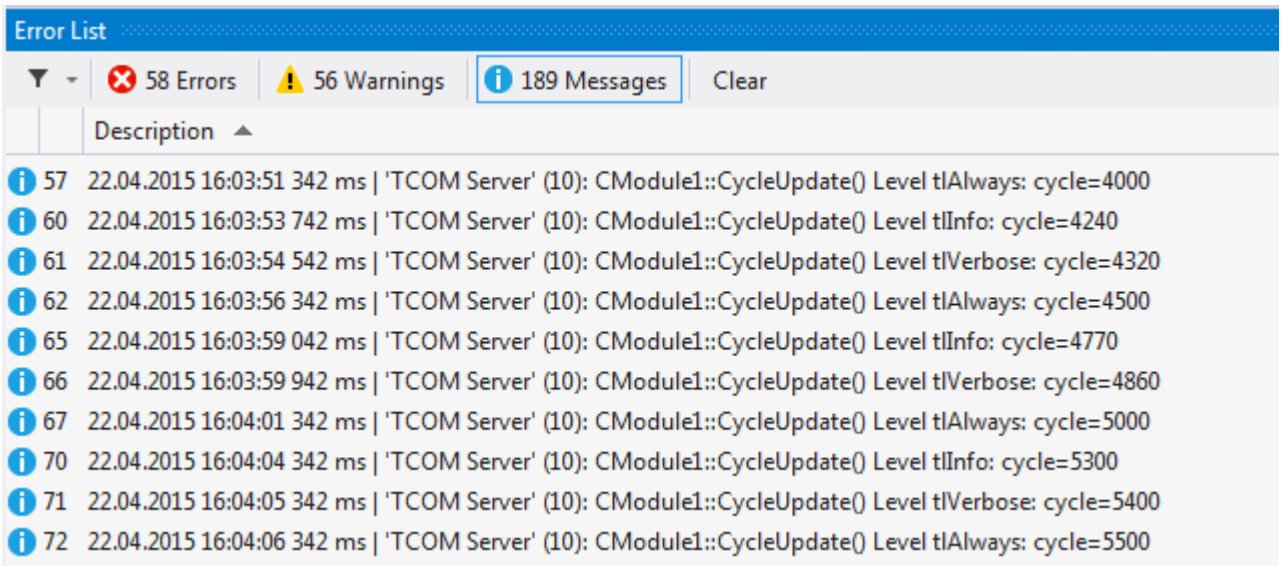
Filter message categories

Visual Studio Error List allows you to filter entries by category. The three categories **Errors**, **Warnings** and **Messages** can be enabled or disabled independently by simply switching the keys.

In this screenshot only **warnings** are enabled - **errors** and **messages** are disabled:



In this screenshot, only **messages** are enabled - **errors** and **warnings**, on the other hand, are disabled for the display:



13 How to...?

The following section contains a set of frequently asked questions about general programming samples and the handling of TwinCAT C++ modules.

13.1 Using the Automation Interface

The Automation Interface can be used for C++ projects.

This includes creating projects [▶ 85] and using the wizard for creating module classes [▶ 86].
In addition, the project properties can be set and the TMC Code Generator and the publishing of modules called. The corresponding documentation [▶ 329] is part of the Automation Interface.

Irrespective of the programming language, access to and creation and handling of TcCOM modules [▶ 333] may be relevant.

From there, common System Manager tasks such as linking of variables can be executed.

13.2 Windows 10 as target system up to TwinCAT 3.1 Build 4022.2

For Windows 10 target systems the transferred files cannot be overwritten; they have to be renamed first.

Up to TwinCAT 3.1 Build 4022.2, the **Rename Destination** option must be enabled for this purpose in the TMC Editor deployment [▶ 127]. In later versions this is done implicitly when the target system uses Windows 10 as operating system.

13.3 Publishing modules on the command line

By means of the following call, the module publishing process in the TwinCAT Engineering (XAE) can also be initiated from the command line:

```
msbuild CppProject.vcxproj /t:TcPublishTMX
```

The `CppProject.vcxproj` parameter must be adapted according to the existing project file.

The output is done into the repository under C:\ProgramData\Beckhoff\TwinCAT\3.1\Repository (TwinCAT 3.1 Build 4024: C:\TwinCAT\3.1\Repository).

13.4 Clone

Runtime data can be transferred from one machine to another via file copy if both machines are from the same platform and are connected with equivalent hardware equipment.

The following steps describe a simple procedure for transferring a binary configuration from one machine “source” to another “target”.

1. Purge the boot folder on the source machine.
   ⇨ Windows < TC3.1.4026.0: *C:\TwinCAT\3.1\Boot*
   Windows >=TC3.1.4026.0: *C:\ProgramData\Beckhoff\TwinCAT\3.1\Boot*
   TwinCAT/BSD: /usr/local/etc/TwinCAT/3.1/Boot/
   Beckhoff RT Linux: /etc/TwinCAT/3.1/Boot/
2. Create (or enable) the module on the source machine.
3. Transfer the boot folder from the source to the target machine.
   This folder also contains the repository which contains the required TMX/TME files.

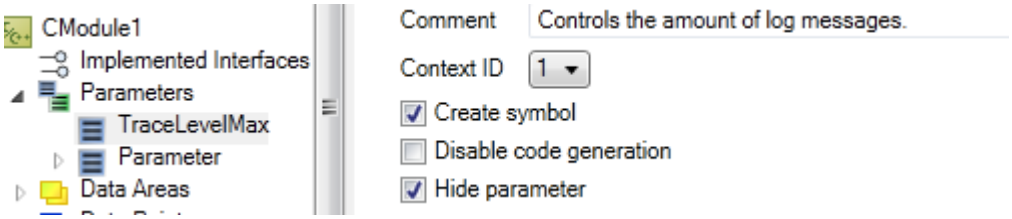
⇨ You can now start the target machine.

Handling licenses

Note that licenses cannot be transferred in this way. Please use preinstalled licenses, volume licenses, or other methods for providing licenses.

13.5 Access Variables via ADS

Variables of C++ modules can be reached via ADS if the variables are marked in the TMC Editor as "Create Symbol":



The name of the variable for access by ADS is derived from the name of the instance.

For the TraceLevelMax parameter it could be:

```
Untitled1_Obj1 (CModule1).TraceLevelMax
```

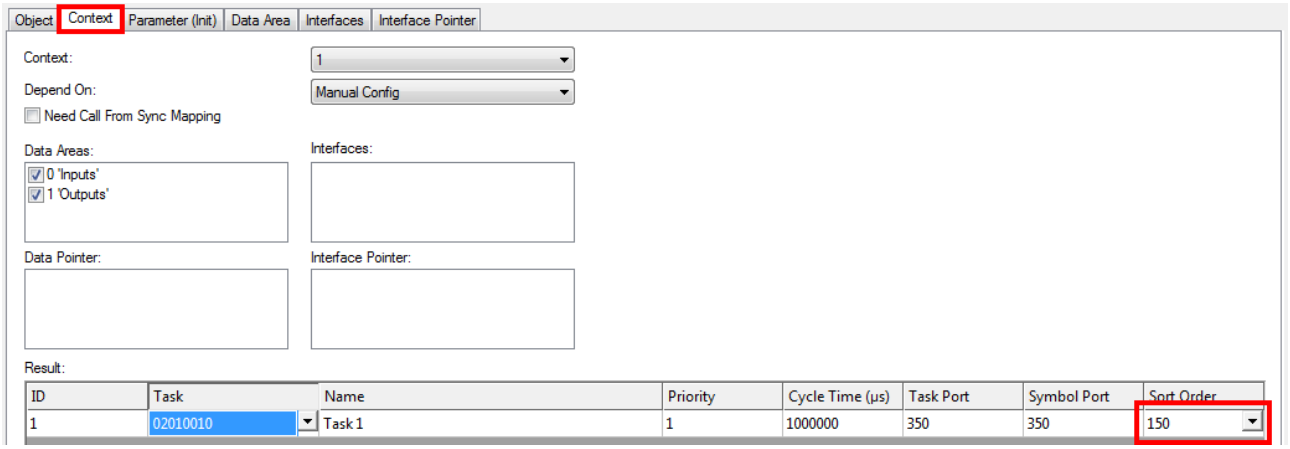
13.6 TcCallAfterOutputUpdate for C++ modules

Comparable with the PLC attribute TcCallAfterOutputUpdate, C++ modules can be called following the output update.
The ITcPostCyclic [▶ 182] interface is used in the same way as the ITcCyclic [▶ 161] interface.

13.7 Order determination of the execution in a task

Different module instances can be assigned to a task, so the user needs a mechanism to determine the order of execution in the task.

It is configured under **Sort Order** in the context [▶ 131] of the TwinCAT Module Instance Configurator [▶ 129].



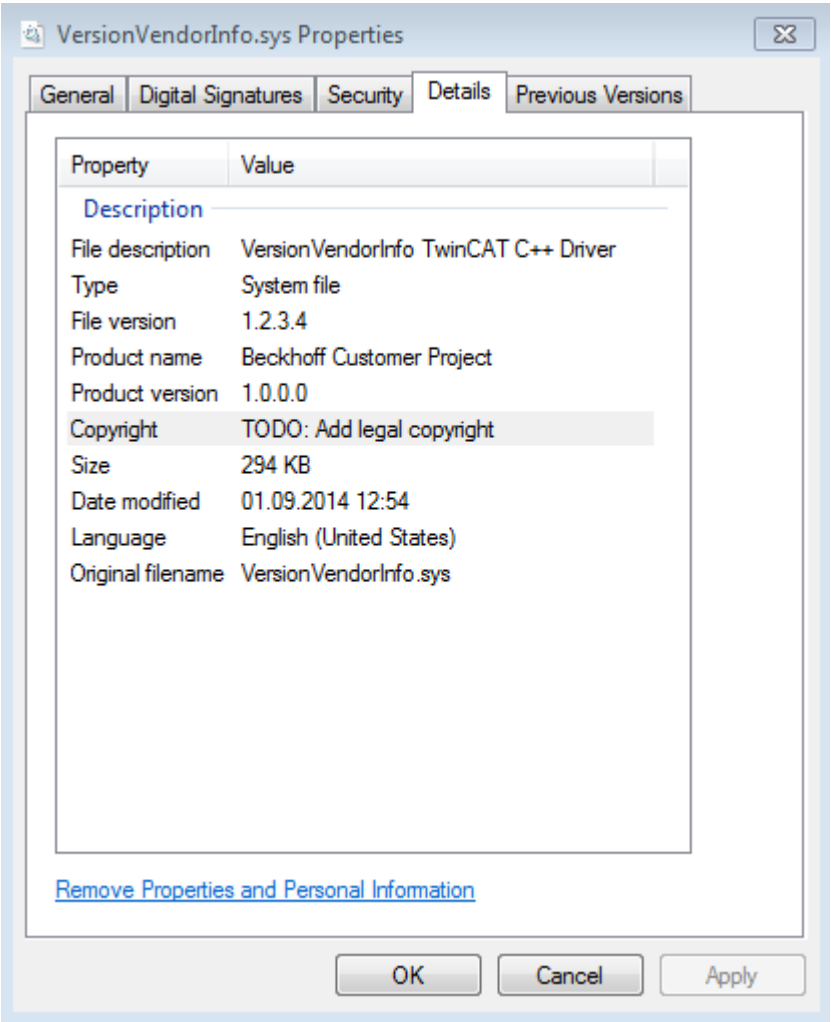
See Sample26: Order of execution in a task [▶ 298], how this is to be implemented.

13.8 Setting version/vendor information

Windows offers a mechanism to query vendor and version resources that are defined in the course of a .rc file for the compilation time.

Windows

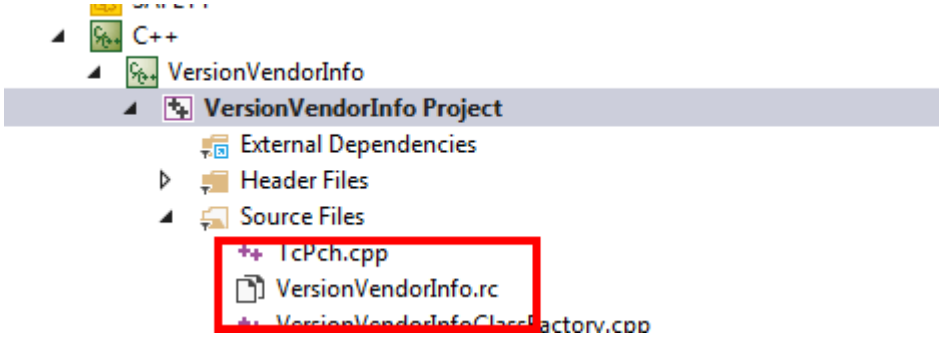
These are accessible, for example, via the **Details** tab of each properties file.



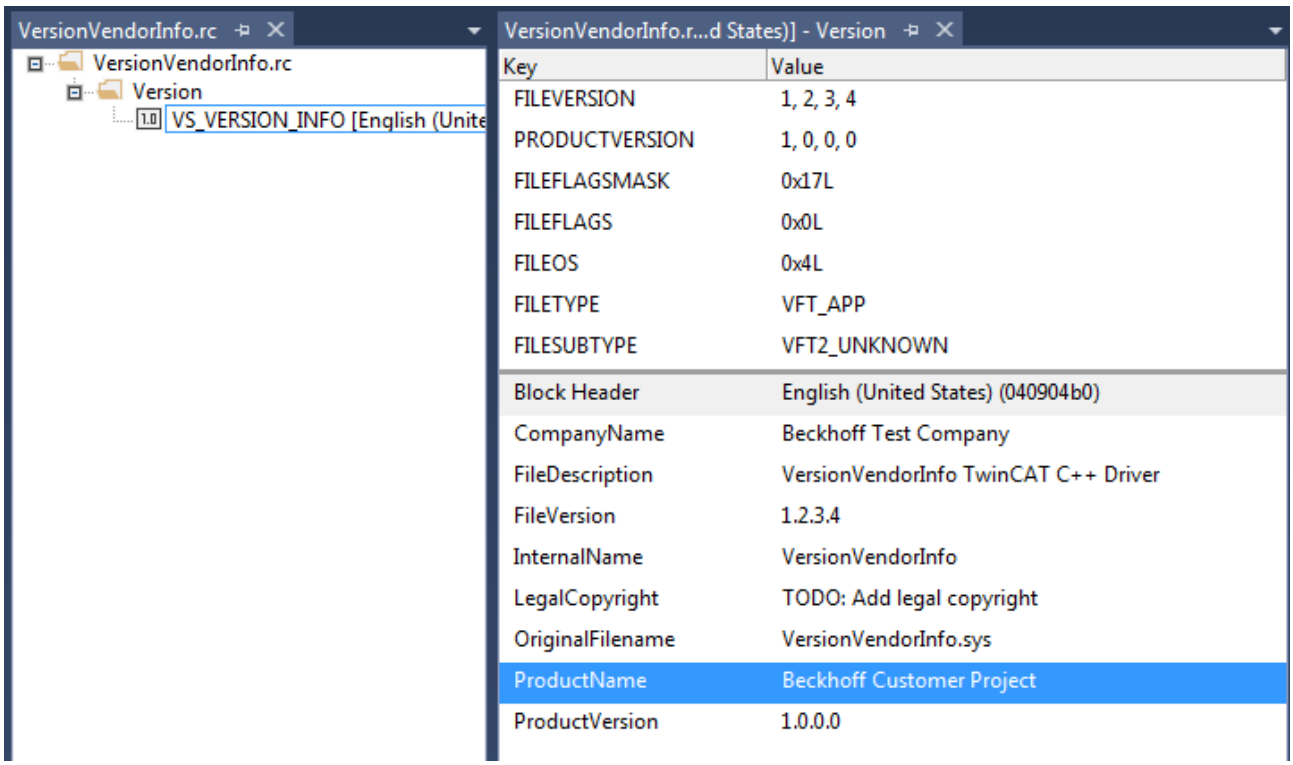
TwinCAT/BSD®

This information is not available in the files.

TwinCAT offers this behavior via the familiar Windows mechanisms of .rc files, which are generated in the course of the TwinCAT C++ project creation.



Edit the .rc file in the Source Files folder with the resource editor in order to define these properties:



13.9 Renaming TwinCAT C++ projects

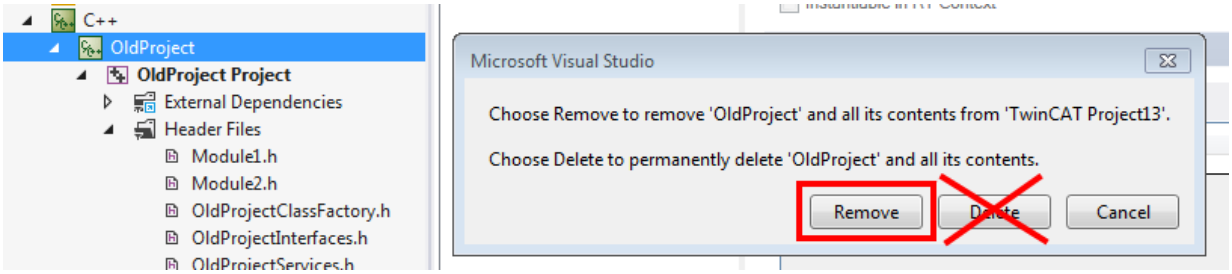
The automated renaming of TwinCAT C++ projects is not possible.

At this point instructions will be given on manually renaming a project.

In summary, one can say that the C++ project will be renamed together with the corresponding files.

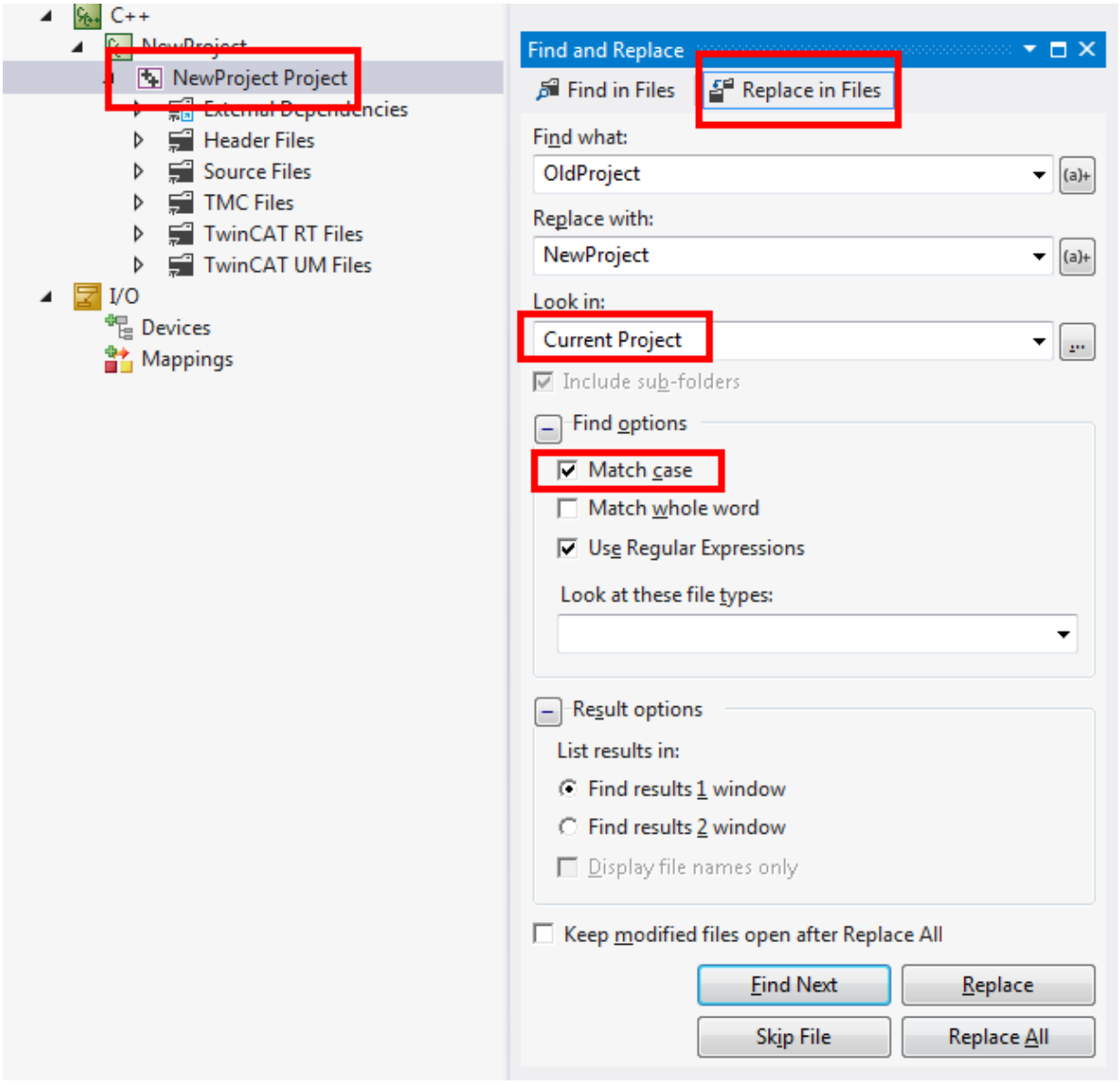
✓ A project, "OldProject", exists and is to be renamed "NewProject".

1. If TcCOM instances exist in the project and are to be retained along with their linking, first move them by drag & drop out of the project into **System->TcCOM Objects**.
2. Remove the old project from the TwinCAT Solution using **Remove**.



3. Compilations of the "OldProject" can be deleted. To do this, delete the corresponding files in "_Deployment".
   Any existing .aps file can also be deleted.
4. Rename the C++ project directory and the project files (.vcxproj, .vcxproj.filters).
   If version management is in use, this renaming must be carried out via the versioning system.
5. If a .vcvproj.user file exists, check the contents; this is where user settings are stored. Also rename this file if necessary.
6. Open the TwinCAT Solution. Re-link the renamed project to the C++ node using **Add existing item**: navigate to the renamed subdirectory and select the .vcxproj file there.
7. Rename the ClassFactory, services and interfaces as well as header/source code files to the new project name. In addition, rename the TMC file and the corresponding files in the project folders "TwinCAT RT Files" and "TwinCAT UM Files".
   This renaming should also be mapped in the version management system; if the version management

system is not integrated in Visual Studio, this step must also be carried out in the version management system. Replace all occurrences in the source code (case-sensitive):
"OLDPROJECT" becomes "NEWPROJECT" and "OldProject" becomes "NewProject".
Use the **Find and Replace** dialog in Visual Studio for this; note that the "NewProject Project" in the Solution Explorer has to be selected.



⇨

NOTICE

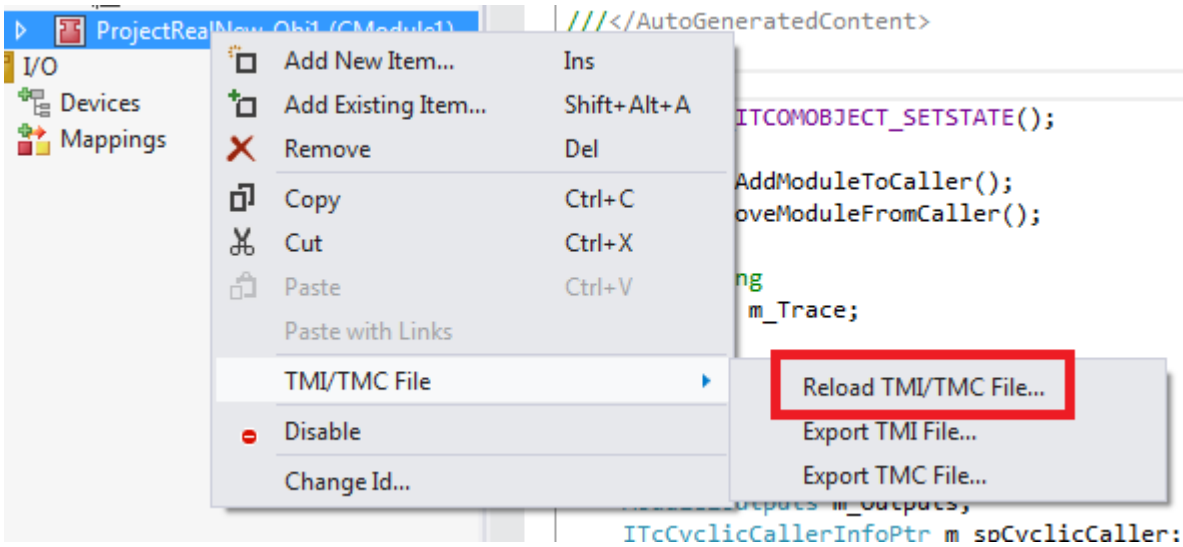
Incorrect source code

The simple renaming of all occurrences of the character string may result in incorrect source code, for example if the project name is used within a method name.

- If such occurrences are possible, carry out the renaming individually (**Replace** instead of **Replace All**).

How to build the project:

1. A) If instances from the project should exist, update them. To do this, right-click on the instance, select **TTMI/TMC File->Reload TMI/TMC File…** and select the renamed new TMC file.



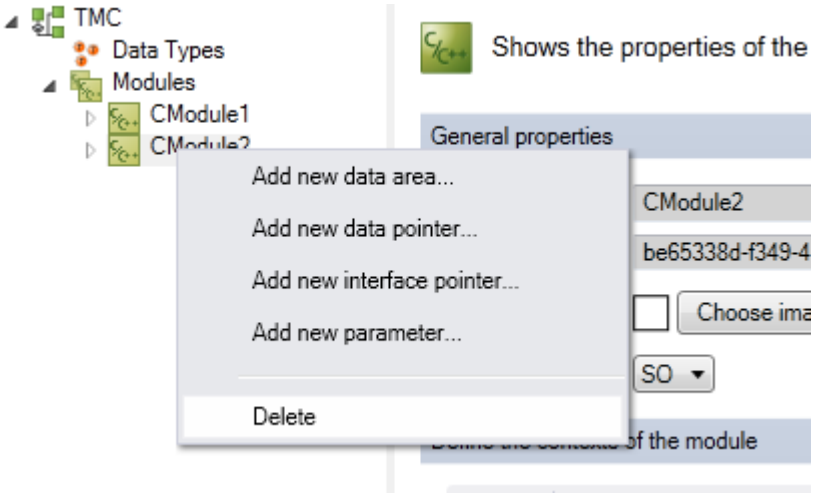
   B) Alternatively, carry this out via **System->TcCOM Objects** and the **Project Objects** tab by right-clicking on the OTCID.

2. Move **System->TcCOM** into the project.
3. Clean up the target system(s).
   For TwinCAT Versioned C++ projects: The repository can be cleaned up under *C:\ProgramData\Beckhoff\TwinCAT\3.1\Repository*
4. Test the project.

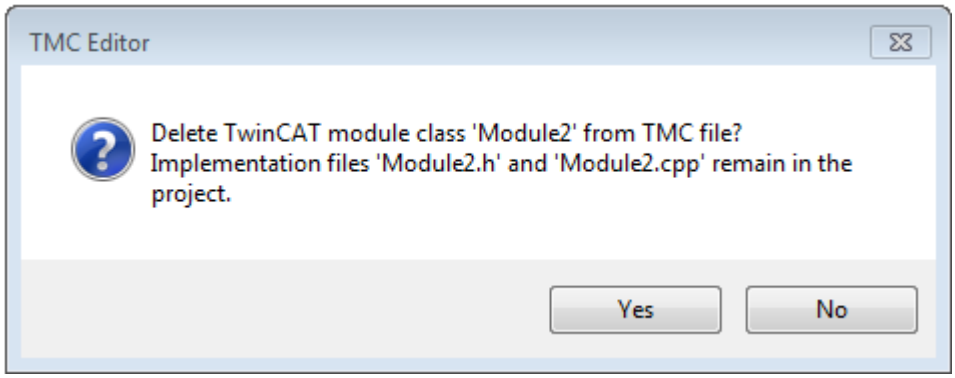
13.10 Delete Module

A TwinCAT C++ module can be deleted from a C++ project with the help of the TMC Editor.

1. Right-click on the module (in this case CModule2)
2. Select **Delete**.



3. Confirm the deletion via TMC.



4. Note that the .cpp and .h files are retained – delete them manually if necessary.
   Delete other components concerned (e.g. header files, structures). See Compiler error messages for more information.

13.11 Add revision control and Online Change subsequently

An existing C++ driver project with C++ TcCOM modules can subsequently be migrated to a versioned C++ project.

The changeover takes place in several steps:

1. Click on **C++ Project TwinCAT Upgrade C+ Project** in the context menu, this is the most convenient option.
   Alternatively, you can find manual instructions here:
   Convert C++ project to a versioned C++ project [▶ 219].
2. You can then add the online change capability; this must be done manually:
   Make C++ module class online change-capable [▶ 222].

Alternatively, it may be useful to create a new versioned C++ project yourself and transfer the differences.

13.11.1 C++ Project -> Revision control

These instructions describe how to add subsequently a revision control to a TwinCAT C++ project.

The code to be changed is stored in **bold** in the source code.

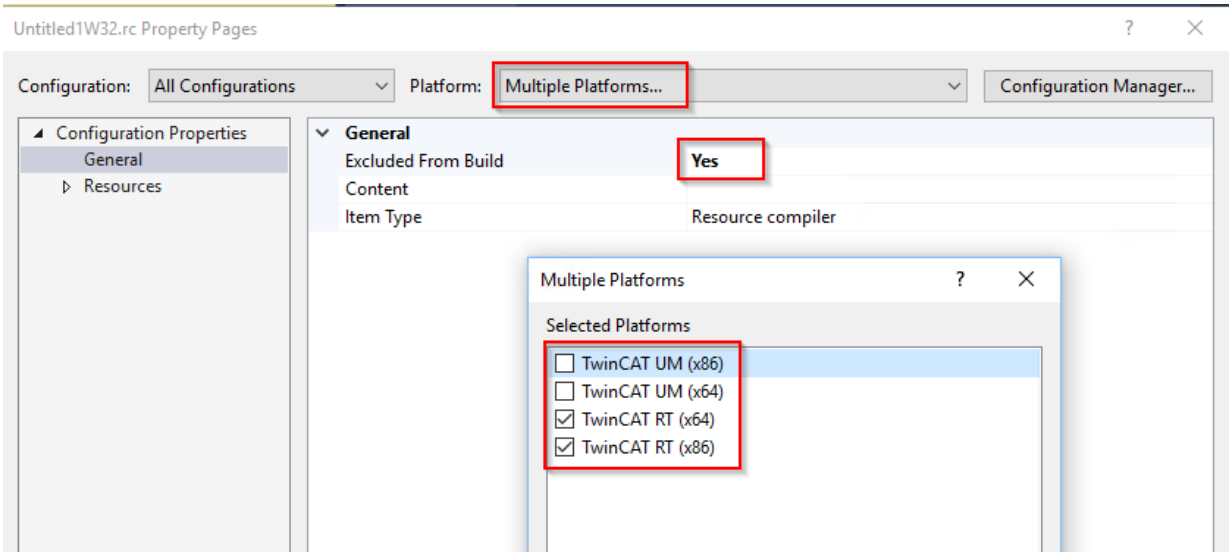
✓ C++ project. For the sample "Untitled1" is used as C++ project name.
In addition, an empty, new project with a versioned C++ project is to be created. This serves as a copy template.

1. Open the file *Untitled1.vcxproj* in an editor.
2. Make the following addition:

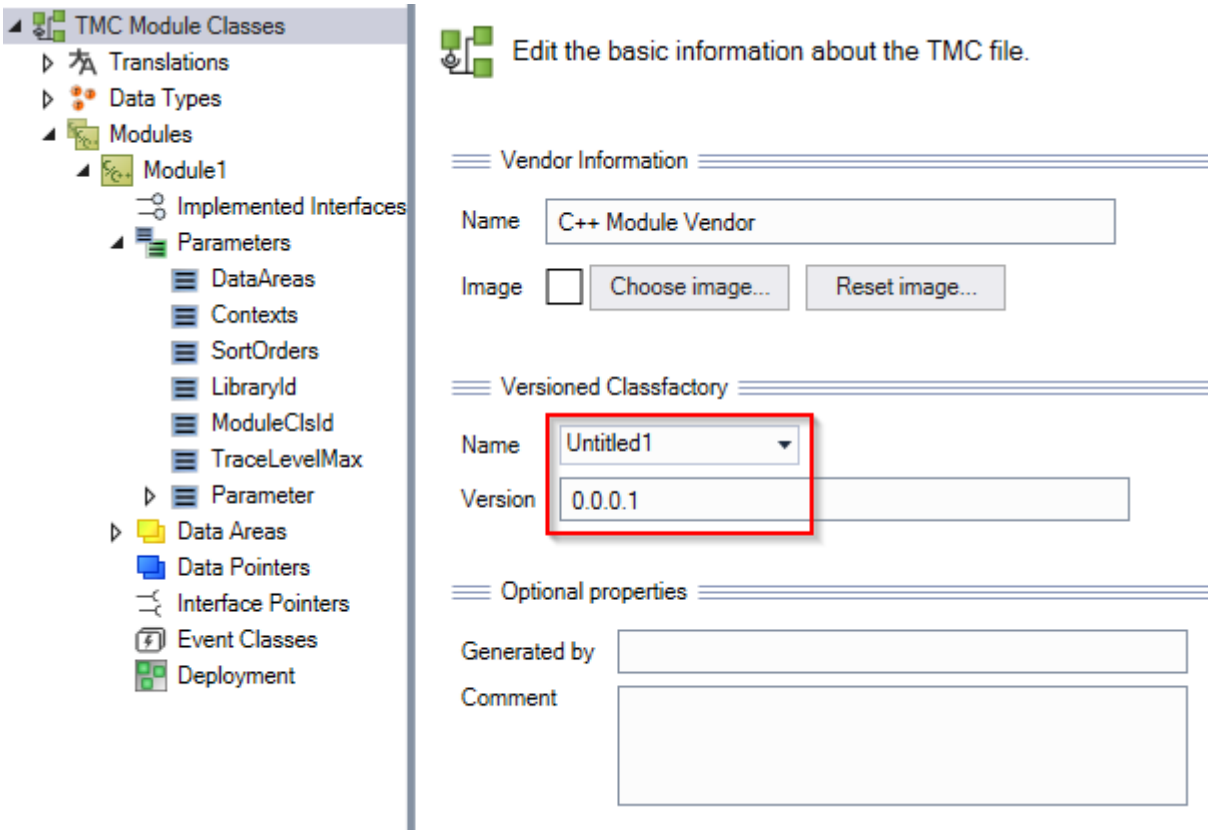
```
<PropertyGroup Label="Globals">
<ProjectGuid>{….}</ProjectGuid>
<RootNamespace>Untitled1</RootNamespace>
<Keyword>Win32Proj</Keyword>
<AutomaticRetargetPlatformVersion>true</AutomaticRetargetPlatformVersion>
</PropertyGroup>
  <PropertyGroup Label="TcGeneral">
<TcGeneralUseTmx>true</TcGeneralUseTmx>
</PropertyGroup>
<Import Project="$(VCTargetsPath)\Microsoft.Cpp.Default.props" />
```

3. Transfer the *Untitled1.rc* and *Untitled1W32.rc* files from the **new** project to the project to be migrated, overwriting the *Untitled1.rc* file.
4. Open the project.
5. Under **TwinCAT UM Files** -> **Context menu / Add Existing Items** select the file *Unititled1W32.rc* and add it to the project.

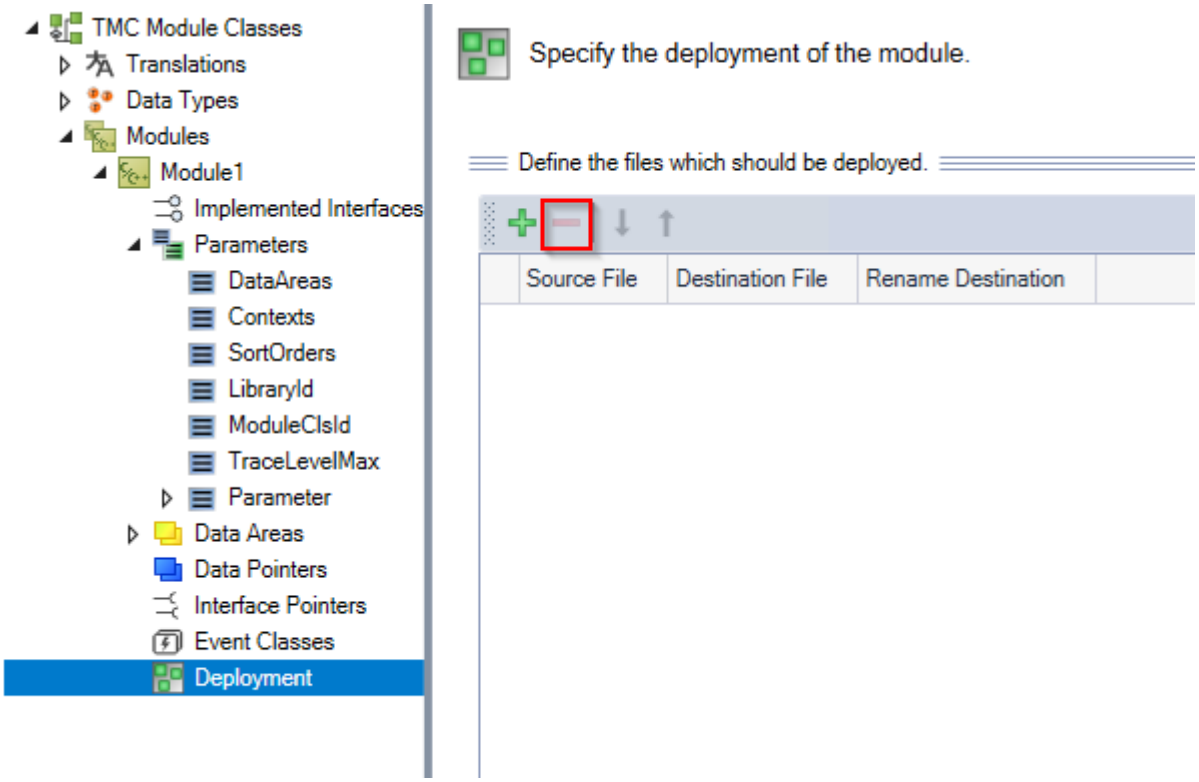
6. This file only needs to be built for TwinCAT UM platforms, so it should otherwise be excluded for the build process. This is done via right-click and properties:



7. Open the TMC Editor.
8. Set the name of the class factory and the version to "0.0.0.1".



9. Under **Deployment,** delete all entries.



10. Change in header *Modul1.h* DECLARE_IPERSIST**_LIB**():

```
public:
DECLARE_IUNKNOWN()
DECLARE_IPERSIST_LIB()
DECLARE_ITCOMOBJECT_LOCKOP()
```

11. Add *Modul1.cpp* to the source code:

```
#pragma hdrstop
#include "Module1.h"
#include "Untitled1Version.h"

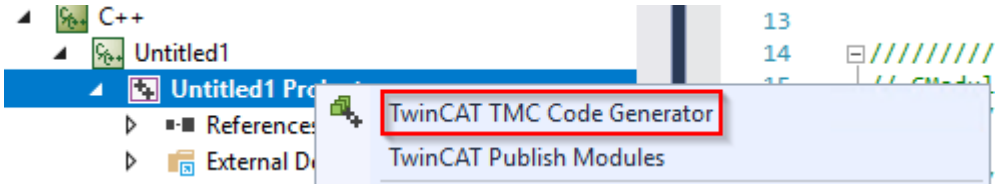
#ifdef _DEBUG
```

12. Add *Modul1.cpp* to the source code:

```
END_INTERFACE_MAP()

IMPLEMENT_IPERSIST_LIB(CModule1, VID_Untitled1,
CID_Untitled1CModule1)
IMPLEMENT_ITCOMOBJECT(CModule1)
IMPLEMENT_ITCOMOBJECT_SETSTATE_LOCKOP2(CModule1)
```

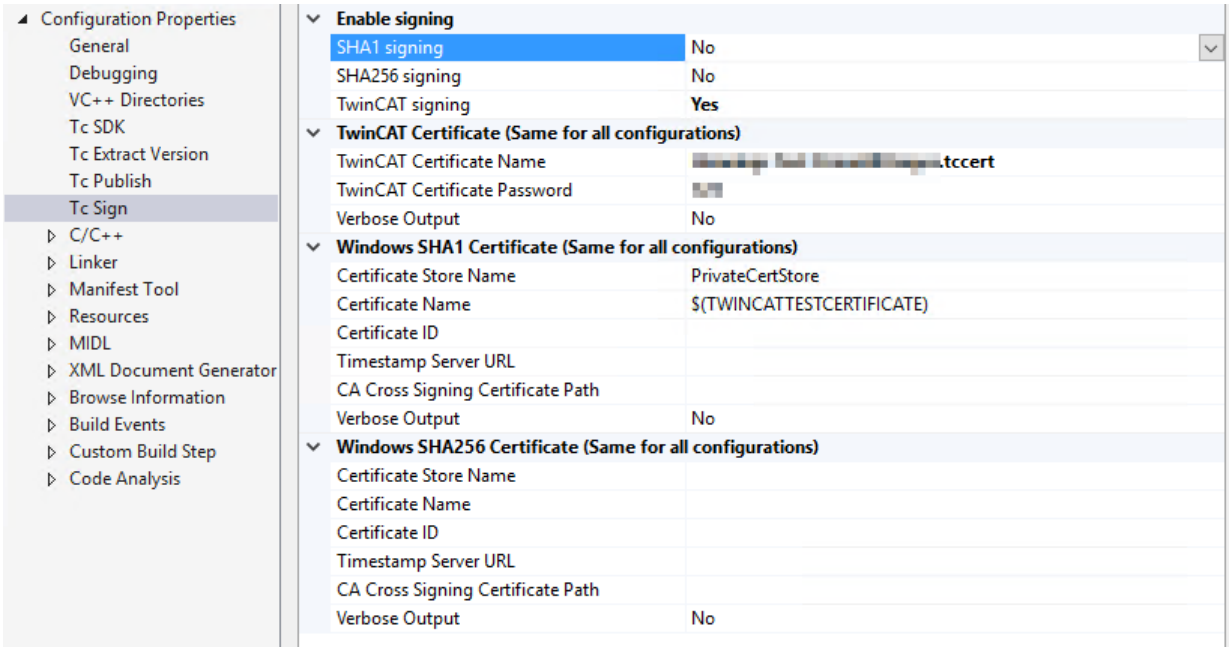
13. Call the code generator.



14. Change in the file *Untitled1Classfactory.cpp*:

```
CUntitled1ClassFactory::CUntitled1ClassFactory() : CObjClassFactory()
{
TcDbgUnitSetImageName(TCDBG_UNIT_IMAGE_NAME_TMX(SRVNAME_UNTITLED1));
#if defined(TCDBG_UNIT_VERSION)
TcDbgUnitSetVersion(TCDBG_UNIT_VERSION(Untitled1));
#endif //defined(TCDBG_UNIT_VERSION)
}
```

15. Change the signing in the project properties in the tab Tc Sign [▶ 144]. To do this, switch **SHA1 signing** off and **TwinCAT signing** on; provide TwinCAT user certificate and password at the same time.



16. Trigger the **Rebuild** of the project.

⇨ The result is a C++ project that supports revision control.

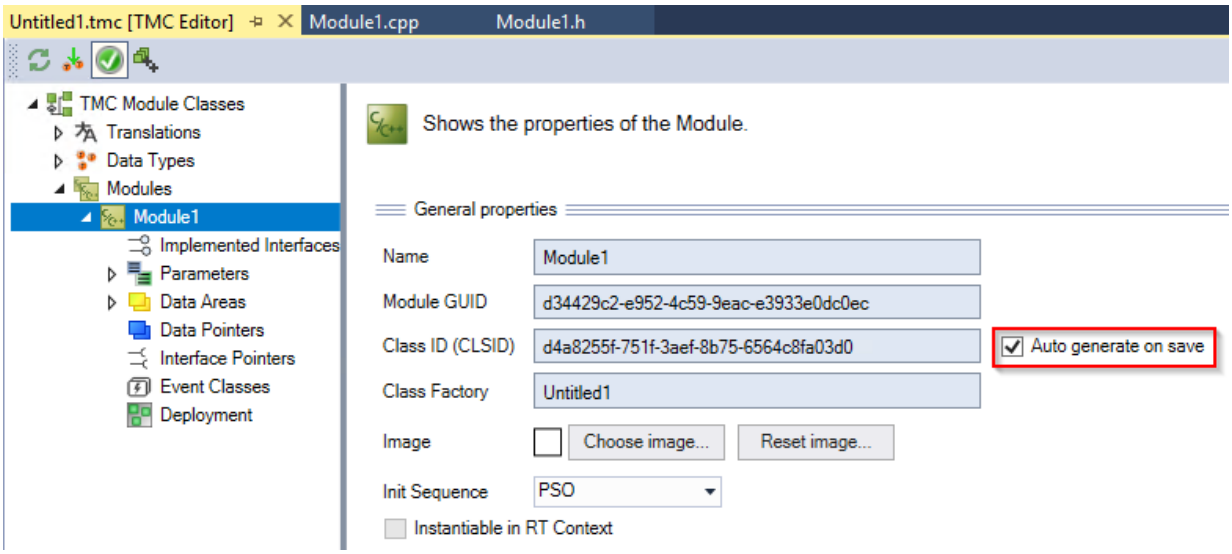
If a module is to be made Online Change capable, this can be achieved by the following instructions [▶ 222].

13.11.2 C++ Module -> OnlineChange

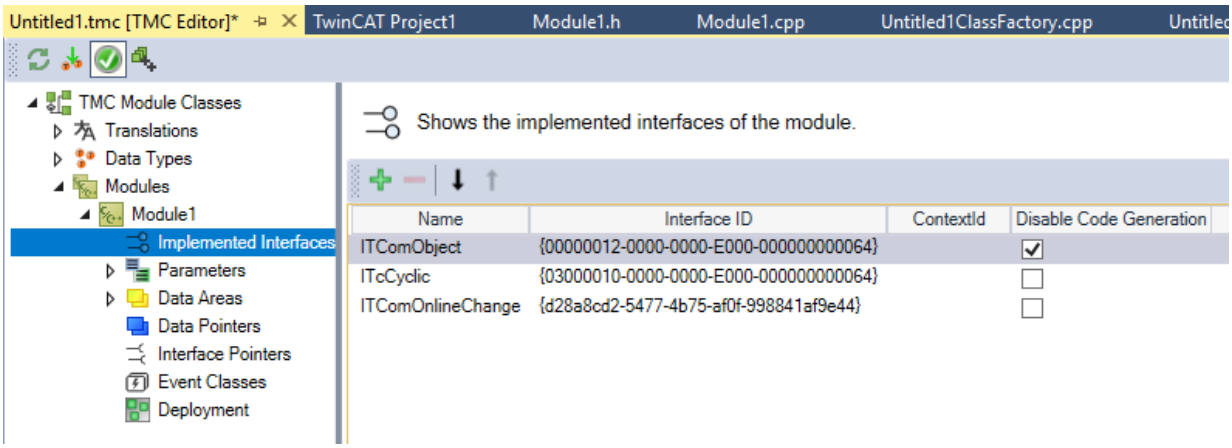
These instructions describe how to subsequently make a module OnlineChange-capable in a versioned TwinCAT C++ project.

✓ Versioned C++ project with C++ module, which is not yet Online Change capable. For this sample, a module "Module1" is assumed. In addition, an empty, new project can be created with an Online Change-capable module, from which the changes can be more easily adopted.

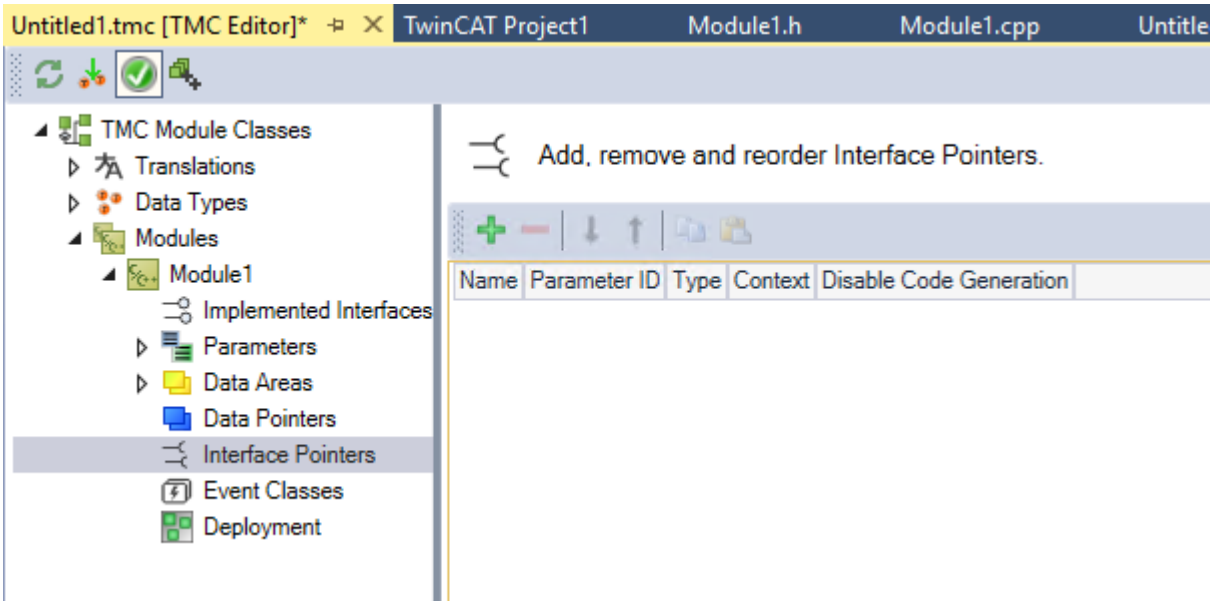
1. Open the project and the TMC Editor.
2. Set the **Auto generate on save** option for the module so that the ClassID is changed automatically.



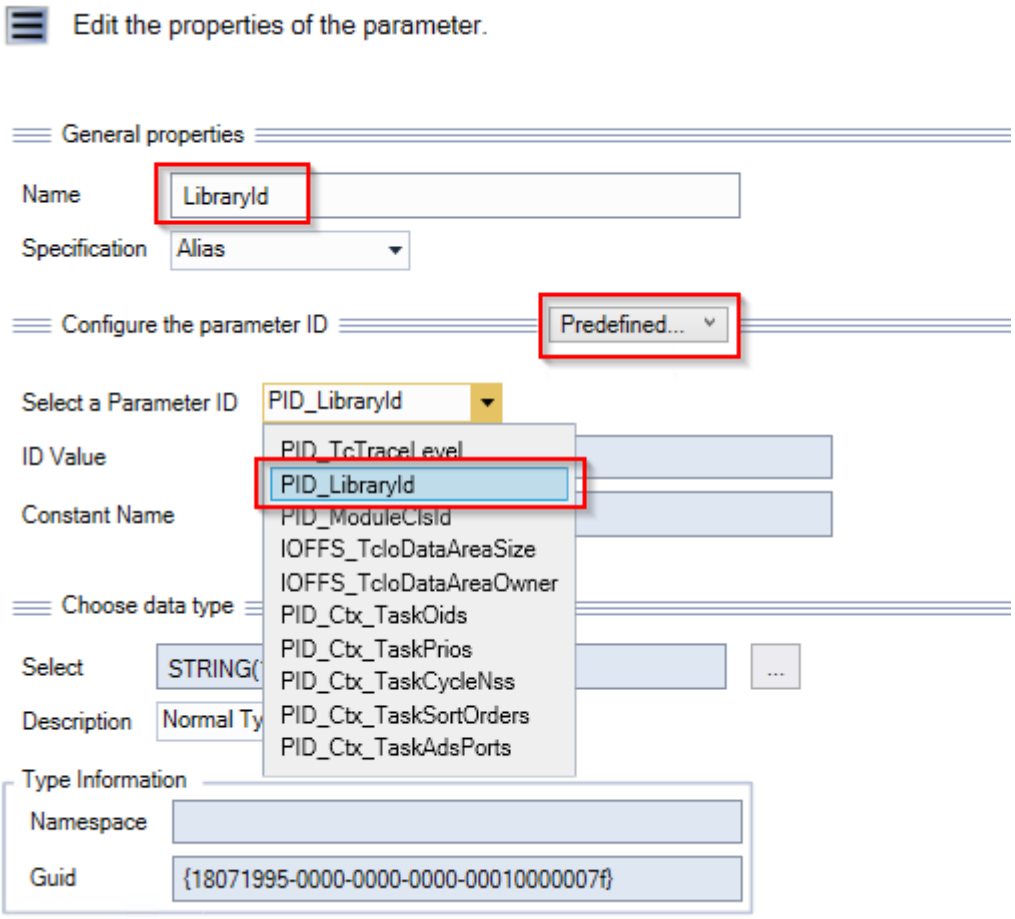
3. Under **Implemented Interfaces,** delete the interfaces ITcADI and ITcWatchsource and add ITComOnlineChange.



4. Delete the CyclicCaller under **Interface Pointer**.

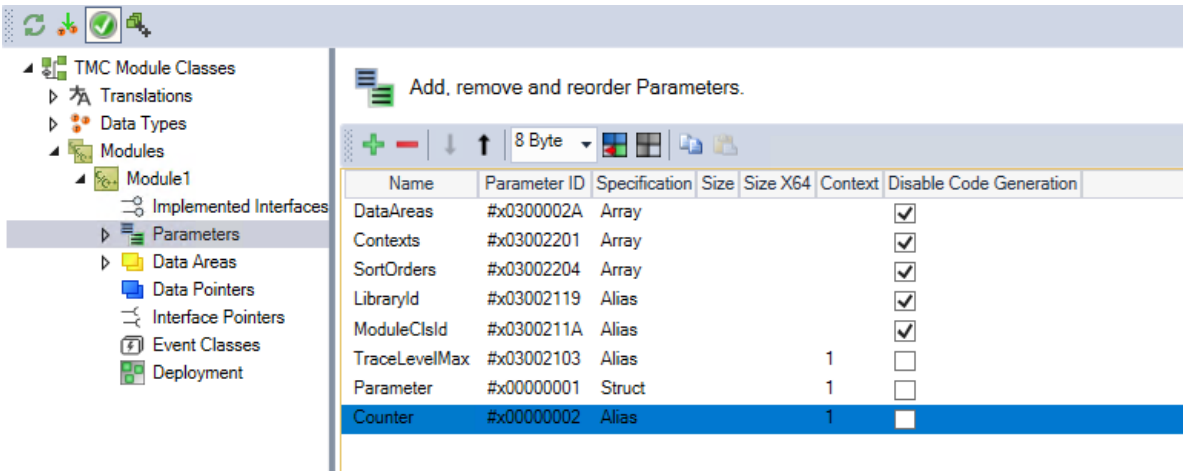


5. Under **Parameters** some predefined parameters have to be added.
   Press **+** under **Parameters** and select **Predefined** to select the predefined ParameterIDs:

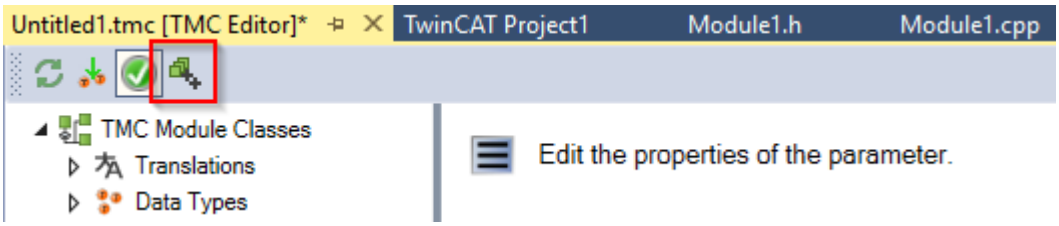


You create the following parameters with the given names, in each case without code generation:
"PID_LibraryID" with the name "LibraryID"
"PID_ModuleClsId" with the name "ModuleClsId"
"PID_Ctx_TaskSortOrders with the name "SortOrders"
"PID_Ctx_TaskOids" with the name "Contexts"
"IOFFS_TcIoDataAreaSize" with the name "DataAreas"

⇨ The result in the overview:



6. Start the code generation.



7. Some changes must be made in the header of the module "Module1.h". First, delete the declarations of the interfaces that are no longer required and the corresponding maps in two places.

```
class CModule1
: public ITComObject
, publicITcADI
, publicITcWatchSource
///<AutoGeneratedContent id="InheritanceList">
, public ITcCyclic
///</AutoGeneratedContent>
{
public:
DECLARE_IUNKNOWN()
DECLARE_IPERSIST(CID_Untitled1CModule1)
DECLARE_ITCOMOBJECT_LOCKOP()
DECLARE_ITCADI()
DECLARE_ITCWATCHSOURCE()
DECLARE_OBJPARAWATCH_MAP()
DECLARE_OBJDATAAREA_MAP()
```

8. Create a new member variable in the header:

```
///</AutoGeneratedContent>
ITcADIPtr m_spADI;
// TODO: Custom variable
```

9. Some changes need to be made to the source code of the module "Module1.cpp". First, delete the implementations of the interfaces that are no longer required in two places.

```
BEGIN_INTERFACE_MAP(CModule1)
INTERFACE_ENTRY_ITCOMOBJECT()
INTERFACE_ENTRY(IID_ITcADI, ITcADI)
INTERFACE_ENTRY(IID_ITcWatchSource, ITcWatchSource)
///<AutoGeneratedContent id="InterfaceMap">
INTERFACE_ENTRY(IID_ITcCyclic, ITcCyclic)
///</AutoGeneratedContent>
END_INTERFACE_MAP()
IMPLEMENT_ITCOMOBJECT(CModule1)
IMPLEMENT_ITCOMOBJECT_SETSTATE_LOCKOP2(CModule1)
IMPLEMENT_ITCADI(CModule1)
IMPLEMENT_ITCWATCHSOURCE(CModule1)
```

10. Delete the implementation of the maps belonging to the deleted interfaces:

```
BEGIN_SETOBJPARA_MAP(CModule1)
SETOBJPARA_DATAAREA_MAP()
///<AutoGeneratedContent id="SetObjectParameterMap">
SETOBJPARA_VALUE(PID_TcTraceLevel, m_TraceLevelMax)
SETOBJPARA_VALUE(PID_Module1Parameter, m_Parameter)
SETOBJPARA_ITFPTR(PID_Ctx_TaskOid, m_spCyclicCaller)
///</AutoGeneratedContent>
END_SETOBJPARA_MAP()
////////////////////////////////////////////////////////////////////////////////
// Get parameters of CModule1
BEGIN_GETOBJPARA_MAP(CModule1)
GETOBJPARA_DATAAREA_MAP()
///<AutoGeneratedContent id="GetObjectParameterMap">
GETOBJPARA_VALUE(PID_TcTraceLevel, m_TraceLevelMax)
GETOBJPARA_VALUE(PID_Module1Parameter, m_Parameter)
GETOBJPARA_ITFPTR(PID_Ctx_TaskOid, m_spCyclicCaller)
///</AutoGeneratedContent>
END_GETOBJPARA_MAP()
////////////////////////////////////////////////////////////////////////////////
// Get watch entries of CModule1
BEGIN_OBJPARAWATCH_MAP(CModule1)
OBJPARAWATCH_DATAAREA_MAP()
///<AutoGeneratedContent id="ObjectParameterWatchMap">
///</AutoGeneratedContent>
END_OBJPARAWATCH_MAP()
////////////////////////////////////////////////////////////////////////////////
```

```
// Get data area members of CModule1
BEGIN_OBJDATAAREA_MAP(CModule1)
///<AutoGeneratedContent id="ObjectDataAreaMap">
OBJDATAAREA_VALUE(ADI_Module1Inputs, m_Inputs)
OBJDATAAREA_VALUE(ADI_Module1Outputs, m_Outputs)
///</AutoGeneratedContent>
END_OBJDATAAREA_MAP()
```

11. The m_spAPI pointer must be obtained in the transition P->S in the state machine:

```
HRESULT CModule1::SetObjStatePS(PTComInitDataHdr pInitData)
{
m_Trace.Log(tlVerbose, FENTERA);
HRESULT hr = S_OK;
IMPLEMENT_ITCOMOBJECT_EVALUATE_INITDATA(pInitData);
// query TcCOM object server for ITcADI interface with own object id,
// which retrieves a reference to the TMC module instance handler
m_spADI.SetOID(m_objId);
hr = FAILED(hr) ? hr : m_spSrv->TcQuerySmartObjectInterface(m_spADI);
// TODO: Add initialization code
```

12. Add the following in the state machine in the transition S->O, the calls to AddModuleToCaller or RemoveModuleFromCaller are omitted.

```
HRESULT hr = S_OK;
// Retrieve pointer to data areas via ITcADI interface from TMC module handler
///<AutoGeneratedContent id="DataAreaPointerInitialization">
///</AutoGeneratedContent>
// TODO: Add any additional initialization
// Cleanup if transition failed at some stage
if ( FAILED(hr) )
{
   SetObjStateOS();
}
```

13. Add the following in the state machine in the transition O->S, the call RemoveModuleFromCaller is omitted:

```
HRESULT hr = S_OK;
// Release pointer to data areas via ITcADI interface from TMC module handler
///<AutoGeneratedContent id="DataAreaPointerRelease">
///</AutoGeneratedContent>
// TODO: Add any additional deinitialization
```

14. The AddModuleToCaller and RemoveModuleFromCaller methods are not required and can be deleted.
15. Change the accesses to the DataAreas:

```
HRESULT CModule1::CycleUpdate(ITcTask* ipTask, ITcUnknown* ipCaller, ULONG_PTR context)
{
  HRESULT hr = S_OK;
  // TODO: Replace the sample with your cyclic code
  m_Counter+=m_pInputs->Value;
  m_pOutputs->Value=m_Counter;
  return hr;
}
```

16. Implement the ITcOnlineChange. The functions are created by the previous code generation, but must not return NOTIMPL.

```
///<AutoGeneratedContent id="ImplementationOf_ITComOnlineChange">
//////////////////////////////////////////////////////////////////////////////
// PrepareOnlineChange is called after this instance has been set to PREOP in non RT context.
// Parameter ipOldObj refers to the currently active instance which is still in OP.
// Retrieve parameter values that are not changed during OP via ipOldObj here.
//
// Parameter pOldInfo refers to instance data which includes the libraryId and
// the module class id. This information can be used to implement switch from one
// specific version to another.
HRESULT CModule1::PrepareOnlineChange(ITComObject* ipOldObj, TmcInstData* pOldInfo)
{
HRESULT hr = S_OK;

ULONG nData = sizeof(m_Parameter);
PVOID pData = &m_Parameter;
ipOldObj->TcGetObjPara(PID_Module1Parameter, nData, pData);
return hr;
}
//////////////////////////////////////////////////////////////////////////////
// PerformOnlineChange is called after this instance has been set to SAFEOP in RT context.
// Parameter ipOldObj refers to old instance which is now in SAFEOP.
// Allows to retrieve data after the last cyclic update of the old instance and
```

```
// before the first cyclic update of this instance.
HRESULT CModule1::PerformOnlineChange(ITComObject* ipOldObj, TmcInstData* pOldInfo)
{
HRESULT hr = S_OK;

ULONG nData = sizeof(m_Counter);
PVOID pData = &m_Counter;
ipOldObj->TcGetObjPara(PID_Module1Counter, nData, pData);
return hr;
}
///</AutoGeneratedContent>
```

17. Start the TMC Code Generator again to generate the code for the initialization of the data area pointers.

13.12 Initialization of TMC-member variables

All member variables of a TcCOM module must be initialized. The TMC Code Generator supports this with:

```
///<AutoGeneratedContent id="MemberInitialization">
```

The TMC Code Generator replaces this with:

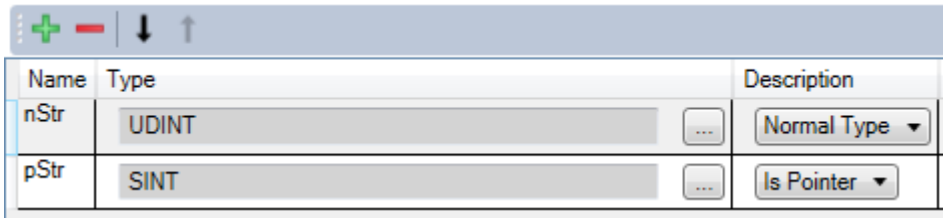
```
 ///<AutoGeneratedContent id="MemberInitialization">
m_TraceLevelMax = tlAlways;
memset(&m_Parameter, 0, sizeof(m_Parameter));
memset(&m_Inputs, 0, sizeof(m_Inputs));
memset(&m_Outputs, 0, sizeof(m_Outputs));
///</AutoGeneratedContent>
```

The projects generated with the TwinCAT C++ Wizard prior to TwinCAT 3.1 Build 4018 do not use this property, but can easily be adapted by inserting this line in the corresponding code (e.g. Constructor):

```
///<AutoGeneratedContent id="MemberInitialization">
```

13.13 Using PLC strings as method parameters

To transfer a character string from PLC to C++ as a method parameter, use a pointer with length information when declaring the method in TMC:



Such a method can be called by means of implementing a method within the wrapper function block.

```
1  METHOD SetString : HRESULT
2  VAR_INPUT
3      sSent : STRING(80);
4  END_VAR
```

```
1  IF (ipStateMachine <> 0) THEN
2      SetString := ipStateMachine.SetString(SIZEOF(sSent),ADR(sSent));
3  END_IF
```

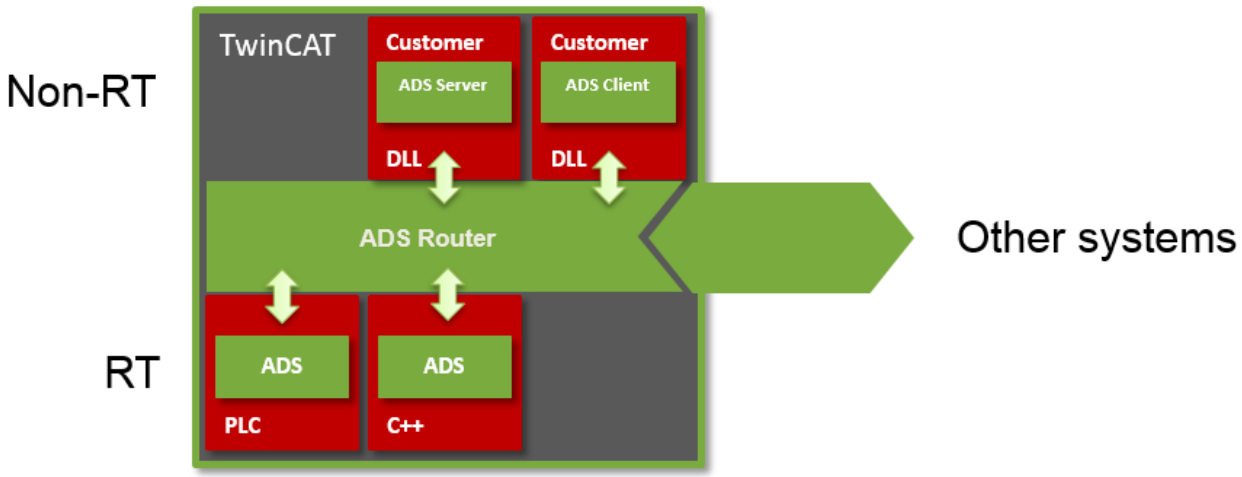
The reason is the different handling of method parameters in the two worlds:

- PLC: Uses the call by value for STRING(nn) data types.
- TwinCAT C++ (TMC): Uses the call by reference.

13.14 Third Party Libraries

C/C++ code existing in Kernel mode cannot be linked with or execute libraries from third parties that were developed for execution in Usermode. There is therefore no possibility to use any DLL directly in TwinCAT C++ modules.

The connection of the TwinCAT 3 real-time environment can be realized via ADS communication instead. You can implement a Usermode application that makes use of the third-party library that provides TwinCAT functions via ADS.



This action of an ADS component in Usermode can take place both as a client (i.e. the DLL transmits data to the TwinCAT real-time if necessary) and as a server (i.e. the TwinCAT real-time fetches data from the Usermode if necessary).

Such an ADS component in Usermode can also be used in the same way from the PLC. In addition, ADS can communicate beyond device limits.

The following samples illustrate the use of ADS in C++ modules:

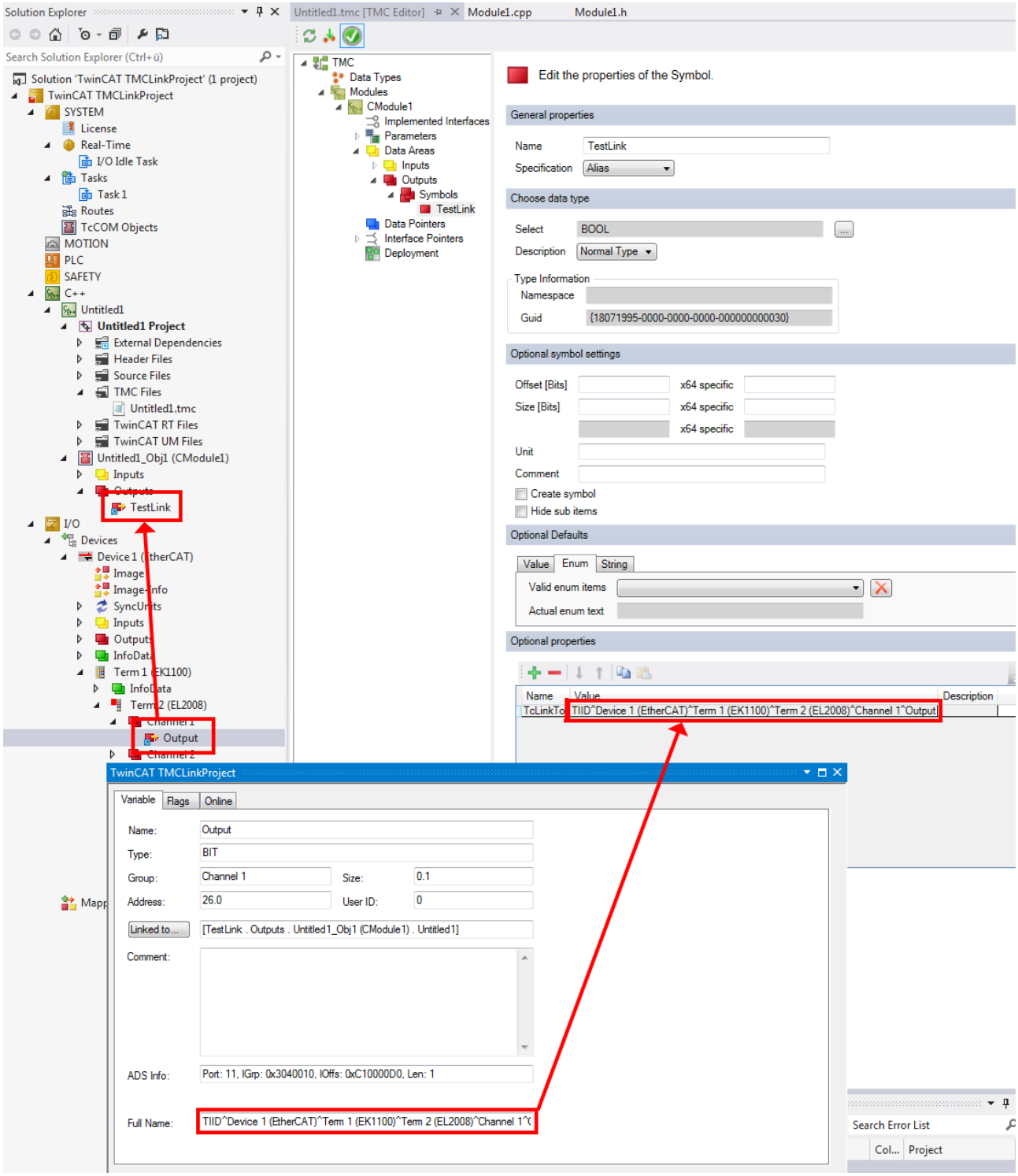
Sample03: C++ as ADS server [▶ 237]

Sample07: Receiving ADS Notifications [▶ 252]

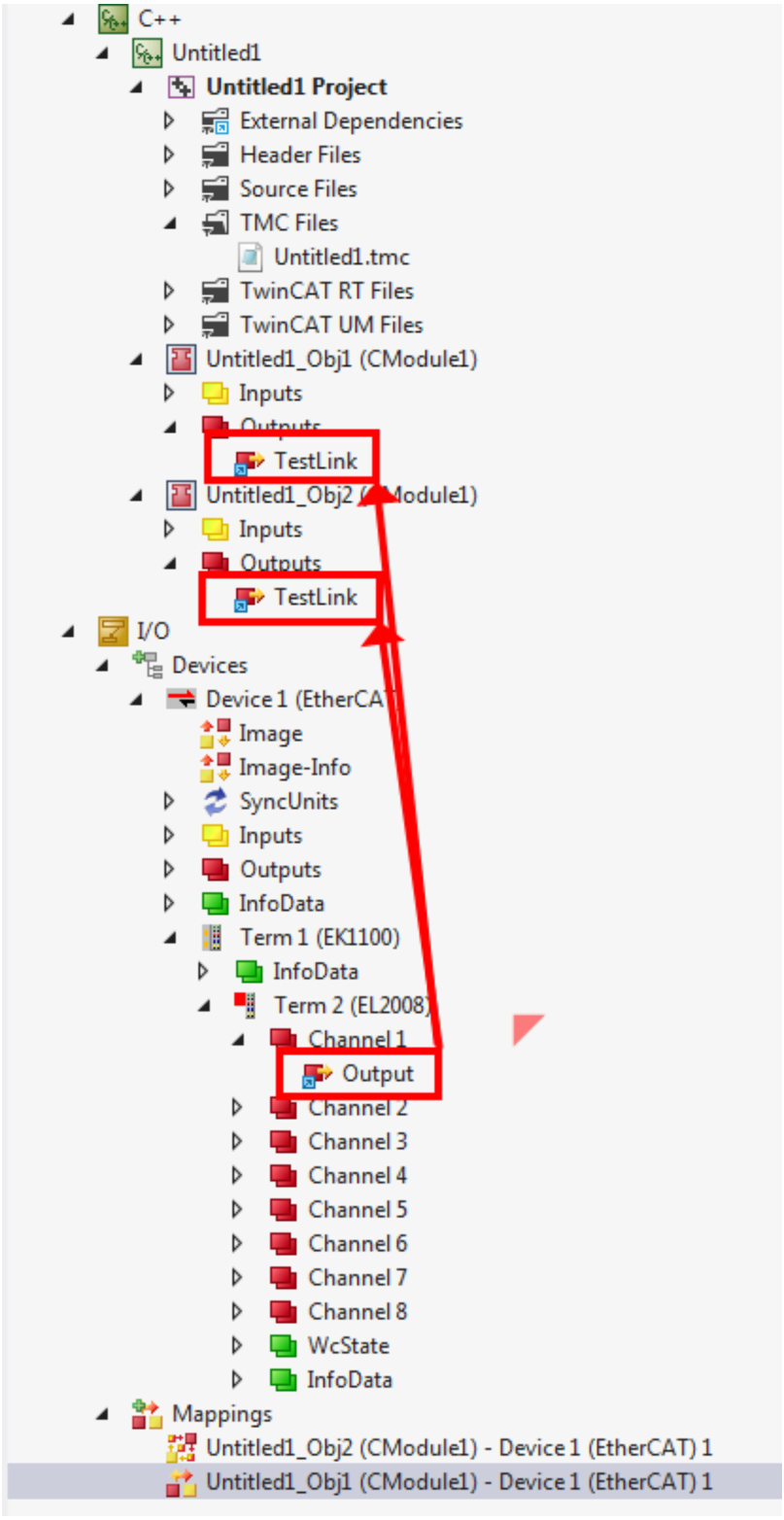
Sample08: provision of ADS-RPC [▶ 253]

13.15 Linking via TMC editor (TcLinkTo)

Similar to the PLC, in TwinCAT C++ a link to the hardware, for example, can be predefined at the time of encoding.
This is done in the TMC editor at the symbol to be linked. A **TcLinkTo** property with the value of the target is specified. The screenshot below illustrates this:



Note that such an instruction applies to all instances of the module:



13.16 Online Change per ADS

The Online Change [▶ 148] is normally carried out from XAE Engineering. However, it is also possible to carry out the Online Change via ADS and thus in your own programs:

- ADS WRITE
- AmsPort: 11
- IdxGrp: ObjectID
- IdxOffset: 16#03002119 (PID_LibraryId)

- Daten: Library Id (“<vendor>|<library>|<version>”)

14 Troubleshooting

This is a list of pitfalls and glitches within the handling of TwinCAT C++ modules.

14.1 Using C++ classes in TwinCAT C++ module

When adding (non-TwinCAT) C++ classes using the Visual Studio context menu **Add->Class...**, the compiler/linker reports:

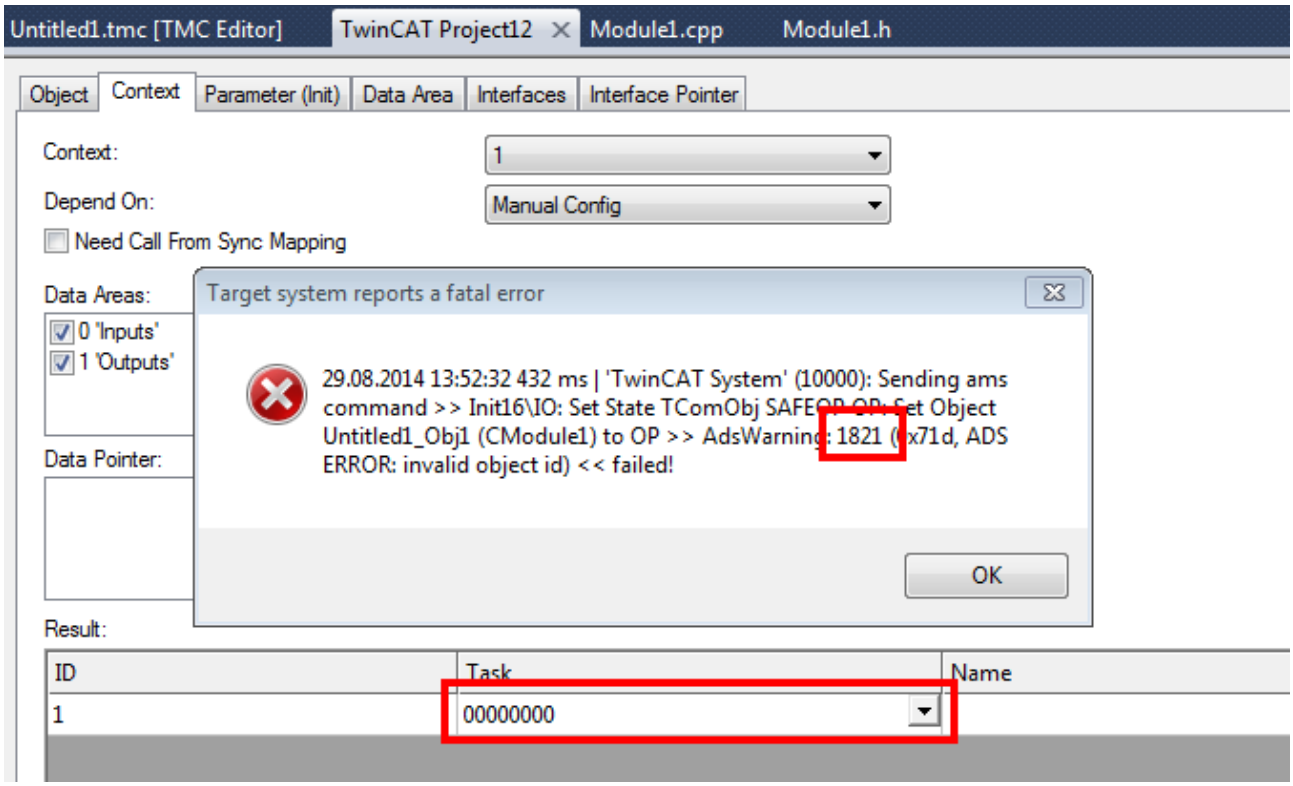
```
Error 4 error C1010: unexpected end of file while looking for precompiled
header. Did you forget to add '#include ""' to your source?
```

Insert the following lines at the start of your generated class file:

```
#include "TcPch.h"
#pragma hdrstop
```

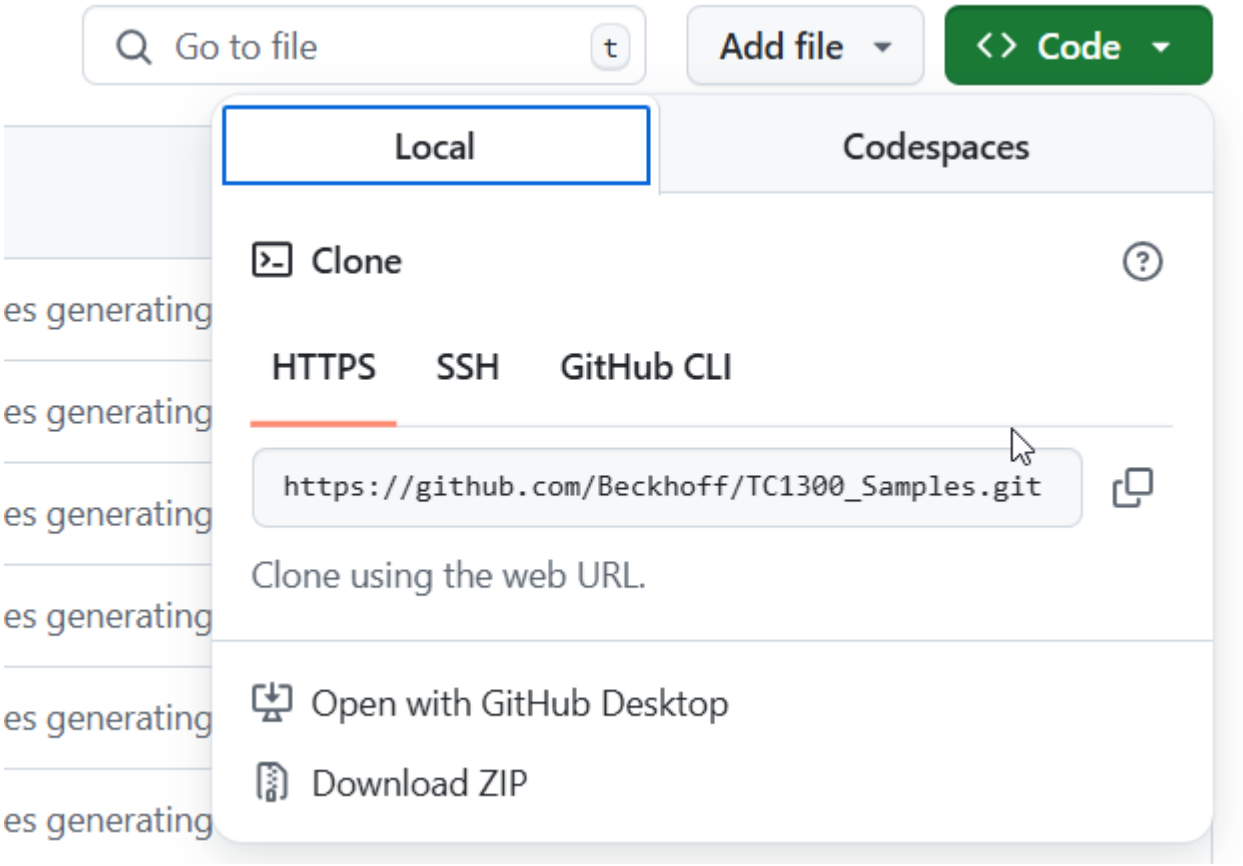
14.2 Activation – “invalid object id” (1821/0x71d)

If the ADS Return Code 1821 / 0x71d is reported during the course of the start, check the context of the module instance as described in Quick start [▶ 71].

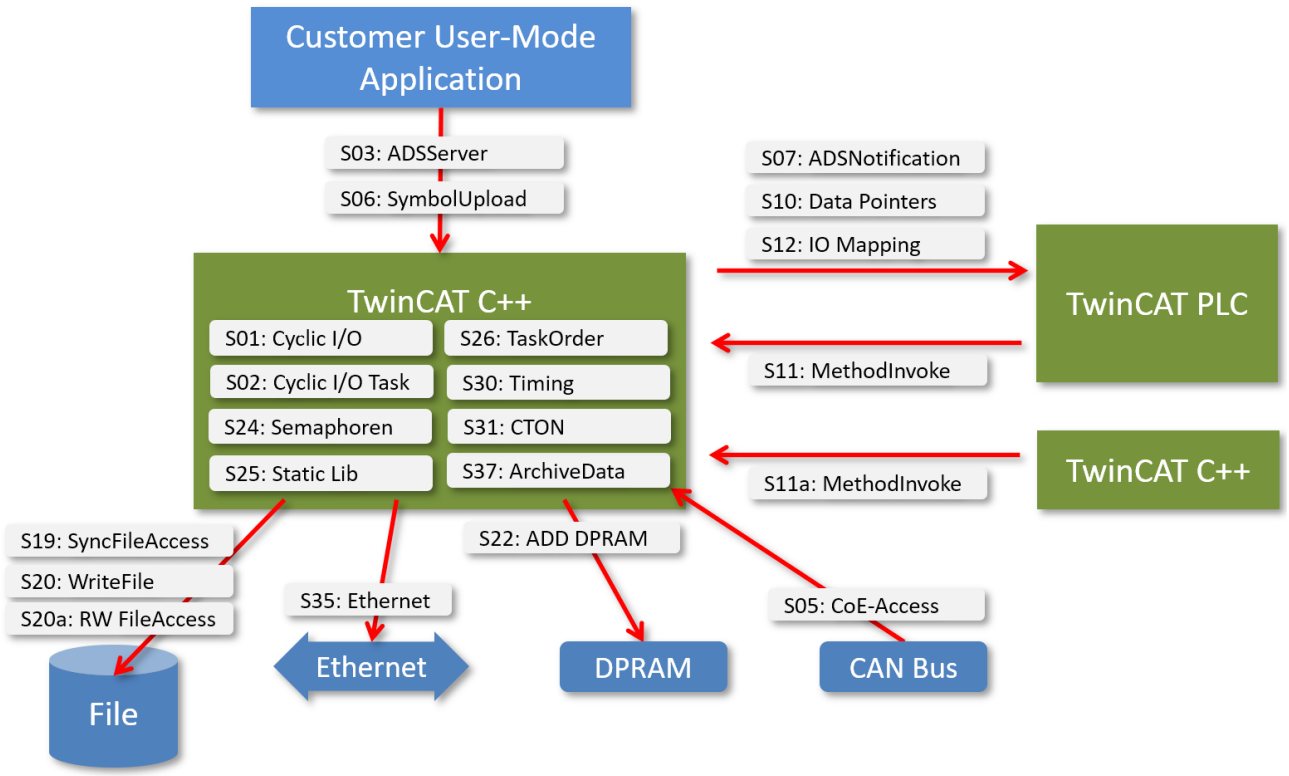


15 C++-samples

Sample code and configurations for this product can be obtained from the corresponding repository on GitHub: https://github.com/Beckhoff/TC1300_Samples. There you have the option to clone the repository or download a ZIP file containing the sample.



This picture provides an overview in graphical form and places the emphasis on the interaction possibilities of a C++ module.



Beyond that, this is a table with brief descriptions of the samples.

Number	Title	Description
01	Sample01: Cyclic module with IO [▶ 236]	This article describes the implementation of a TwinCAT 3 C++ module that uses an IO module mapped with physical IO. This sample describes the quick start for the purpose of creating a C++ module that increments a counter at each cycle and assigns the counter to the logical output "Value" in the data area. The data area can be assigned to the physical IO or another logical input of another module instance.
02	Sample02: Cyclic with IO Task [▶ 237]	Describes the flexibility of C++ code when working with IOs that are configured at the task. Thanks to this approach, a finally compiled C++ module can affect various IOs connected with the IO task much more flexibly. One application could be to check cyclic analog input channels, where the number of input channels can differ from one project to another.
03	Sample03: ADS Server Client [▶ 237]	Describes the design and implementation of one's own ADS interface in a C++ module. The sample contains two parts: • ADS server implemented in TwinCAT 3 C++ with user-specific ADS interface, ADS Client UI implemented in C#, which transmits user-specific ADS messages to the ADS server.
05	Sample05: CoE access via ADS [▶ 246]	Shows how CoE registers of EtherCAT devices can be accessed over ADS.
06	Sample06: ADS C# client uploads ADS symbols [▶ 247]	Shows how symbols in an ADS server can be accessed via the ADS interface. C# ADS client connects to a module implemented in PLC/C++/ Matlab®. Upload the available symbol information and read/write subscription for process values.
07	Sample07: Receiving ADS Notifications [▶ 252]	Describes the implementation of a TwinCAT 3 C++ module that receives ADS notifications regarding data changes on other modules.

Number	Title	Description
08	Sample08: Provision of ADS-RPC [▶ 253]	Describes the implementation of methods that can be called by ADS via the task.
10	Sample10: Module communication: Use of data pointers [▶ 256]	Describes the interaction between two C++ modules with a direct data pointer. The two modules must be implemented on the same CPU core in the same real-time context.
11	Sample11: Module communication: Method call PLC module to C++ module [▶ 257]	This sample contains two parts: • A C++ module which functions as a state machine that provides an interface with methods for starting/stopping and also for setting/ maintaining the state machine. Second PLC module for interacting with the first module by calling methods from the C++ module.
11a	Sample11a: Module communication: Method call C++ module to C++ module [▶ 284]	This sample contains two classes in one driver (can also be done between two drivers). • One module that provides a calculation method. Access is protected through a CriticalSection. A second module that acts as the caller to use the methods in the other module.
12	Sample12: Module communication: Using IO mapping [▶ 285]	Describes how two modules can interact with each other via mapping of symbols from the data area of different modules. The two modules can be executed on the same or different CPU cores.
13	Sample13: Module communication: Method call C++ module to PLC module [▶ 286]	Describes how a TwinCAT 3 C++ module calls a PLC function block using TcCOM interface methods.
19	Sample19: Synchronous File Access [▶ 289]	Describes how the file IO functionality can be used in a C++ module in a synchronous manner. The sample writes process values to a file. Writing to the file is triggered by a deterministic cycle - the execution of file IO is decoupled (asynchronous), i.e. the deterministic cycle continues and is not hindered by writing to the file. The status of the routine for decoupled writing to the file can be checked.
20	Sample20: FileIO-Write [▶ 290]	Describes how the file IO functionality can be used with the C++ module. The sample writes process values to a file. Writing to the file is triggered by a deterministic cycle - the execution of file IO is decoupled (asynchronous), i.e. the deterministic cycle continues and is not hindered by writing to the file. The status of the routine for decoupled writing to the file can be checked.
20a	Sample20a: FileIO-Cyclic Read / Write [▶ 290]	A more extensive sample than S20 and S19. It describes the cyclic read and/or write access to files from a C++ module.
22	Sample22: Automation Device Driver (ADD): Access DPRAM [▶ 292]	Describes how the TwinCAT Automation Device Driver (ADD) is to be written for access to the DPRAM.

Number	Title	Description
23	Sample23: Structured Exception Handling (SEH) [▶ 293]	Describes the use of Structured Exception Handling (SEH) based on five variants.
24	Sample24: Semaphores [▶ 295]	Describes the use of semaphores.
25	Sample25: Static Library [▶ 296]	Describes how to use the C++ static library contained in another C++ module.
26	Sample26: Order of execution in a task [▶ 298]	Describes the determination of the task execution order, if a task is assigned to more than one module.
30	Sample30: Timing Measurement [▶ 300]	Describes the measurement of the C++ cycle or execution time.
31	Sample31: Functionblock TON in TwinCAT3 C++ [▶ 301]	Describes the implementation of a behavior in C++, which is comparable to a TON function block of PLC / 61131.
37	Sample37: Archive data [▶ 303]	Describes the loading and saving of the state of an object during the initialization and de-initialization.
TcCOM	TcCOM samples [▶ 304]	Several samples are provided to illustrate the module communication between PLC and C++.

15.1 Sample01: Cyclic module with IO

This article describes how to implement a TwinCAT 3 C++ module which is using the module IO mapped to physical IO.

Download

You can obtain the source code for this sample here:
https://github.com/Beckhoff/TC1300_Samples/tree/main/S01-CyclicIO

1. Using a Visual Studio with TwinCAT installed, open the project via **Open Project ....**
2. If you want to run the project on Windows or TwinCAT/BSD® (not required for Beckhoff RT Linux®): Configure signing for this project by right-clicking on the project on the page, selecting **Project -> Properties -> TC Sign**, enabling TwinCAT signing, and configuring your certificate and password if necessary.
   Further information on signing C++ projects can be found in the chapter TwinCAT [▶ 21].
3. Select your target system.
4. Build the sample (e.g. **Build->Build Solution**).
5. Activate the configuration by clicking .

⇨ The sample is ready for operation.

Description

This sample describes the quick start for the purpose of creating a C++ module that increments a counter on each cycle and assigns the counter to the logical output Value in the data area.
The data area can be assigned to the physical IO or another logical input or another module instance.

The sample is described step by step in the short instructions [▶ 58].

15.2 Sample02: Cyclic C++ logic, which uses IO from the IO Task

This article describes the implementation of a TwinCAT 3 C++ module that uses an image of an IO Task.

Download

Here you can access the source code for this sample:

https://github.com/Beckhoff/TC1300_Samples/tree/main/S02-CyclicIOTask

1. Using a Visual Studio with TwinCAT installed, open the project via **Open Project ....**
2. If you want to run the project on Windows or TwinCAT/BSD® (not required for Beckhoff RT Linux®): Configure signing for this project by right-clicking on the project on the page, selecting **Project -> Properties -> TC Sign**, enabling TwinCAT signing, and configuring your certificate and password if necessary.
   Further information on signing C++ projects can be found in the chapter TwinCAT [▶ 21].
3. Select your target system.
4. Build the sample (e.g. **Build->Build Solution**).
5. Activate the configuration by clicking  .

⇨ The sample is ready for operation.

Source code that is not automatically generated by the wizard is marked with a start flag "//sample code" and end flag "//sample code end".
This allows you to search for these character strings in the files to get an idea of the details.

Description

This sample describes the flexibility of C++ code when working with IOs configured at the task. This approach enables a compiled C++ module to respond more flexibly, if a different number of IOs are linked to the IO task. One application option would be cyclic testing of analog input channels with a different number of channels, depending on the project.

The sample contains

- the C++ module TcIoTaskImageAccessDrv with a module instance TcIoTaskIMageAccessDrv_Obj1
- A "Task1" with an image, 10 input variables (Var1..Var10) and 10 output variables (Var11..Var20).
- They are linked: The instance is called by the task and uses the image of Task1.

The C++ code accesses the values via a data image, which is initialized during the transition from SAFEOP to OP (SO).

In the cyclically executed method CycleUpdate the value of each input variable is checked by calling the helper method CheckValue. If it is less than 0, the corresponding output variable is set to 1, if it is greater than 0, it is set to 2, if it is 0, the output is set to 3.

After activation of the configuration you can access the variables via the Solution Explorer and set them. Double-click on the Task1 image of system for an overview.
The input variables can be opened and then set with the **Online** tab.

15.3 Sample03: C++ as ADS server

This article describes:

- The creation of a TwinCAT 3 C++ module that acts as an ADS server.
  The server provides an ADS interface for starting / stopping / resetting a counter variable in the C++ module. The counter is available as a module output and can be assigned to an output terminal

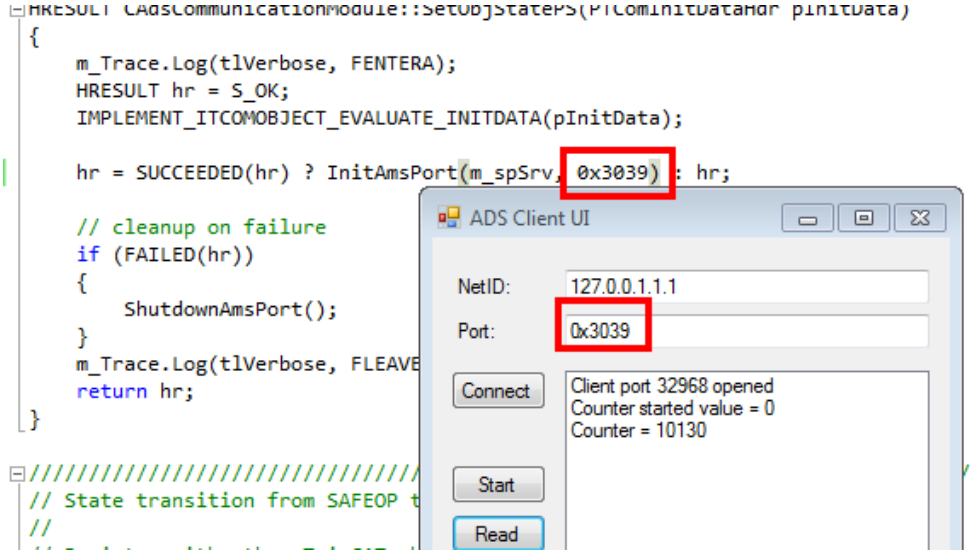
(analog or a number of digital IOs).
How the TC3 ADS server function written in C++ is to be implemented. [▶ 238]

- The creation of a C# ADS client to interact with the C++ ADS server.
  The client provides a UI for connection locally or via a network to an ADS server with the ADS interface to be counted. The UI enables the starting / stopping / reading / overwriting and resetting of the counter.
  Sample code: ADS Client UI written in C# [▶ 242].

Understanding the sample

Options for the automatic determination of an ADS port are used in the sample. The disadvantage of this is that the client has to be configured at each start in order to access the correct ADS port.

Alternatively, the ADS port can be hard-coded in the module as shown below.
Disadvantage here: The C++ module cannot be instanced more than once as it is not possible to share an ADS port.



15.3.1 Sample03: TC3 ADS Server written in C++

This article describes how to create a TC3-C++ module acting as a ADS-server.
The server will provide an ADS interface to start / stop / reset an counter variable insight the C++ module.

Download

Here you can access the source code for this sample:

https://github.com/Beckhoff/TC1300_Samples/tree/main/S03-ADSServer

.

1. Using a Visual Studio with TwinCAT installed, open the project via **Open Project ....**
2. If you want to run the project on Windows or TwinCAT/BSD® (not required for Beckhoff RT Linux®): Configure signing for this project by right-clicking on the project on the page, selecting **Project -> Properties -> TC Sign**, enabling TwinCAT signing, and configuring your certificate and password if necessary.
   Further information on signing C++ projects can be found in the chapter TwinCAT [▶ 21].
3. Select your target system.
4. Build the sample (e.g. **Build->Build Solution**).
5. Activate the configuration by clicking  .

⇨ The sample is ready for operation.

Description

This sample contains a C++ module that acts as an ADS server. The server grants access to a counter that can be started, stopped and read.

The header file of the module defines the counter variable m_bCount, and the corresponding .cpp file initializes the value in the constructor and implements the logic in the CycleUpdate method.

The AdsReadWriteInd method in the .cpp file analyzes the incoming messages and returns the return values. A define in the header file is added for a further added message type.

Details such as the definition of the ADS message types are described in the following cookbook, where you can compile the sample manually.

Cookbook

This is a step by step description about the creation of the C++ module.

1. Create a new TwinCAT 3 project solution

Follow the steps for creating a new TwinCAT 3 project [▶ 58].

2. Create a C++ project with ADS port

Follow the steps for the creation of a new TwinCAT 3 C++ project [▶ 59].

Select **TwinCAT Module Class with ADS port** in the **Class templates** dialog.

3. Add the sample logic to the project

1. Open the header file <MyClass>.h (in this sample Module1.h) and add the counter m_bCount to the protected area as a new member variable:

```
class CModule1
    : public ITComObject
    , public ITcCyclic
    ,...
{
public:
    DECLARE_IUNKNOWN()
    ....
protected:
    DECLARE_ITCOMOBJECT_SETSTATE();
///<AutoGeneratedContent id="Members">
    ITcCyclicCallerInfoPtr m_spCyclicCaller;
    .....
///</AutoGeneratedContent>
    ULONG m_ReadByOidAndPid;
    BOOL m_bCount;
};
```

2. Open the class file <MyClass>.cpp (in this sample Module1.cpp) and initialize the new values in the constructor:

```
CModule1::CModule1()
    .....
{
    memset(&m_Counter, 0, sizeof(m_Counter));
    memset(&m_Inputs, 0, sizeof(m_Inputs));
    memset(&m_Outputs, 0, sizeof(m_Outputs));
    m_bCount = FALSE; // by default the counter should not increment
    m_Counter = 0;    // we also initialize this existing counter
}
```

⇨ The sample code has been added.

3.a. Add the sample logic to the ADS server interface.

Usually, the ADS server receives an ADS message, which contains two parameters (indexGroup and indexOffset) and perhaps further pData.

Designing an ADS interface
Our counter is to be started, stopped, reset, overwritten with a value or send a value to the ADS client on request:

indexGroup	indexOffset	Description
0x01	0x01	m_bCount = TRUE, counter is incremented.
0x01	0x02	Counter value is transferred to ADS client.
0x02	0x01	m_bCount = FALSE, counter is no longer incremented.
0x02	0x02	Reset counter.
0x03	0x01	Overwrite counter with value transferred by ADS client.

These parameters are defined in modules1Ads.h – change the source code to add a new command for IG_RESET.

```
#include "TcDef.h"
enum Module1IndexGroups : ULONG
{
Module1IndexGroup1 = 0x00000001,
Module1IndexGroup2 = 0x00000002, // add command
IG_OVERWRITE = 0x00000003 // and new command
};

enum Module1IndexOffsets : ULONG
{
Module1IndexOffset1 = 0x00000001,
Module1IndexOffset2 = 0x00000002
};
```

Change the source code in your <MyClass>::AdsReadWriteInd() method (in this case in Module1.cpp).

```
switch(indexGroup)
{
case Module1IndexGroup1:
switch(indexOffset)
{
case Module1IndexOffset1:
...
// TODO: add custom code here
m_bCount = TRUE; // receivedIG=1 IO=1, start counter
AdsReadWriteRes(rAddr, invokeId, ADSERR_NOERR, 0,NULL);
break;
case Module1IndexOffset2:
...
// TODO: add custom code here
// map counter to data pointer
pData = &m_Counter; // received IG=1 IO=2, provide counter value via ADS
AdsReadWriteRes(rAddr, invokeId, ADSERR_NOERR, 4 ,pData);
//comment this: AdsReadWriteRes(rAddr, invokeId,ADSERR_NOERR, 0, NULL);
break;
}
break;
case Module1IndexGroup2:
switch(indexOffset)
{
case Module1IndexOffset1:
...
// TODO: add custom code here
// Stop incrementing counter
m_bCount = FALSE;
// map counter to data pointer
pData = &m_Counter;
AdsReadWriteRes(rAddr, invokeId, ADSERR_NOERR, 4,pData);
break;
case Module1IndexOffset2:
```

```
 ...
// TODO: add custom code here
// Reset counter
m_Counter = 0;
// map counter to data pointer
pData = &m_Counter;
AdsReadWriteRes(rAddr, invokeId, ADSERR_NOERR, 4, pData);
break;
}
break;
case IG_OVERWRITE:
switch(indexOffset)
{
case Module1IndexOffset1:
...
// TODO: add custom code here // override counter with value provided by ADS-client
unsigned long *pCounter = (unsigned long*) pData;
m_Counter = *pCounter;
AdsReadWriteRes(rAddr, invokeId, ADSERR_NOERR, 4, pData);
break;
}
break;
}
break;
default:
__super::AdsReadWriteInd(rAddr, invokeId, indexGroup,indexOffset, cbReadLength, cbWriteLength,
pData;
break;
}
```

3.b. Insert the sample logic into the cyclic part

The <MyClass>::CycleUpdate() method is cyclically called – this is the point where the logic is to be changed.

```
// TODO: Replace the sample with your cyclic code
m_Counter+=m_Inputs.Value; // replace this line
m_Outputs.Value=m_Counter;
```

In this case the counter **mCounter** is incremented if the boolean variable **m_bCount** is TRUE.

Insert this if-case into your cyclic method.

```
HRESULT CModule1::CycleUpdate(ITcTask* ipTask,
ITcUnknown* ipCaller, ULONG context)
{
HRESULT hr = S_OK;
// handle pending ADS indications and confirmations
CheckOrders();
....
// TODO: Replace the sample with your cyclic code
if (m_bCount) // new part
{
m_Counter++;
}
m_Outputs.Value=m_Counter;
}
```

4. Execute server sample

1. Run the TwinCAT TMC Code Generator in order to provide the inputs/outputs for the module.
2. Save the project.
3. Compile the project.
4. Create a module instance.
5. Create a cyclic task and configure the C++ module for the execution in this context.
6. Scan the hardware IO and assign the symbol Value of outputs to certain output terminals (this is optional).
7. Activate [▶ 73] the TwinCAT project.

⇨ The sample is ready for operation.

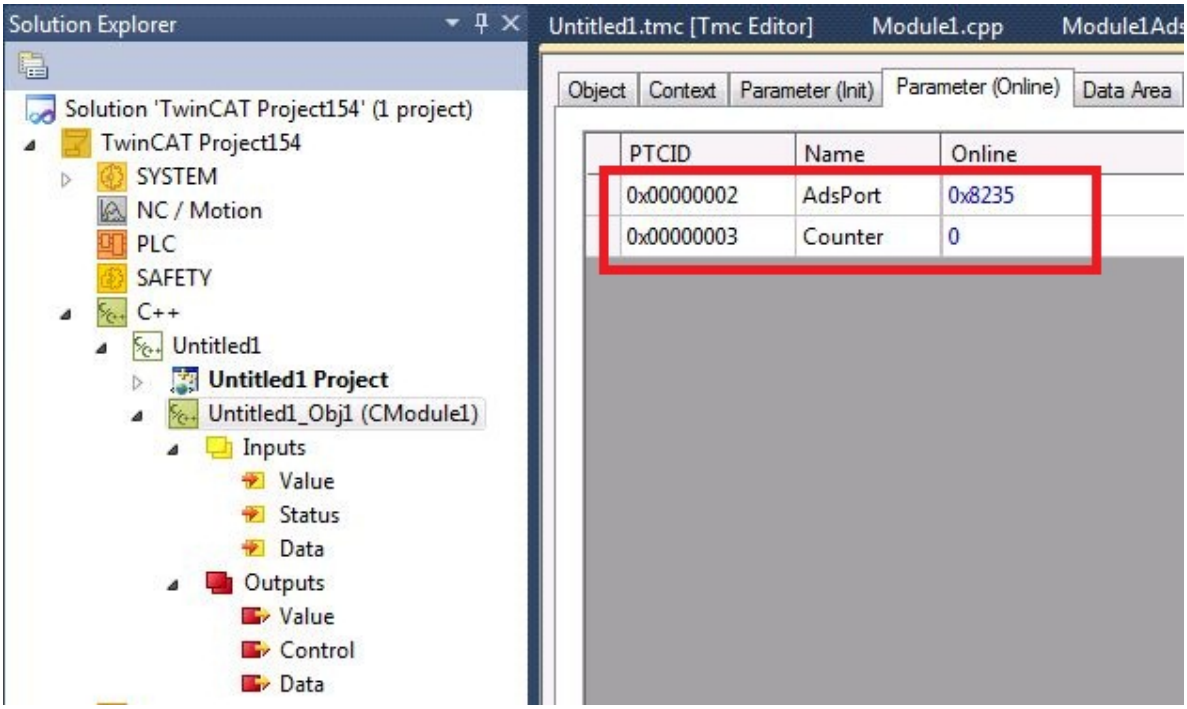
5. Determine the ADS port of the module instance

Generally the ADS port may be

- pre-numbered, so that the same port is always used for this module instance.
- kept customizable, in order to offer several module instances the option to have their own ADS port assigned on startup of the TwinCAT system.

In this sample the default setting (keep flexible) is selected. First of all you have to determine the ADS port that was assigned to the module that has just been activated.

1. Navigate to the module instance.
2. Select the **Parameter Online** tab.
   ⇨ 0x8235 or decimal 33333 is assigned to the ADS port (this may be different in your sample). If more and more instances are created, each instance is allocated its own unique AdsPort.
   ⇨ The counter is still at "0" because the ADS message to start the incrementation has not been sent.



⇨ The server part is completed - continue with ADS client sends the ADS messages [▶ 242].

15.3.2 Sample03: ADS client UI in C#

This article describes the ADS client, which sends ADS messages to the previously described ADS server.

The implementation of the ADS server depends neither on the language (C++ / C# / PLC / ...) nor on the TwinCAT version (TwinCAT 2 or TwinCAT 3).

Download

Here you can access the source code for this sample:

https://github.com/Beckhoff/TC1300_Samples/tree/main/S03-ADSClient

✓ This code requires .NET Framework

1. Open the sln file contained in it with Visual Studio.
2. Create the sample on your local machine (right-click on the project and click on **Build**).
3. Start the program with a right-click on **Project**, **Debug->Start new instance**.

Description

This client performs two tasks:

- Testing the ADS-server, which was described before.
- Providing sample code for implementing a ADS-client

Using the client

Selecting a communication partner

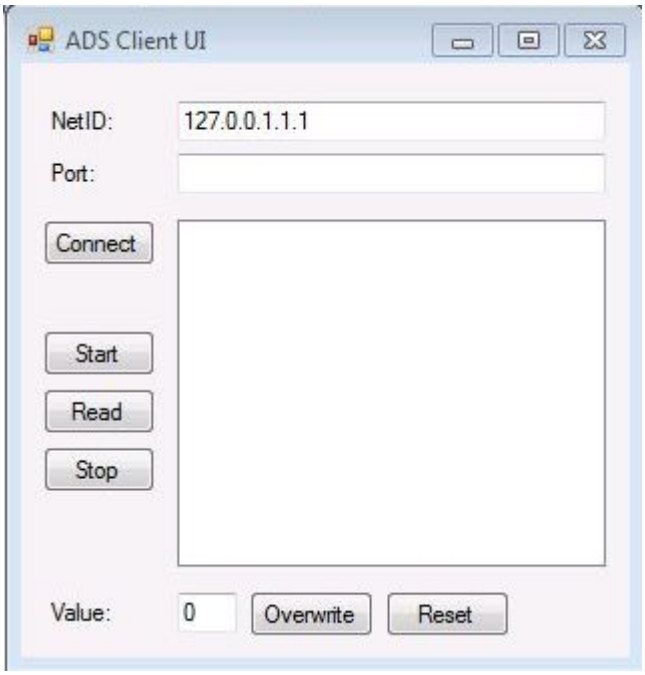
Enter both ADS parameters in order to specify your ADS communication partner:

- NetID:
  127.0.0.1.1.1 (for ADS partners also connected to local ADS Message Router)
  Enter a different NetID if you want to communicate via the network with an ADS partner connected to another ADS router.
  You must first establish an ADS route between your device and the remote device once.

- AdsPort
  Enter the AdsServerPort of your communication partner.
  Do not confuse the ADS server port (which has explicitly implemented your own message handler) with the regular ADS port for the purpose of accessing symbols (this is provided automatically, without the need for user intervention).
  Find the assigned AdsPort [▶ 238], in this sample the AdsPort was 0x8235 (dec 33333).

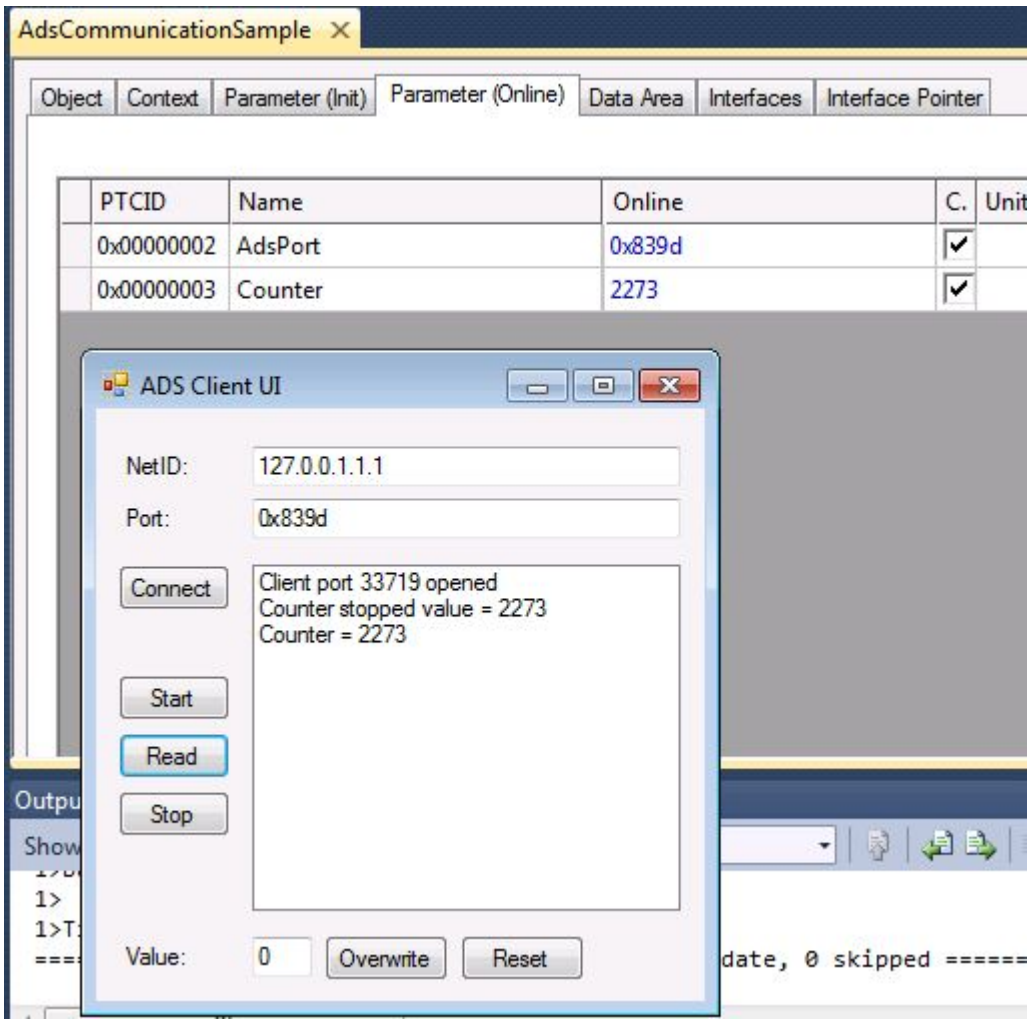
Create link with communication partner

Click on **Connect** to call the method TcAdsClient.Connect for the purpose of creating a link with the configured port.



The **Start** / **Read** / **Stop** / **Overwrite** / **Reset** buttons are used to send ADS messages to the ADS server. The specific indexGroup / indexOffset commands have already been designed in the ADS interface of the ADS server [▶ 238].

The result of clicking on the command buttons can also be seen in the module instance in the **Parameters (online)** tab.



C# program

Here is the "Core" code of the ADS client – download for the GUI or ZIP file above.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using TwinCAT.Ads;

namespace adsClientVisu
{
public partial class form : Form
{
public form()
{
InitializeComponent();
}

private void Form1_Load(object sender, EventArgs e)
{
// create a new TcClient instance
_tcClient = new TcAdsClient();
adsReadStream = new AdsStream(4);
adsWriteStream = new AdsStream(4);
}

/*
* Connect the client to the local AMS router
*/

private void btConnect_Click(object sender, EventArgs e)
```

```
{
AmsAddress serverAddress = null;
try
{
serverAddress = new AmsAddress(tbNetId.Text,
Int32.Parse(tbPort.Text));
}
catch
{
MessageBox.Show("Invalid AMS NetId or Ams port");
return;
}

try
{
_tcClient.Connect(serverAddress.NetId, serverAddress.Port);
lbOutput.Items.Add("Client port " + _tcClient.ClientPort + " opened");

}
catch
{
MessageBox.Show("Could not connect client");
}
}

private void btStart_Click(object sender, EventArgs e)
{
try
{
_tcClient.ReadWrite(0x1, 0x1, adsReadStream, adsWriteStream);
byte[] dataBuffer = adsReadStream.ToArray();
lbOutput.Items.Add("Counter started value = " + BitConverter.ToInt32(dataBuffer, 0));
}

catch (Exception err)
{
MessageBox.Show(err.Message);
}

}

private void btRead_Click(object sender, EventArgs e)
{
try
{
_tcClient.ReadWrite(0x1, 0x2, adsReadStream, adsWriteStream);
byte[] dataBuffer = adsReadStream.ToArray();
lbOutput.Items.Add("Counter = " + BitConverter.ToInt32(dataBuffer, 0));
}

catch (Exception err)
{
MessageBox.Show(err.Message);
}
}

private void btStop_Click(object sender, EventArgs e)
{
try
{
_tcClient.ReadWrite(0x2, 0x1, adsReadStream, adsWriteStream);
byte[] dataBuffer = adsReadStream.ToArray();
lbOutput.Items.Add("Counter stopped value = " + BitConverter.ToInt32(dataBuffer, 0));
}

catch (Exception err)
{
MessageBox.Show(err.Message);
}
}

private void btReset_Click(object sender, EventArgs e)
{
try
{
_tcClient.ReadWrite(0x2, 0x2, adsReadStream, adsWriteStream);
byte[] dataBuffer = adsReadStream.ToArray();
lbOutput.Items.Add("Counter reset Value = " + BitConverter.ToInt32(dataBuffer, 0));
}
```

```
catch (Exception err)
{
MessageBox.Show(err.Message);
}
}
}
}
```

15.4 Sample05: C++ CoE access via ADS

This article describes how to implement a TwinCAT 3 C++ modules which can access the CoE (CANopen over EtherCAT) register of a EtherCAT terminal.

Download

Here you can access the source code for this sample:

https://github.com/Beckhoff/TC1300_Samples/tree/main/S05-CoEAccess

1. Open the zip file that it contains in TwinCAT 3 by clicking on **Open Project …**.
2. Select your target system.
3. Build the sample on your local machine (e.g. **Build->Build Solution**).
4. Note the actions listed on this page under **Configuration**.
5. Activate the configuration by clicking .

⇨ The sample is ready for operation.

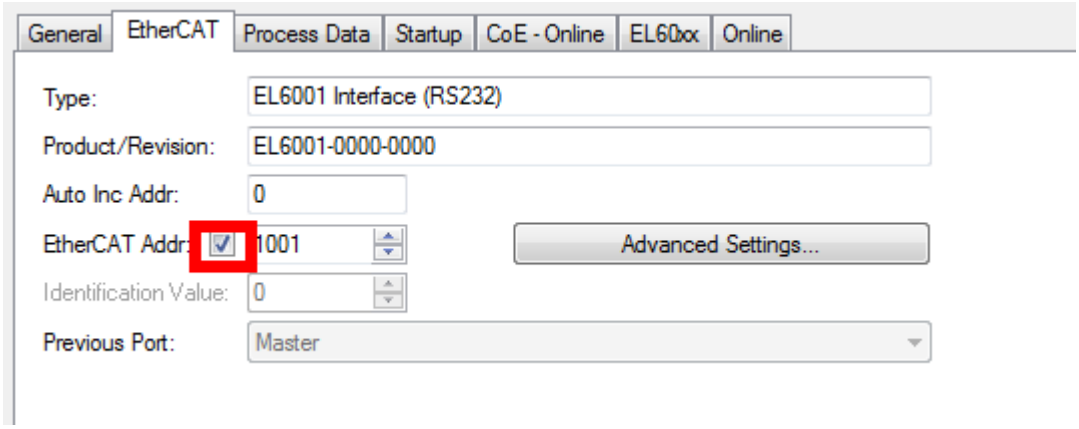
Description

This sample describes access to an EtherCAT Terminal, which reads the manufacturer ID and specifies the baud rate for serial communication.

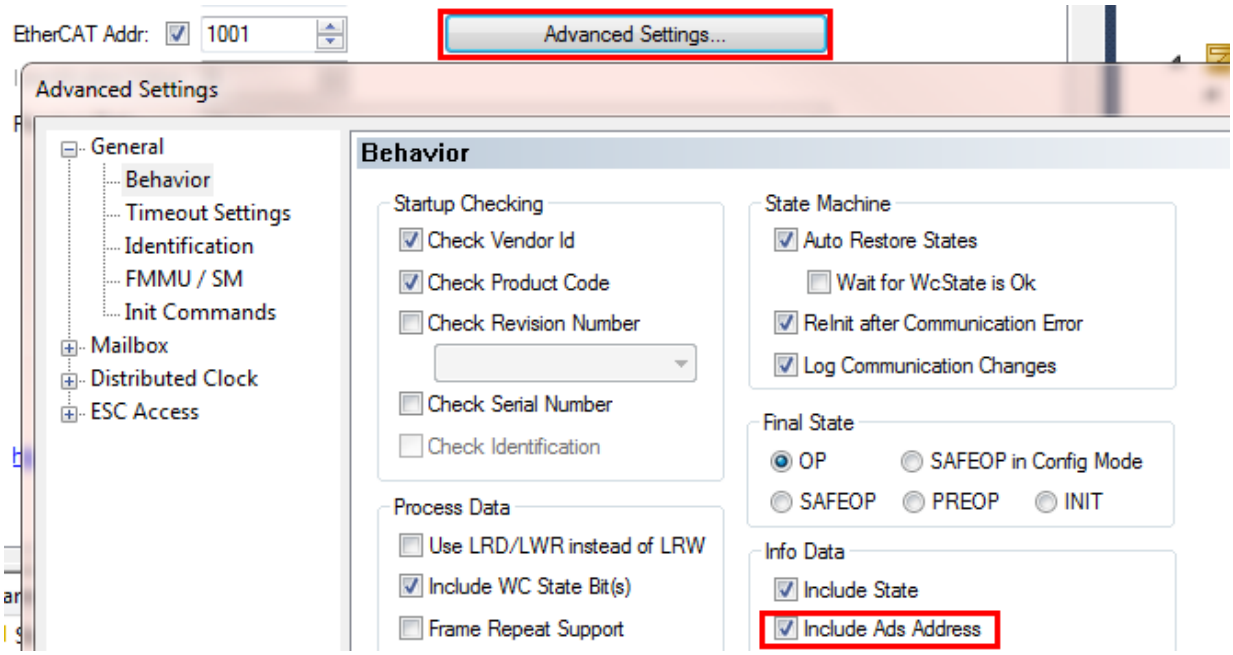
This sample describes the quick start for the purpose of creating a C++ module that increments a counter on each cycle and assigns the counter to the logical output Value in the data area.

Configuration

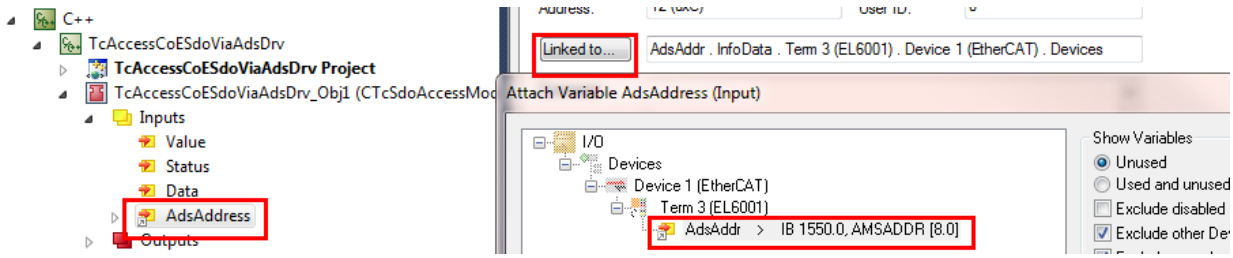
1. Activate the EtherCAT address of the terminal concerned and assign it.



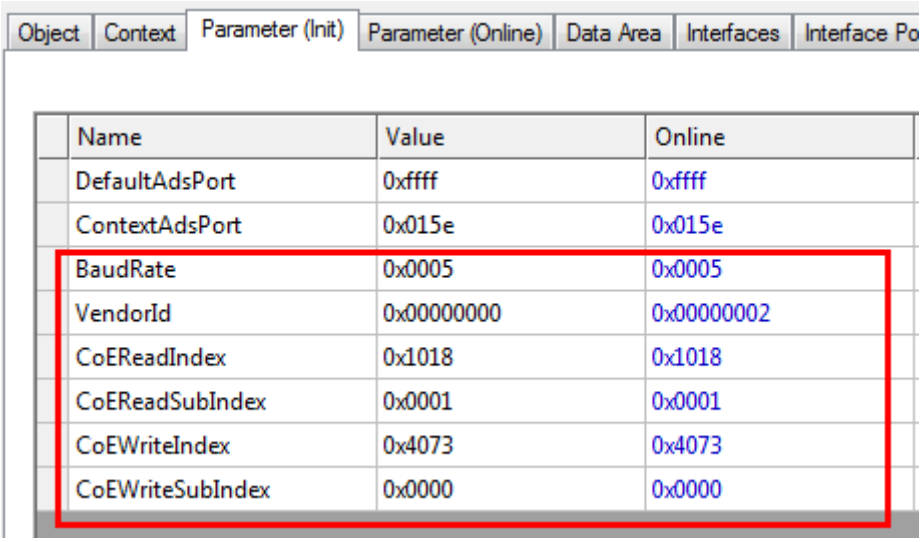
2. Activate inclusion of the ADS address in the advanced settings for the EtherCAT Terminal:



3. Assign the ADS address (including netId and port) to the module input AdsAdress:



4. The module parameters are read out and displayed by the sample code during the course of the initialization:



Name	Value	Online
DefaultAdsPort	0xffff	0xffff
ContextAdsPort	0x015e	0x015e
BaudRate	0x0005	0x0005
VendorId	0x00000000	0x00000002
CoEReadIndex	0x1018	0x1018
CoEReadSubIndex	0x0001	0x0001
CoEWriteIndex	0x4073	0x4073
CoEWriteSubIndex	0x0000	0x0000

15.5 Sample06: UI-C#-ADS client uploading the symbolic from module

This article describes the implementation of an ADS client to

- connect to an ADS server that provides a process image (data area); the connection can be established locally or remotely via a network,
- upload symbol information,

- read / write data synchronously,
- subscribe to symbols, in order to obtain values on change as callback.

Download

Here you can access the source code for this sample:

https://github.com/Beckhoff/TC1300_Samples/tree/main/S06-SymbolUploadClient

✓ This code requires .NET Framework 3.5 or higher!

1. Open the sln file contained in it with Visual Studio.
2. Create the sample on your local machine (right-click on the project and click on "Build").
3. Start the program with a right-click on **Project**, **Debug**->**Start new instance**.

The client sample should be used with Sample 03 "C++ as ADS server".

Open Sample 03 [▶ 238] before you begin with this client-side sample!

Description

The possibilities of the ADS are described on the basis of this sample.

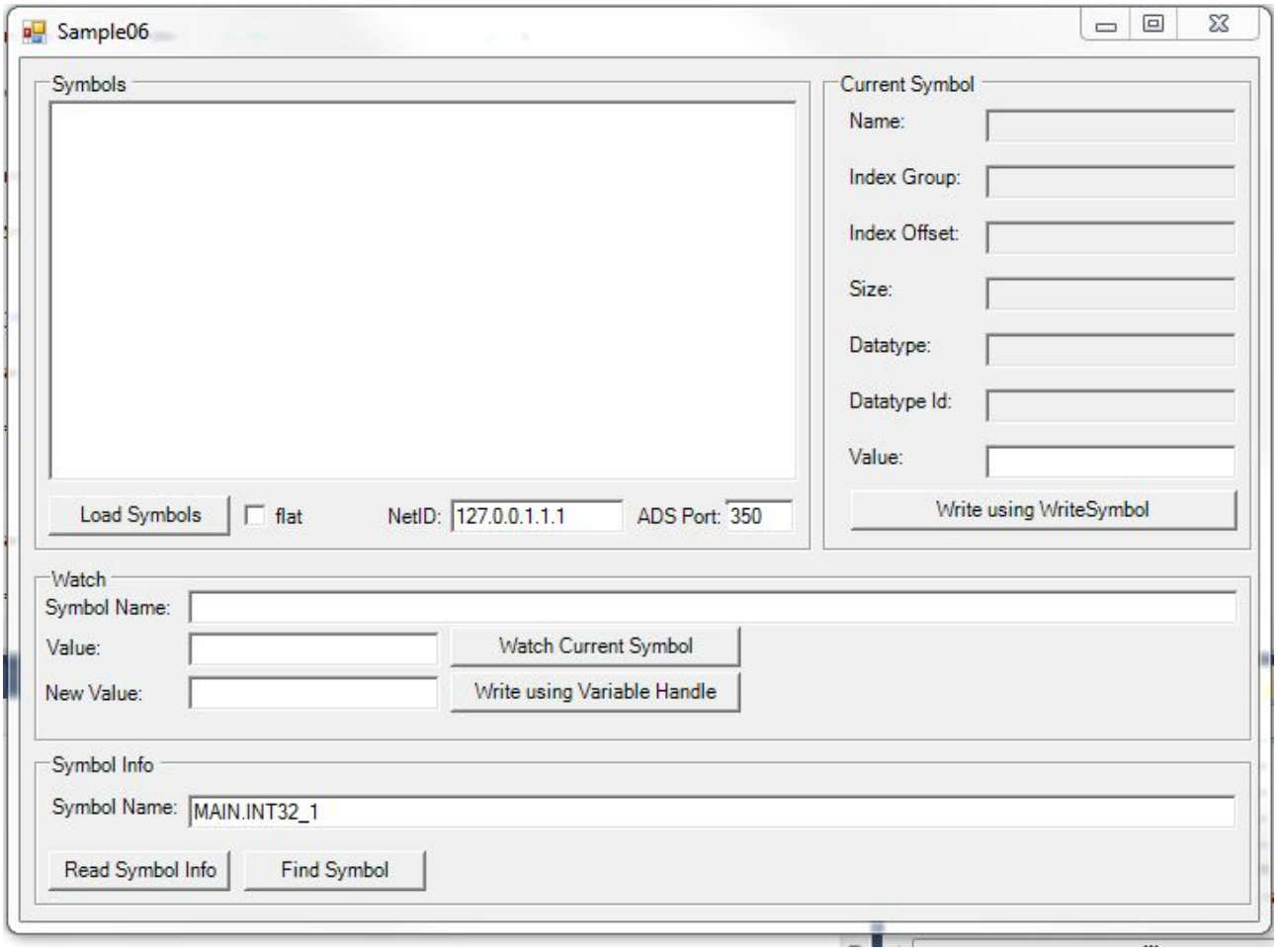
The details of the implementation are described in Form1.cs, which is included in the download. The connection via ADS with the target system is established in the btnLoad_Click method, which is called on clicking on the **Load Symbols** button. From there you can explore the different GUI functions.

Background information:

For this ADS client it is irrelevant whether the ADS server is based on TwinCAT 2 or TwinCAT 3. It also doesn't matter if the server is a C++ module, a PLC module or an IO task without any logic.

The ADS client UI

On starting of the sample the user interface (UI) is displayed.



Selecting a communication partner

After starting the client, enter the two ADS parameters to determine your ADS communication partner.

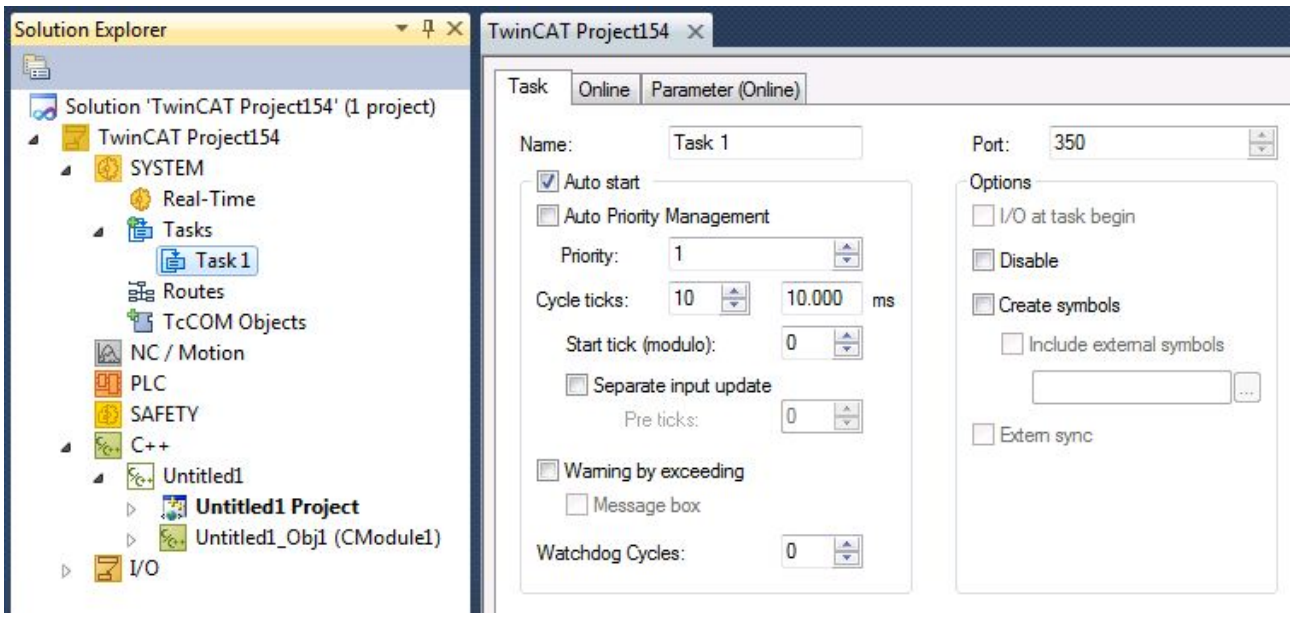
- NetID:
  127.0.0.1.1.1 (for ADS partner also linked with local ADS message router).
  Enter another NetID, if you want to communicate with an ADS partner connected to another ADS router via the network.
  First you have to create an ADS route between your device and the remote device.
- AdsPort
  Enter the AdsPort of your communication partner: 350 (in this sample).

Do not confuse the ADS server port with the regular ADS port.

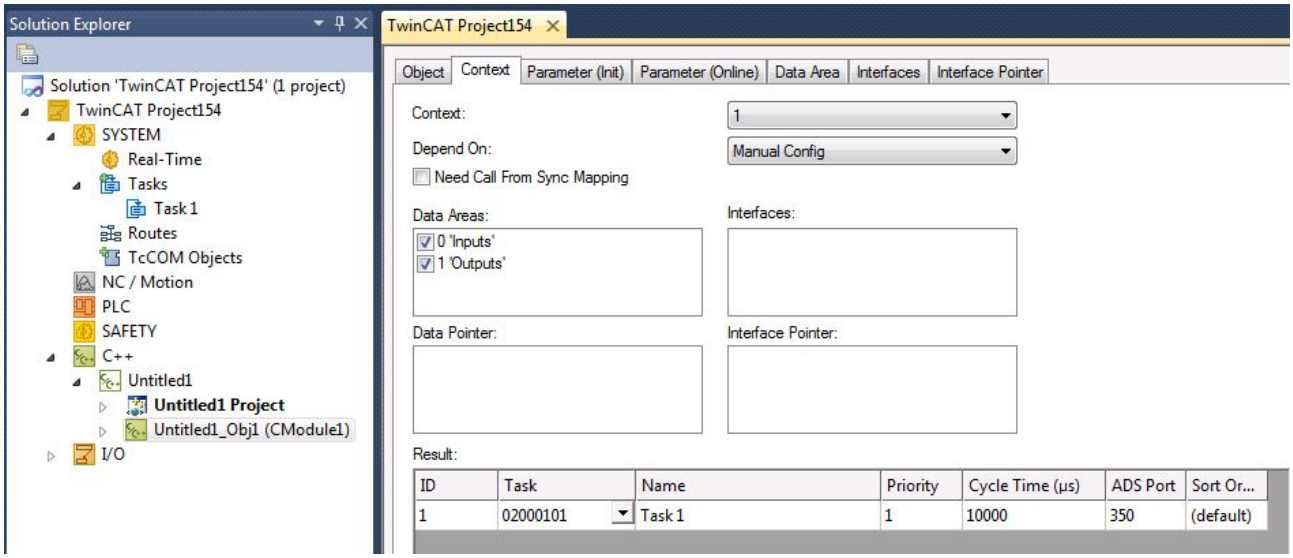
Do not confuse the ADS server port (which was explicitly implemented in Sample 03 for providing your own message handler) with the regular ADS port for the purpose of access to symbols (this is provided automatically, without the need for user intervention):

The regular ADS port is required to access symbols. You can find the AdsPort for the IO task of your instance or the module instance yourself (since the module is executed in the context of the IO task).

Navigate to IO task Task1 and note the value of the port: 350.



Since the C++ module instance is executed in the context of Task1, the ADS port is also 350.

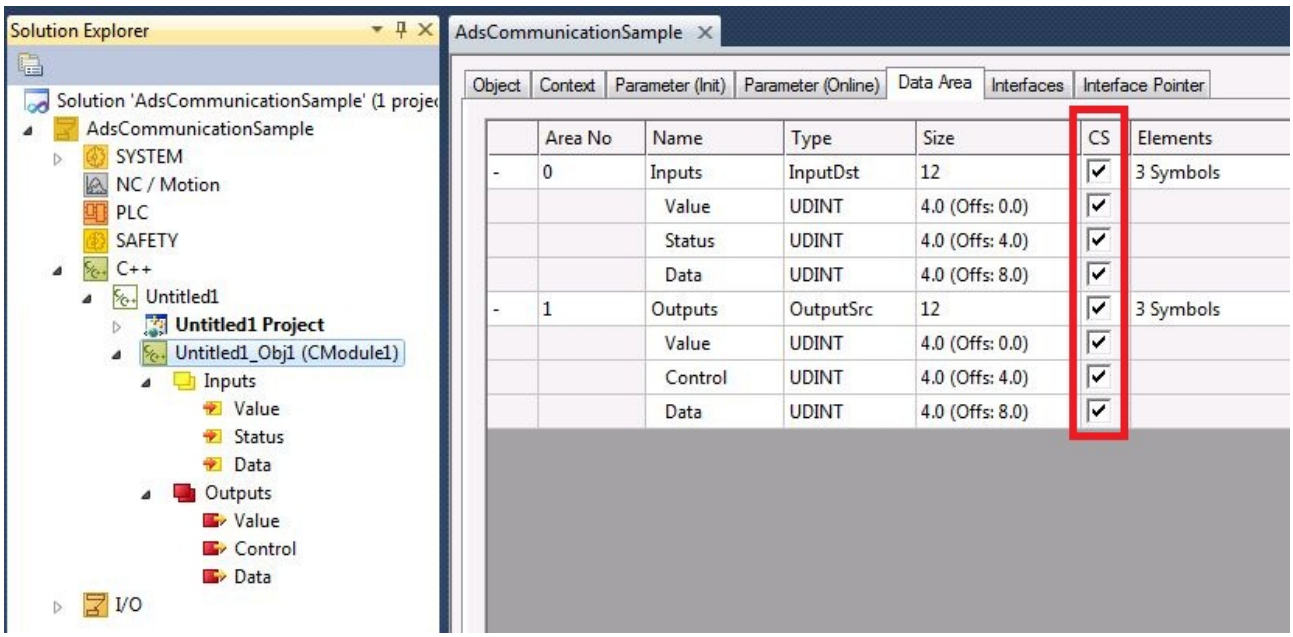


Enabled symbols for access available via ADS

It is possible to provide or not provide individual symbols or even complete data areas for access via ADS. Navigate to the **Data Area** tab of your instance and enable/disable the **C/S** column.

In this sample all symbols are marked and therefore available for ADS access.

After making the changes, click on **Activate configuration**.



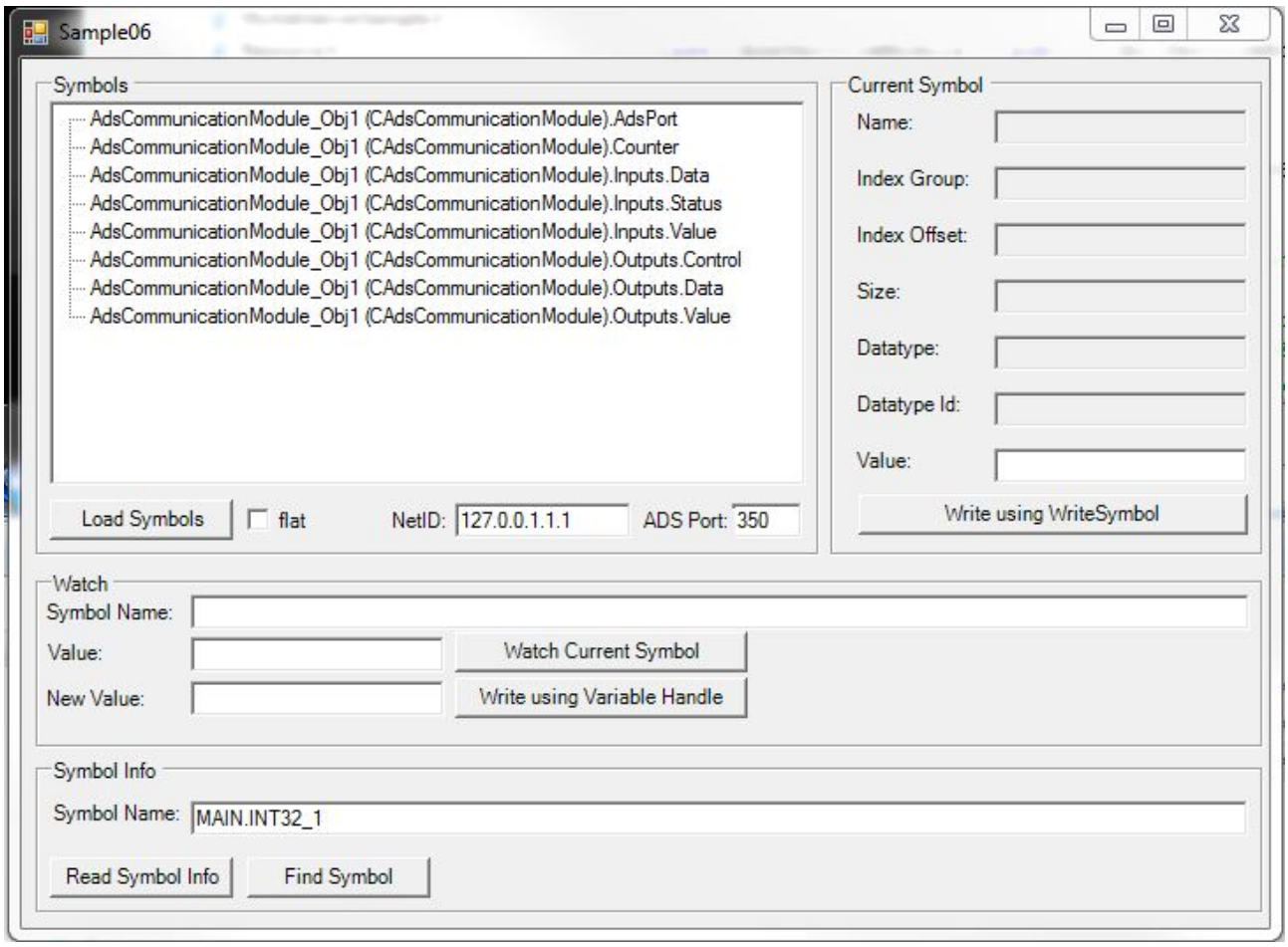
Load symbols

Once the NetID and the ADS port have been set up, click on the **Load Symbols** button to make a connection with the target system and load the symbols.

All available symbols are then visible. You can then:

- Write a new value:
  Select a symbol in the left-hand tree, e.g. **Counter**.
  Enter a new value in the **Value** edit field on the right-hand side and click **Write using WriteSymbol**.
  The new value is written to the ADS server.

  After writing a new value with **Write using WriteSymbol**, the C# application also receives a Callback with the new value.
- Subscribe to receive a Callback when values change:
  select an icon in the left-hand tree, e.g. **Counter.**
  click on **Watch Current Symbol**.



15.6 Sample07: Receiving ADS Notifications

This article describes how to implement a TwinCAT 3 C++ module which receives ADS Notifications about data changes on other modules.
Since all other ADS communication has to be implemented in a similar way, this sample is the general entry point to initialize ads communication from TwinCAT 3 C++ modules.

Download

Here you can access the source code for this sample:

https://github.com/Beckhoff/TC1300_Samples/tree/main/S07-AdsNotifications

1. Using a Visual Studio with TwinCAT installed, open the project via **Open Project ....**
2. If you want to run the project on Windows or TwinCAT/BSD® (not required for Beckhoff RT Linux®): Configure signing for this project by right-clicking on the project on the page, selecting **Project -> Properties -> TC Sign**, enabling TwinCAT signing, and configuring your certificate and password if necessary.
   Further information on signing C++ projects can be found in the chapter TwinCAT [▶ 21].
3. Select your target system.
4. Build the sample (e.g. **Build->Build Solution**).
5. Activate the configuration by clicking  .

⇨ The sample is ready for operation.

Description

This sample describes the reception of ADS notifications in a TwinCAT 3 C++ module.

The solution contains 2 modules for this purpose.

- A C++ module, which registers for querying ADS notifications of a variable.
- For a simple understanding: A PLC program that provides a variable MAIN.PlcVar.
  If its value changes, an ADS notification is sent to the C++ module.
- The C++ module utilizes the message recording options. For a better understanding of the code, simply start the sample and note the output / error log when you change the value Main.PlcVar of the PLC module.

The address is prepared during the module transition PREOP->SAFEOP (SetObjStatePS). The CycleUpdate method contains a simple state machine, which sends the required ADS command. Corresponding methods show the receipts.

The inherited and overloaded method AdsDeviceNotificationInd is called when a notification is received.

During shutdown, ADS messages are sent during transition for the purpose of logoff (SetObjStateOS), and the module waits for receipts of confirmation until a timeout occurs.

Start of the module development

Creating a TwinCAT 3 C++ module with the aid of the ADS port wizard. This sets up everything you need for establishing an ADS communication. Simply use and overwrite the required ADS methods of ADS.h, as shown in the sample.

See also

ADS Communication [▶ 189]

15.7 Sample08: provision of ADS-RPC

This article describes the implementation of methods that can be called by ADS via the task.

Download

Here you can access the source code for this sample:

https://github.com/Beckhoff/TC1300_Samples/tree/main/S08-ADSRPC

1. Using a Visual Studio with TwinCAT installed, open the project via **Open Project ....**
2. If you want to run the project on Windows or TwinCAT/BSD® (not required for Beckhoff RT Linux®): Configure signing for this project by right-clicking on the project on the page, selecting **Project -> Properties -> TC Sign**, enabling TwinCAT signing, and configuring your certificate and password if necessary.
   Further information on signing C++ projects can be found in the chapter TwinCAT [▶ 21].
3. Select your target system.
4. Build the sample (e.g. **Build->Build Solution**).
5. Activate the configuration by clicking  .

⇨ The sample is ready for operation.

Description

The download contains 2 projects:

- The TwinCAT project, which contains a C++ module. This offers some methods that can be called by ADS.
- Also included is a Visual C++ project that calls the methods from the User mode as a client.

Four methods with different signatures are provided and called. These are organized in two interfaces, so that the composition of the ADS symbol names of the methods becomes clear.

Understanding the sample

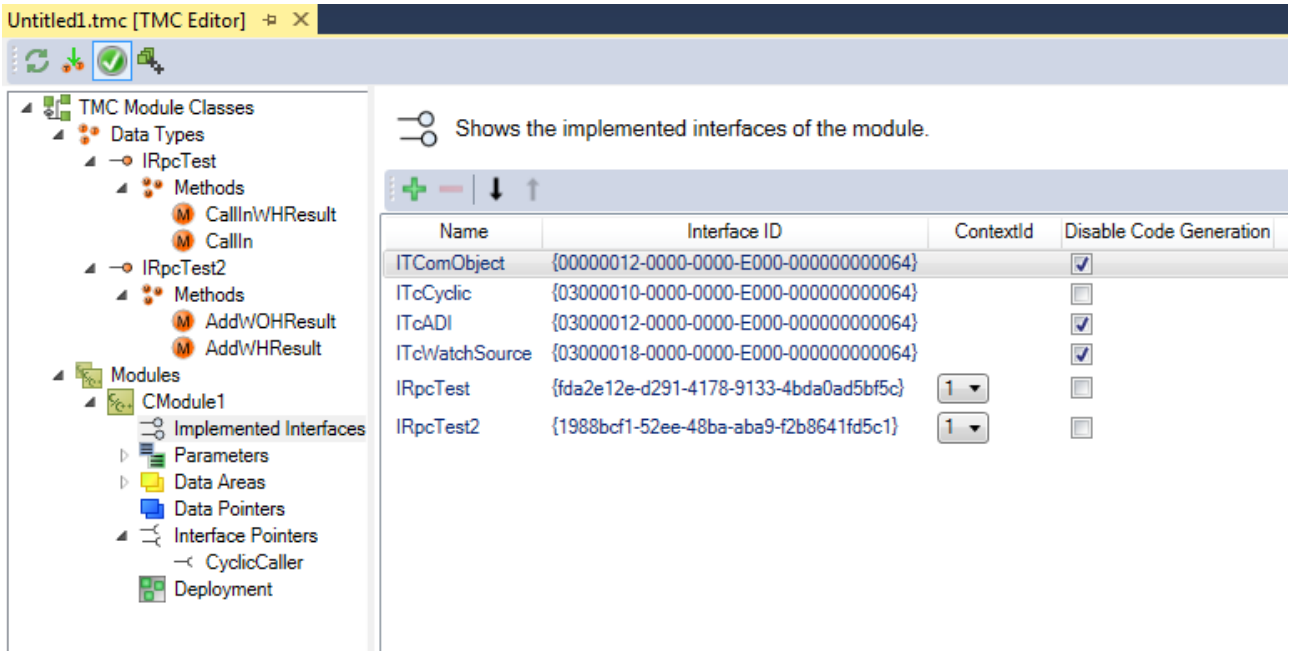
The sample consists of the TwinCAT C++ module, which offers the RPC methods and a C++ sample program that calls them.

TwinCAT C++ module

The TwinCAT C++ project contains a module and an instance of the module with the name "foobar".

RPC methods are normal methods that are described by interfaces in the TMC editor and are additionally enabled by an **RPC enable** checkbox. The options are described in more detail in the Description of the TMC Editor [▶ 88].

In this module two interfaces are described and implemented, as can be seen in the TMC Editor:



The methods, four in all, have different signatures of call and return values.

The ADS symbol name is formed according to the pattern: `ModulInstance.Interface#Methodenname`
The ContextId, which defines the context for execution, is particularly important in the implementing module.

As can be seen in the C++ code itself, the methods are generated by the code generator and implemented like normal methods of a TcCOM module.

Module1.cpp | Untitled1.tmc [TMC Editor]

Untitled1 | CModule1 | AddModuleToCaller()

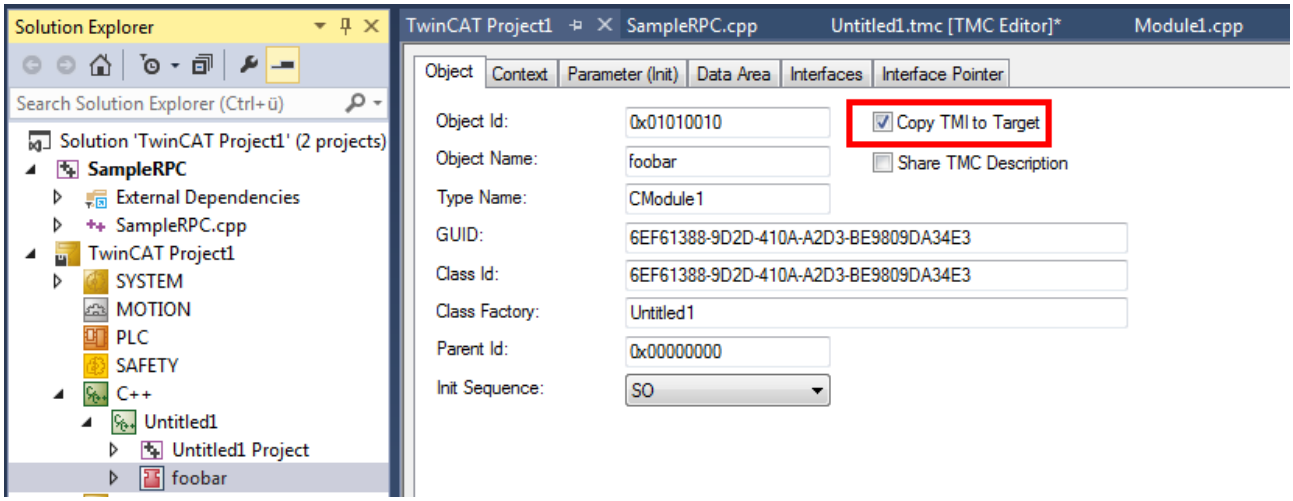
```
///<AutoGeneratedContent id="ImplementationOf_IRpcTest">
HRESULT CModule1::CallInWHResult(LONG in)
{
    HRESULT hr = S_OK;
    return hr;
}

HRESULT CModule1::CallIn(LONG in)
{
    HRESULT hr = S_OK;
    return hr;
}
///</AutoGeneratedContent>

///<AutoGeneratedContent id="ImplementationOf_IRpcTest2">
HRESULT CModule1::AddWOHResult(LONG a, LONG b, LONG& sum)
{
        HRESULT hr = S_OK;
        sum = a+b;
        m_Trace.Log(tlAlways, FNAMEA "got called with %d %d -> %d", a, b, sum);
        return hr;
}

HRESULT CModule1::AddWHResult(LONG a, LONG b, LONG& sum)
{
    HRESULT hr = S_OK;
    sum = a + b;
    m_Trace.Log(tlAlways, FNAMEA "got called with %d %d -> HRESULT %d ", a, b, sum);
    return hr;
}
///</AutoGeneratedContent>
```

If the type information of the methods is to be available on the target system, the TMI file of the module can be transferred to the target system.



The TwinCAT OPC UA Server offers the option to call these methods also by OPC UA – the TMI files are required on the target system for this.

C++ sample client

Directly after starting, the C++ client will fetch the handles and then call the methods any number of times; however a [RETURN] is expected between the procedures. Every other key leads to the enabling of the handle and the termination of the program.
The outputs illustrate the calls:

```
OK: AdsSyncReadWriteReq (getHdl foobar.IRpcTest#CallIn)
OK: AdsSyncReadWriteReq (getHdl foobar.IRpcTest#CallInWHResult)
OK: AdsSyncReadWriteReq (getHdl foobar.IRpcTest2#AddWOHResult)
OK: AdsSyncReadWriteReq (getHdl foobar.IRpcTest2#AddWHResult)

Press key to call all methods

Calling foobar.IRpcTest#CallIn
Send: 0

Calling foobar.IRpcTest#CallInWHResult
Value given: 1
ReturnCode: 0

Calling foobar.IRpcTest2#AddWOHResult
Value given A: 1
Value given B: 2
Value got (A+B): 3

Calling foobar.IRpcTest2#AddWHResult
Value given A: 1
Value given B: 2
ReturnCode: 0
Value got (A+B): 3
```

15.8 Sample10: module communication: Using data pointer

This article describes the implementation of two TwinCAT 3 C++ modules, which communicate via a data pointer.

Download

Here you can access the source code for this sample:

https://github.com/Beckhoff/TC1300_Samples/tree/main/S10-Mod2ModDataPointer

1. Using a Visual Studio with TwinCAT installed, open the project via **Open Project ....**
2. If you want to run the project on Windows or TwinCAT/BSD® (not required for Beckhoff RT Linux®): Configure signing for this project by right-clicking on the project on the page, selecting **Project -> Properties -> TC Sign**, enabling TwinCAT signing, and configuring your certificate and password if necessary.
   Further information on signing C++ projects can be found in the chapter TwinCAT [▶ 21].
3. Select your target system.
4. Build the sample (e.g. **Build->Build Solution**).
5. Activate the configuration by clicking .

⇨ The sample is ready for operation.

Description

This communication is based on a "split" data area: Provided by a module and accessible from another module via pointers.

It is not possible that two different data pointers are linked with the same entry in an output or input data area; without this limitation there could be synchronization problems. For this reason a ModuleDataProvider module consolidates input and output in a standard data area, which is not subject to this restriction.

All in all, this sample includes the following modules:

- ModuleDataProvider provides a data area, which can be accessed by the other modules.
  The data area contains 4 bits (2 for input, 2 for output) and 2 integers (1 for input, 1 for output).
- ModuleDataInOut provides "normal" input variables, which are written to the data area of the ModuleDataProvider, and output variables, which are read from
  the data area.
  This instance of the CModuleDataInOut class serve as a simulation for real IO.

- ModuleDataAccessA accesses the data area of ModuleDataProvider and cyclically processes Bit1 / BitOut1 and the integer.
- ModuleDataAccessB accesses the data area of ModuleDataProvider and cyclically processes Bit2 / BitOut2 and the integer.

The user of the sample triggers ModuleDataInOut by setting the variables ValueIn / Bit1 / Bit2:

- When setting the input Bit1, the output Switch1 will be set accordingly.
- When setting the input Bit2, the output Switch2 will be set accordingly.
- When the input ValueIn is set, the output ValueOut is incremented twice in each cycle.

All modules are configured such that they have the same task context, which is necessary since access via pointers offers no synchronization mechanism. The order of execution corresponds to the order specified on the context configuration tab. This value is passed on as parameter SortOrder and stored in the smart pointer of the cyclic caller (m_spCyclicCaller), which also contains the object ID of the cyclic caller.

Understanding the sample

The module ModuleDataInOut has input and output variables. They are linked with the corresponding variables of the data provider.

The module ModuleDataProvider provides an input and output data array and implements the ITcIoCylic interface. The method InputUpdate copies data from the input variables to the DataIn symbol of the standard data area Data, and the method OutputUpdate copies data from the DataOut symbol to the output variables.

The modules ModuleDataAccessA and ModuleDataAccessB contain pointers to data areas of the data provider via links. These pointers are initialized during the transition from SAFEOP to OP. ModuleDataAccessA cyclically sets BitOut1 according to Bit1. ModuleDataAccessB accordingly, with BitOut2 / Bit2. Both increment ValueOut through multiplication of the internal counter with the value ValueIn.

It is important that all modules are executed in the same context, since there is no synchronization mechanism via data pointers. The order of execution is defined by the **Sort Order** on the **Context** tab of the respective module. This is provided as parameter **SortOrder** in the SmartPointer (m_SpCyclicCaller), which also includes the ObjectID.

15.9 Sample11: Module communication: method call PLC module to C++ module

This article describes the implementation:

- A C++ module [▶ 258] that provides methods for controlling a state machine.
  Follow this step-by-step introduction to the implementation of a C++ module that provides an interface to the state machine.

- Of a PLC module [▶ 272] for calling the function of the C++ module.
  The fact that no hard-coded link exists between the PLC and the C++ module is a great. Instead, the called C++ instance can be configured in the System Manager.
  Follow this step-by-step introduction with regard to the implementation of a PLC project that calls methods from a C++ module.

Download

Here you can access the source code for this sample:

https://github.com/Beckhoff/TC1300_Samples/tree/main/S11-Mod2ModMethod

1. Using a Visual Studio with TwinCAT installed, open the project via **Open Project ....**
2. If you want to run the project on Windows or TwinCAT/BSD® (not required for Beckhoff RT Linux®): Configure signing for this project by right-clicking on the project on the page, selecting **Project -> Properties -> TC Sign**, enabling TwinCAT signing, and configuring your certificate and password if necessary.
   Further information on signing C++ projects can be found in the chapter TwinCAT [▶ 21].

3. Select your target system.
4. Build the sample (e.g. **Build->Build Solution**).

5. Activate the configuration by clicking .

⇨ The sample is ready for operation.

15.9.1 TwinCAT 3 C++ module providing methods

This article describes the creation of a TwinCAT 3 C++ module that provides an interface with several methods that can be called by a PLC and also by other C++ modules.

The idea is to create a simple state machine in C++ that can be started and stopped from the outside by other modules, but which also allows the setting or reading of the particular state of the C++ state machine.

Two further articles use the result from this C++ state machine.

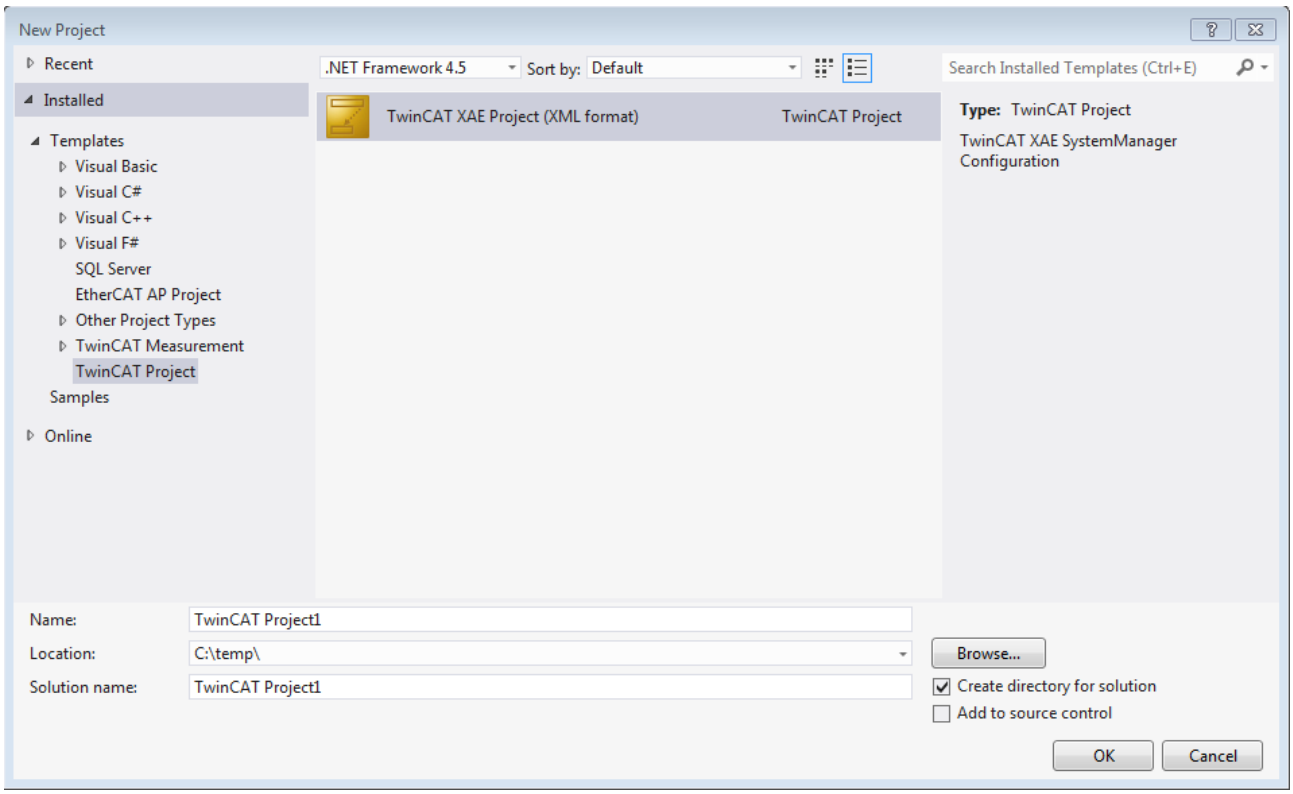
- Calling the function from the PLC logic [▶ 257] - i.e. affecting the C++ code from the PLC.
- Calling the function from the C++ logic [▶ 284] - i.e. interaction between two C++ modules.

This article describes:

- Step 1: create a new TwinCAT 3 project [▶ 258].
- Step 2: create a new TwinCAT 3 C++ driver [▶ 259].
- Step 3: create a new TwinCAT 3 interface.
- Step 4: add methods to the interface [▶ 262].
- Step 5: add a new interface to the module.
- Step 6: start the TwinCAT TMC Code Generator to generate code for the module class description [▶ 267].
- Step 7: implement the member variables and the constructor.
- Step 8: implement methods.
- Step 9: implement cyclic update.
- Step 10: compile code [▶ 270]
- Step 11: create an instance of the C++ module.
- Step 12: done. Check the results.

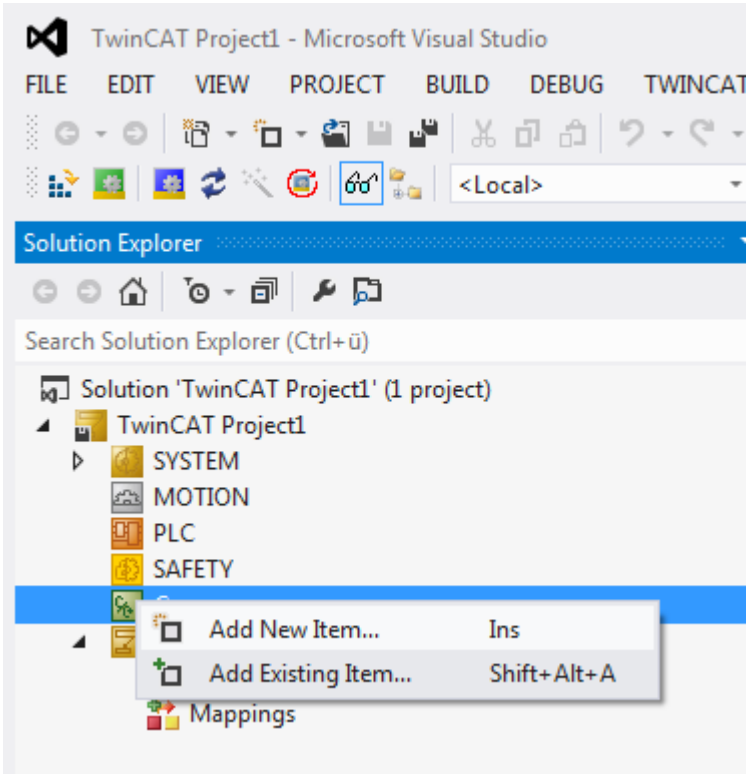
Step 1: Creating a TwinCAT 3 project

First of all, create a TwinCAT project as usual.

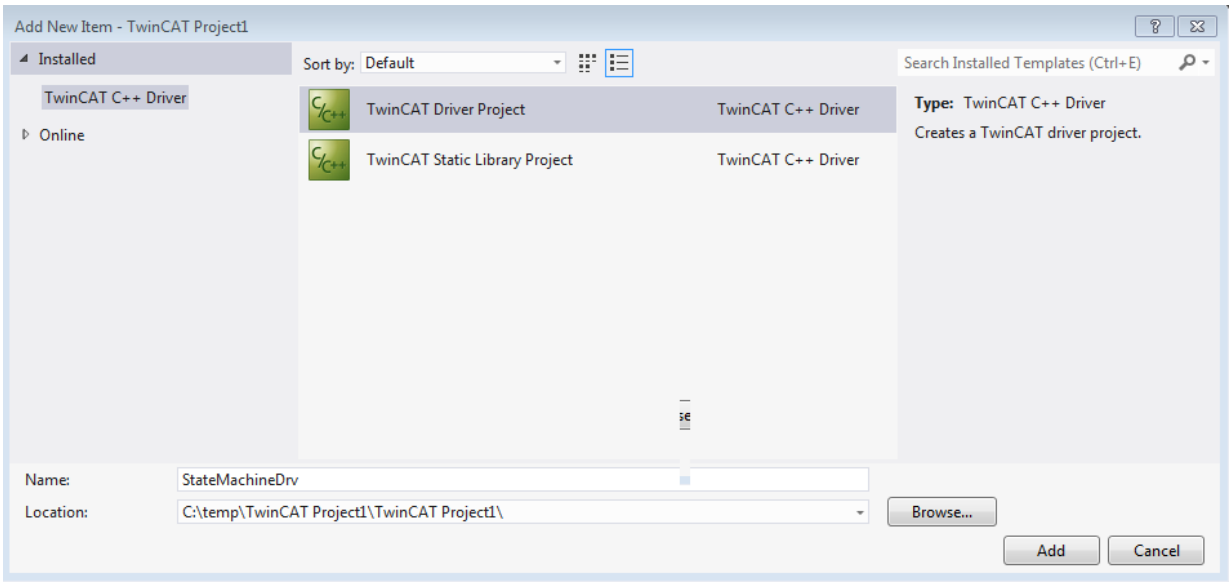


Step 2: create a new TwinCAT 3 C++ driver

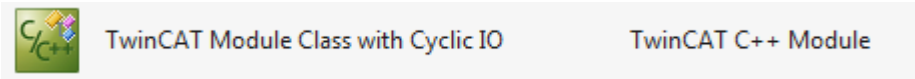
1. Right-click on **C++** and **Add New Item…**



2. Select the template TwinCAT Driver Project and enter a driver name, "StateMachineDrv" in this sample. Click on **Add** to continue.

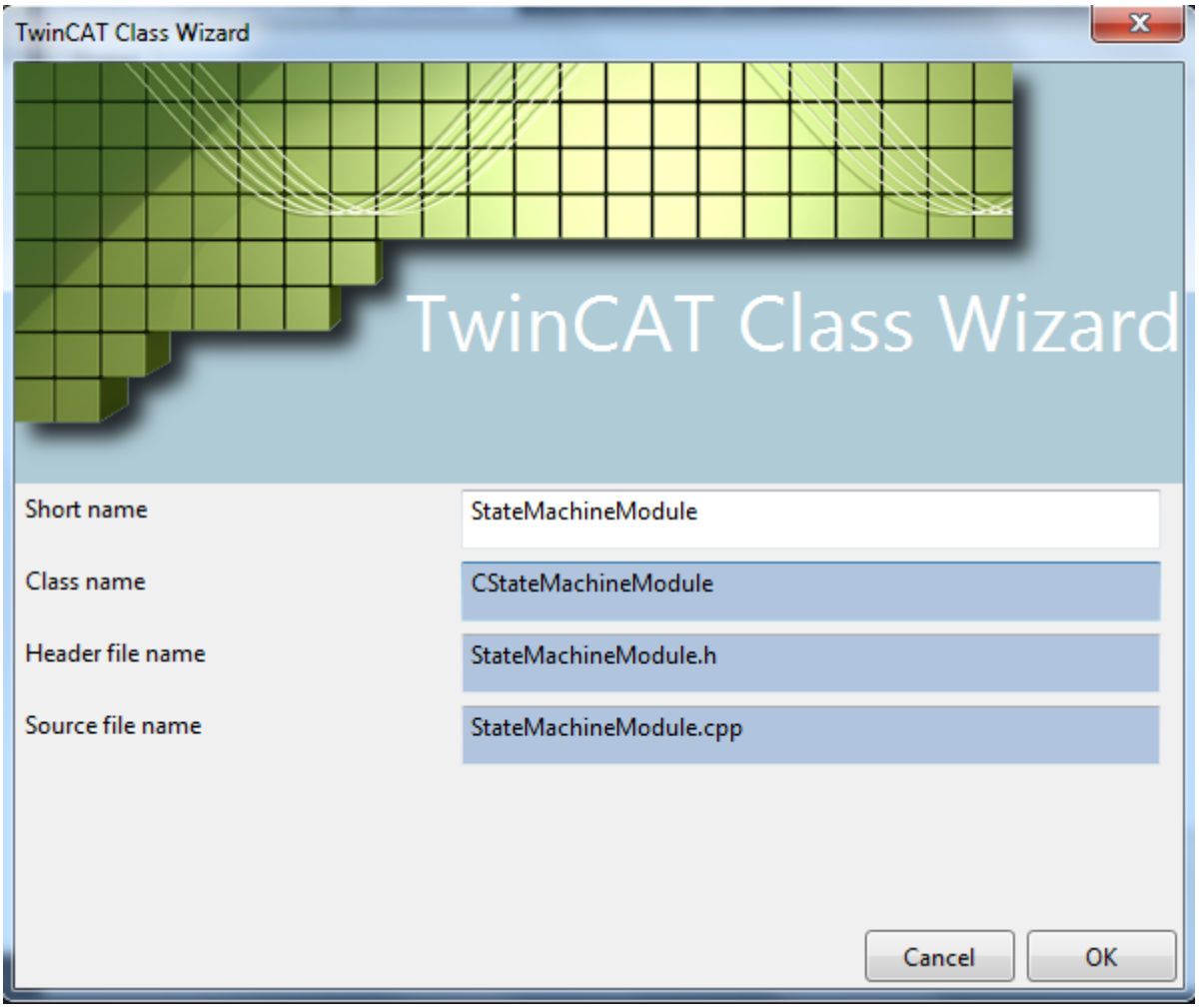


3. Select a template to be used for this new driver. In this sample "TwinCAT Module Class with Cyclic IO" is selected, since the internal counter of the state machine is available for assigning to the IO.
4. Click on **Add** to continue.



5. Specify a name for the new class in the C++ driver "StateMachineDrv".
The names of the module class and the header and source files are derived from the specified "Short Name".

6. Click on **OK** to continue.



⇨ The wizard then creates a C++ project, which can be compiled error-free.

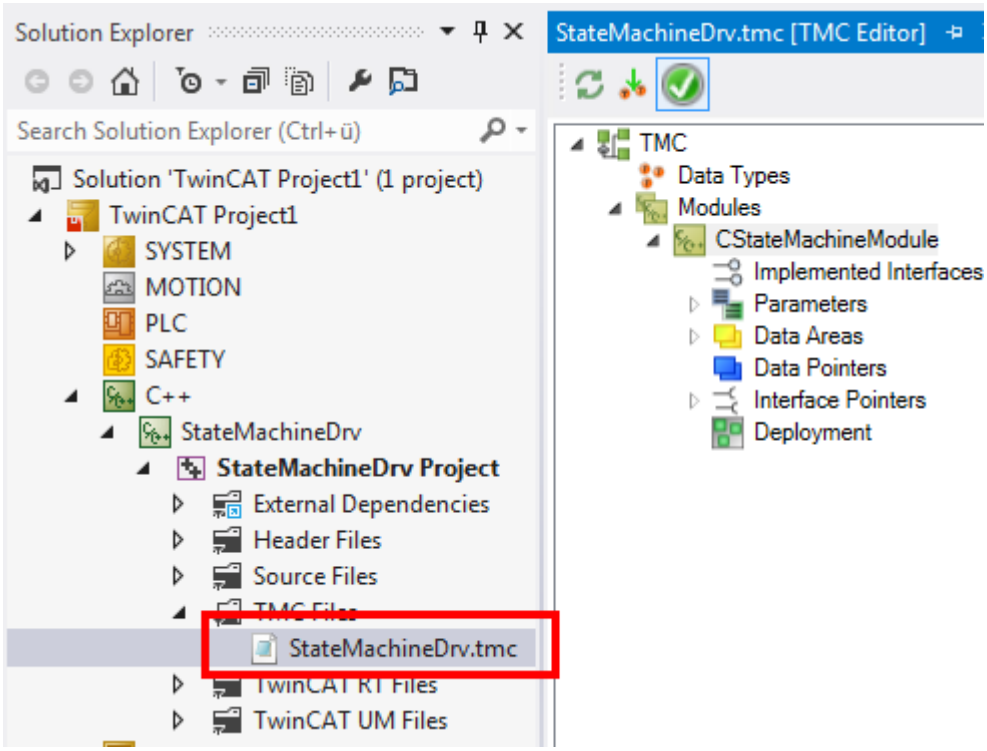
Step 3: create a new TwinCAT 3 interface

NOTICE

Name conflict

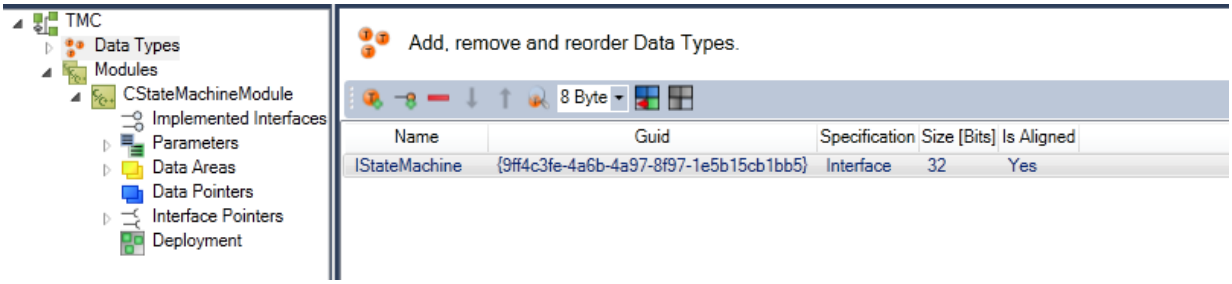
A name collision can occur if the driver is used in combination with a PLC module.
Do not use any of the keywords that are reserved for the PLC as names.

1. Start the TMC editor by double-clicking on **StateMachineDrv.tmc**.



2. Select **Data Types** within the TMC editor.

3. Add a new interface by clicking on **Add a new interface** .
   ⇨ A new entry **IInterface1** is then listed.
4. Open **IInterface1** by double-clicking in order to change the properties of the interface.
5. Enter a meaningful name - in this sample "IStateMachine".

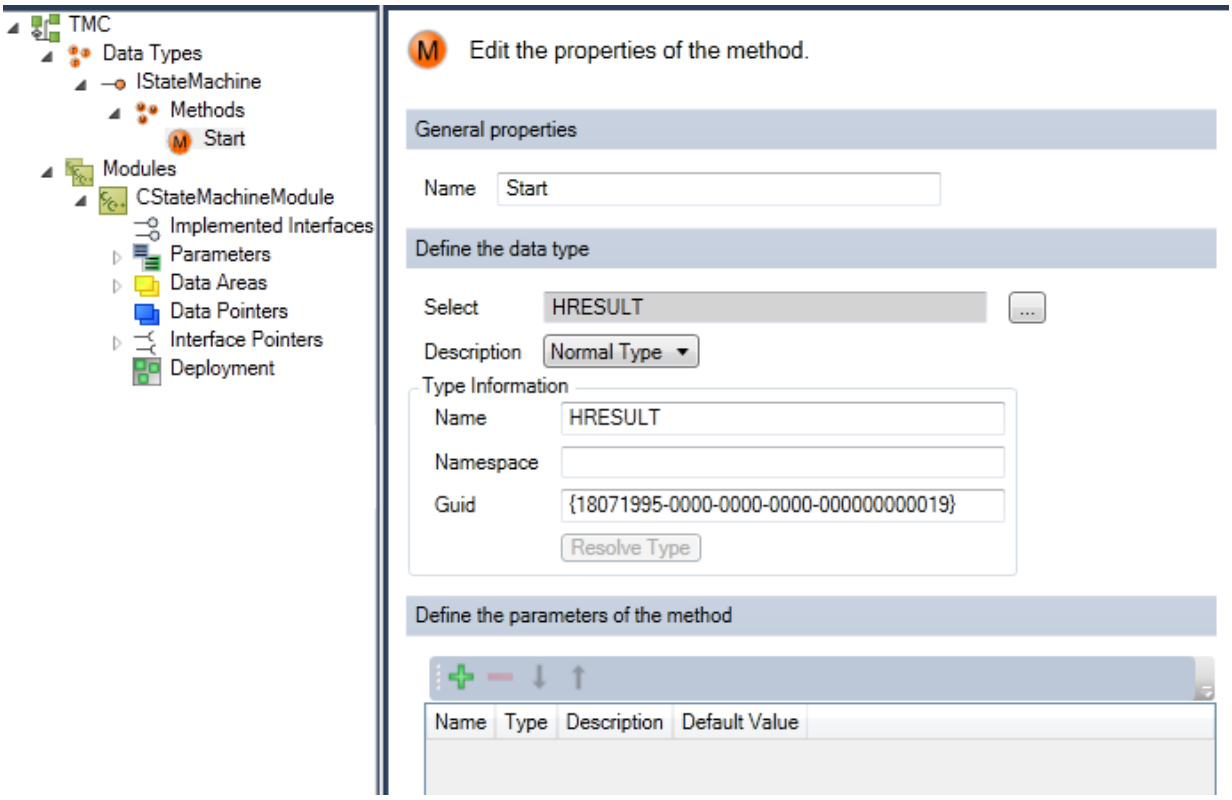


⇨ The interface has been created.

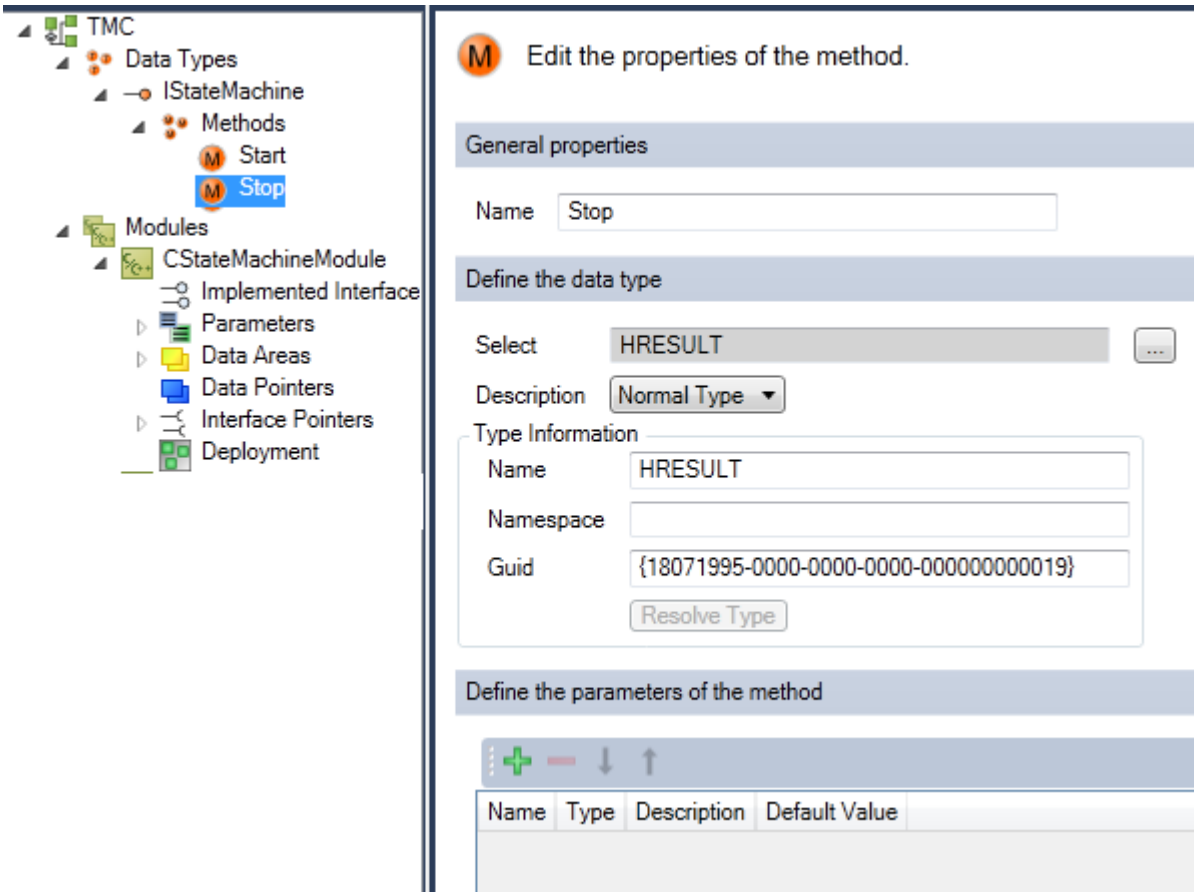
Step 4: add methods to the interface

1. Click on **Edit Methods...** to get a list of the methods of this interface:
   Click on the **+** button to create a new default method, Method1.

2. Replace the default name Method1 by a more meaningful name, in this sample "Start".

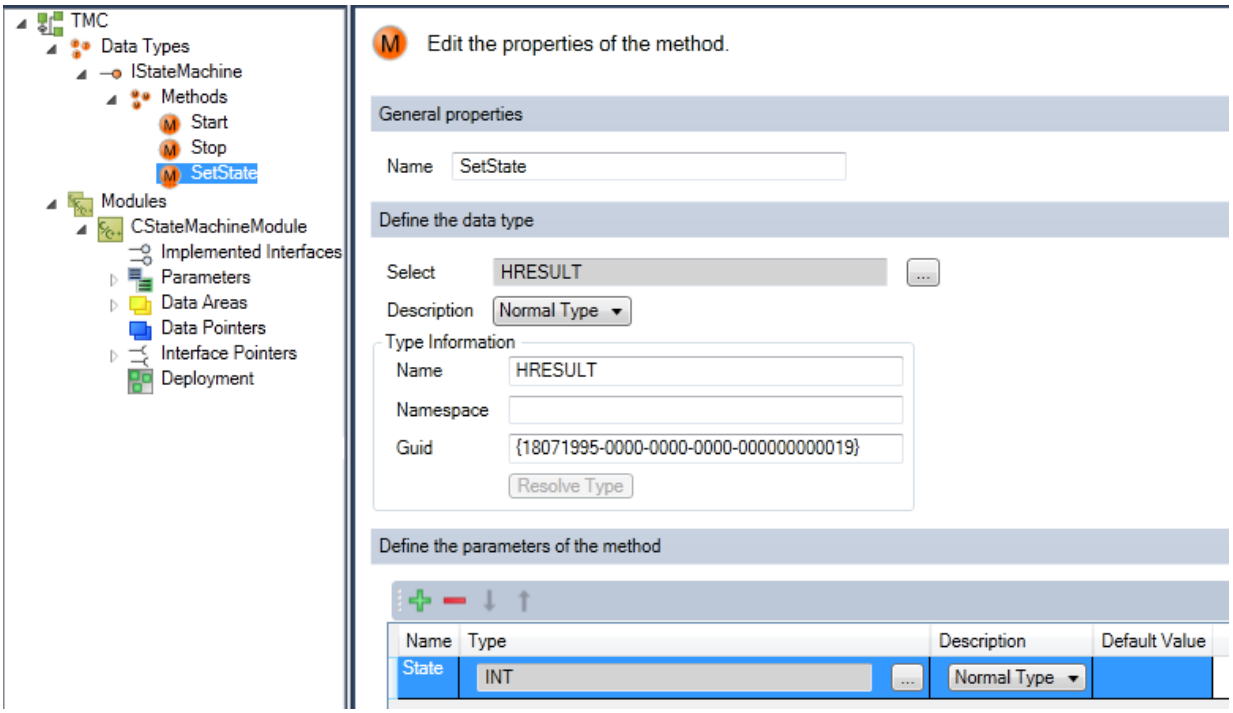


3. Add a second method and name it "Stop".



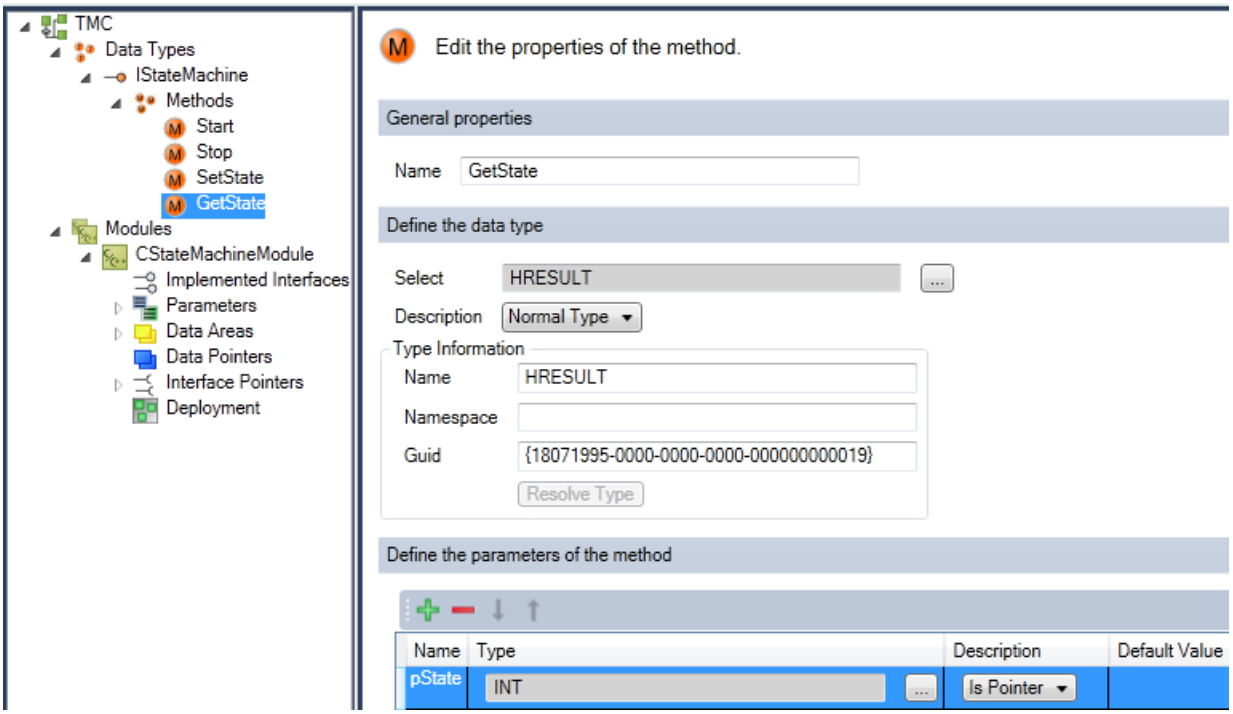
4. Add a third method and name it "SetState".

5. Subsequently, you can add parameters by clicking on **Add a new parameter** or edit parameters of the SetState method.

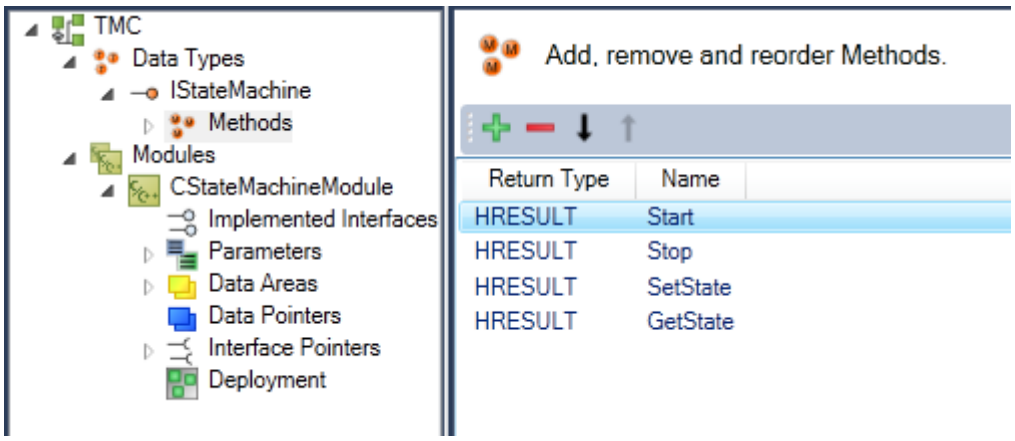


⇨ The new parameter, Parameter1, is generated by default as **Normal Type INT**.

6. Click on the name "Parameter1" and change the name in the edit box to "State".
7. After Start, Stop and SetState have been defined, define a further method.
8. Rename it "GetState".
9. Add a parameter and name it "pState" (which is conceived to become a pointer later on).
10. Change Normal Type to Is Pointer.

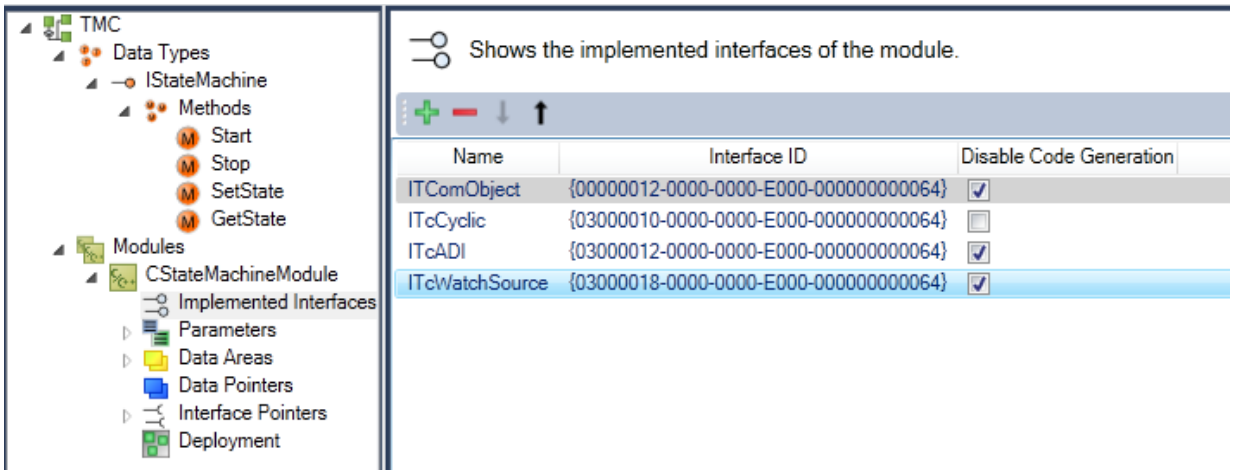


⇨ You then obtain a list of all methods. You can change the order of the methods with the  buttons.

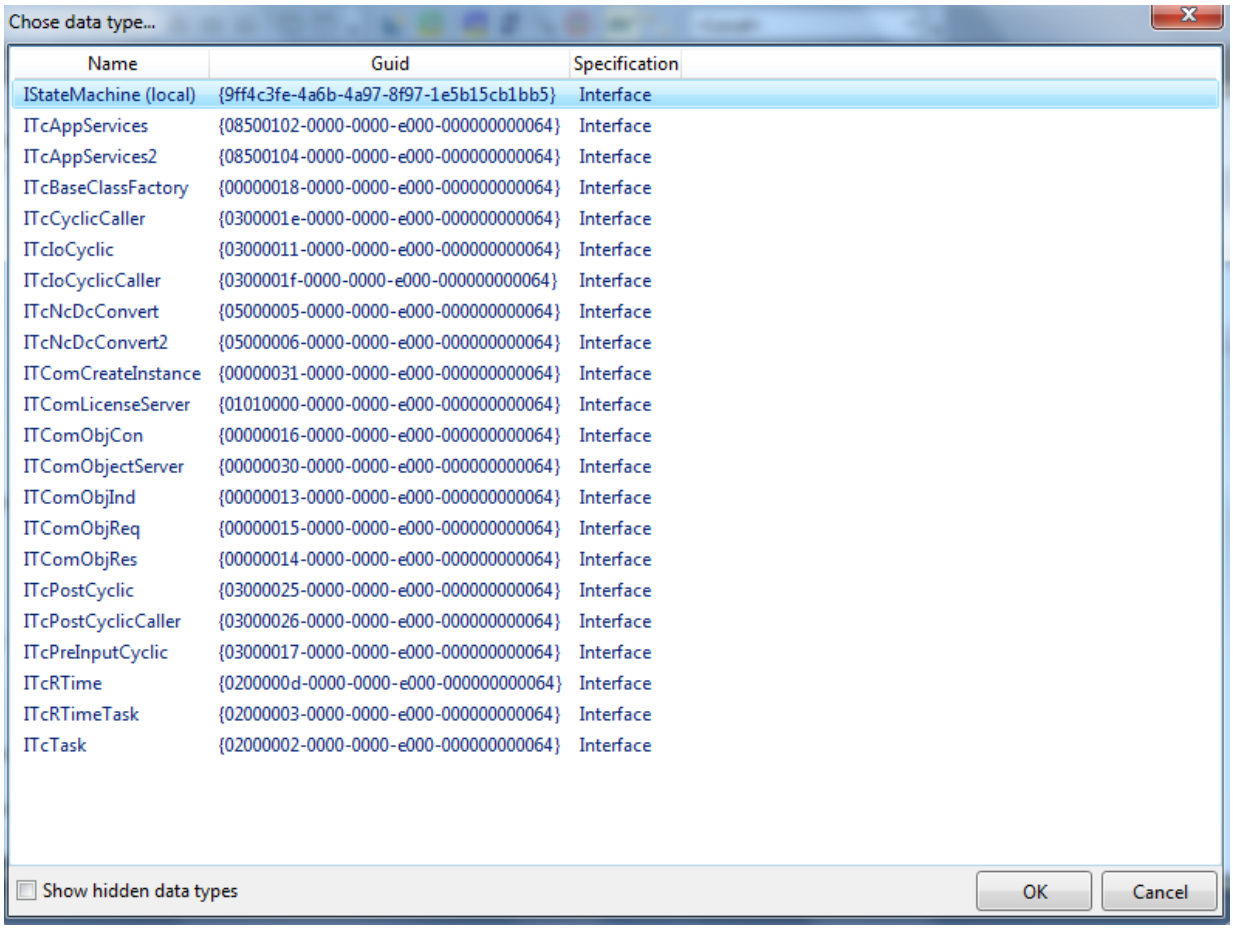


Step 5: add a new interface to the module

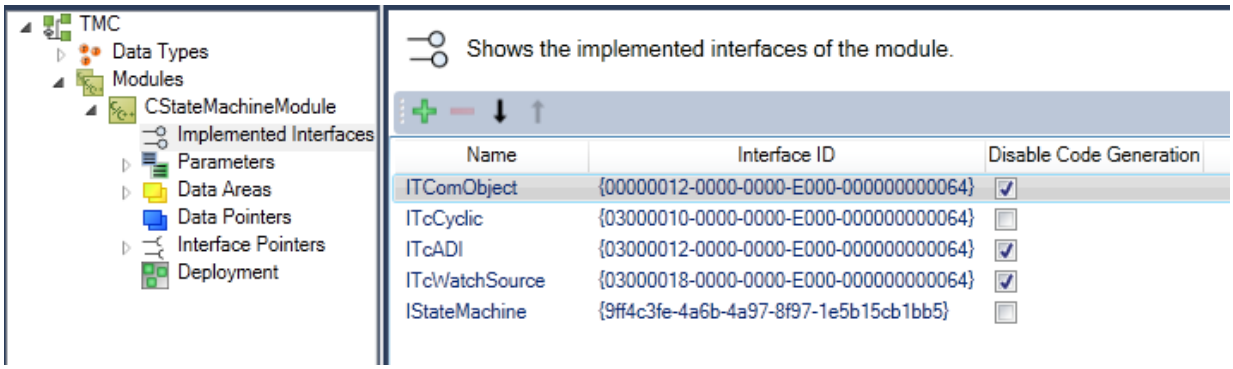
1. Select the module that is to be extended by the new interface - in this case select the destination **Modules->CStateMachineModule**.
2. Extend the list of implemented interfaces by a new interface with **Add a new interface to the module** by clicking on the **+** button.



3. All available interfaces are listed - select the new interface IStateMachine and end with **OK.**

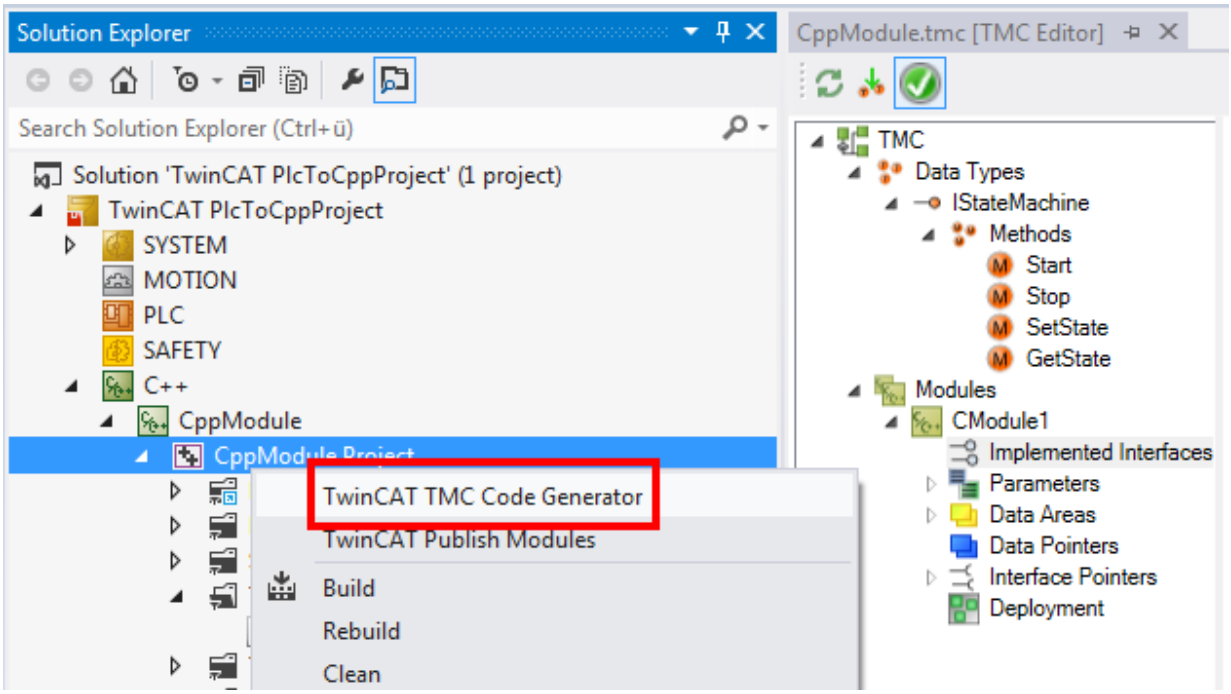


⇨ The new interface IStateMachine is part of the module description.



Step 6: Start the TwinCAT TMC Code Generator

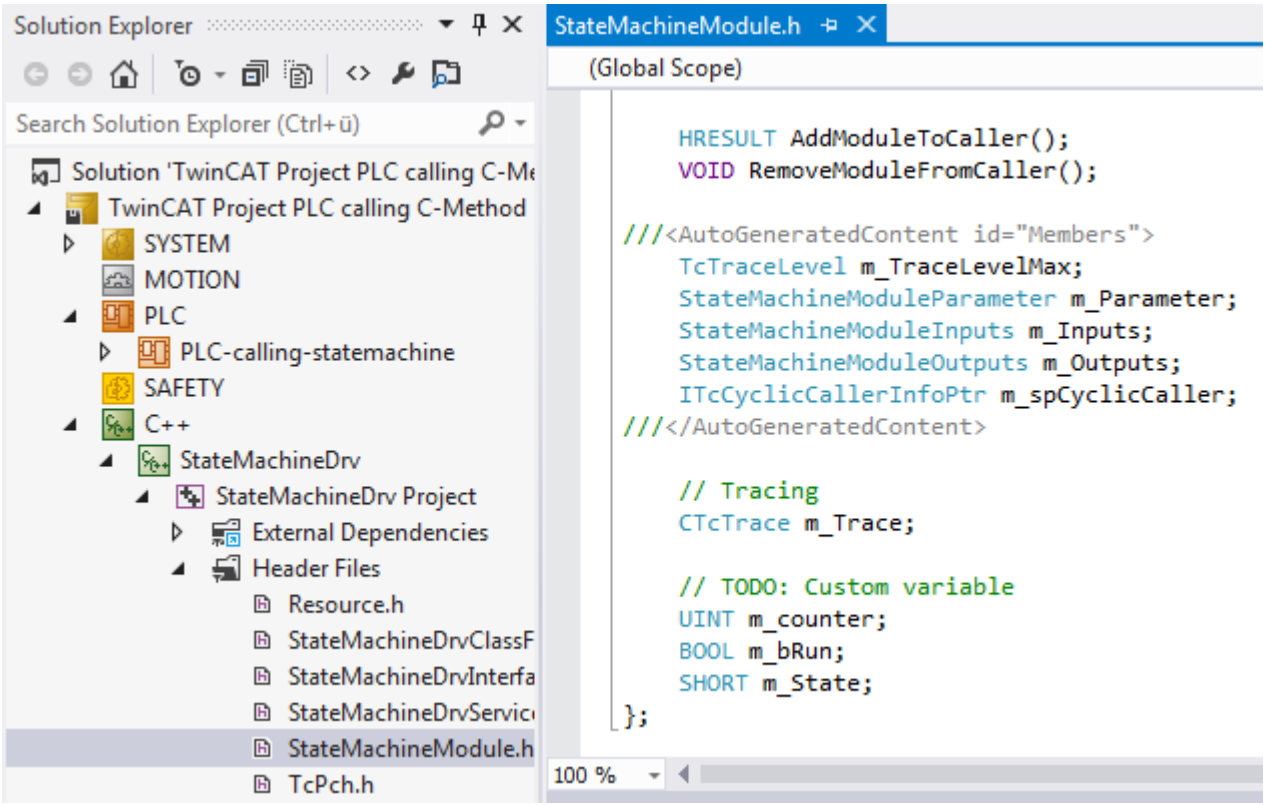
1. In order to generate the C/C++ code on the basis of this module, right-click in the C/C++ project and then select the **TwinCAT TMC Code Generator.**



⇨ The module StateMachineModule.cpp now contains the new interfaces
CModule1: Start()
CModule1: Stop()
CModule1: SetState(SHORT State)
CModule1: GetState(SHORT* pState).

Step 7: Implementing the member variables and the constructor

Add the member variables to the header file StateMachineModule.h.



Step 8: Implementing methods

Implement the code for the four methods in the StateMachineModule.cpp:

```
///<AutoGeneratedContent id="ImplementationOf_IStateMachine">
HRESULT CModule1::Start()
{
    HRESULT hr = S_OK;
    m_bRun = TRUE;
    return hr;
}

HRESULT CModule1::Stop()
{
    HRESULT hr = S_OK;
    m_bRun = FALSE;
    return hr;
}

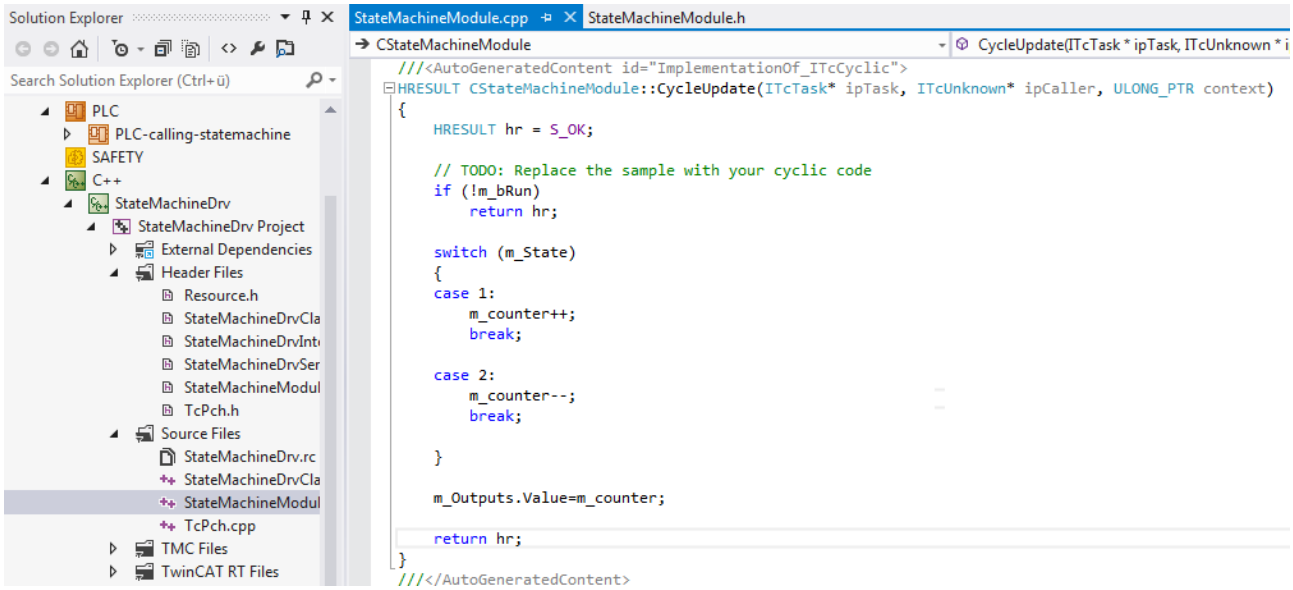
HRESULT CModule1::SetState(SHORT State)
{
    HRESULT hr = S_OK;
    m_State = State;
    return hr;
}

HRESULT CModule1::GetState(SHORT* pState)
{
    HRESULT hr = S_OK;
    *pState = m_State;
    return hr;
}
///</AutoGeneratedContent>
```

Step 9: Implementing cyclic updating

The C++ module instance is cyclically called, even if the internal state machine is in Stop mode.

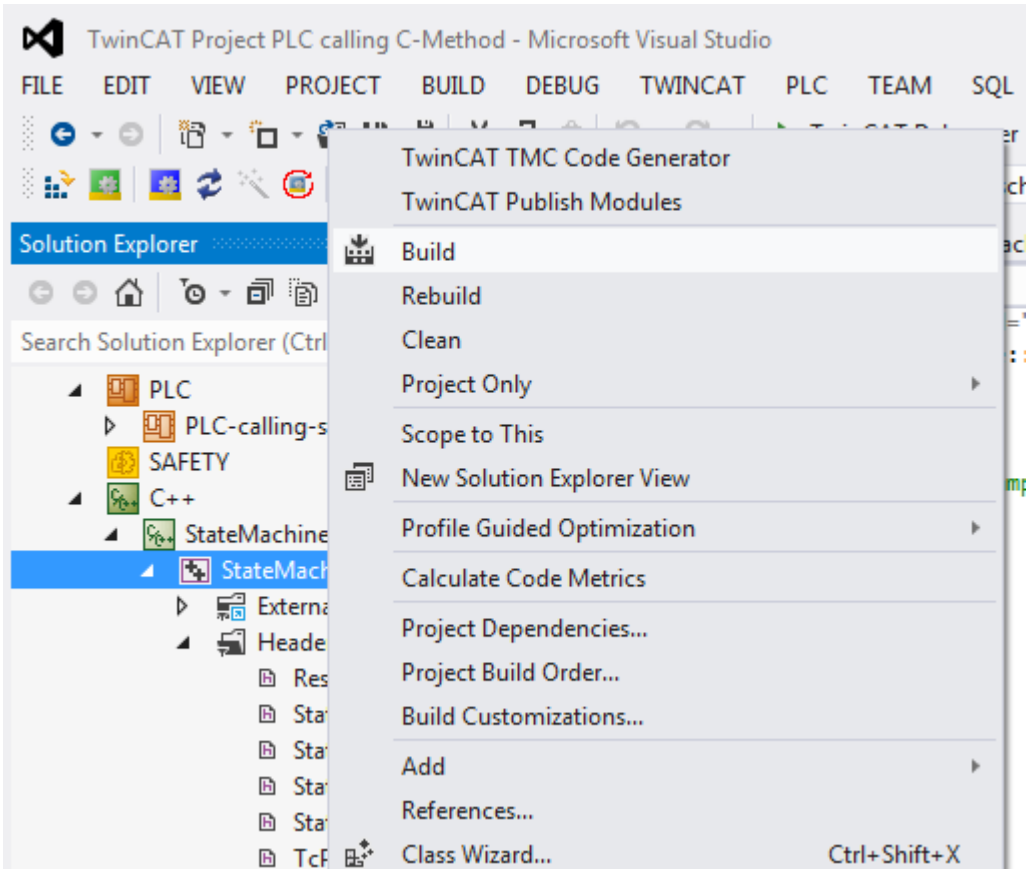
- If the state machine is not to be executed, the m_bRun Flag signals that the code execution of the internal state machine is to be exited.
- If the state is "1" the counter must be incremented.
- If the state is "2" the counter must be decremented.
- The resulting counter value is assigned to Value, which is a member variable of the logical output of the data area. This can be assigned to the physical IO level or to other data areas of other modules at a later time.



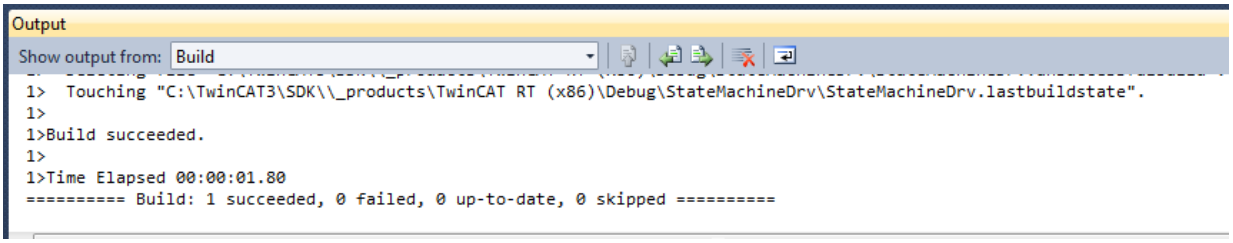
.

Step 10: compilation of code

1. Following the implementation of all interfaces, compile the code by right-clicking on the state machine and selecting **Build**.

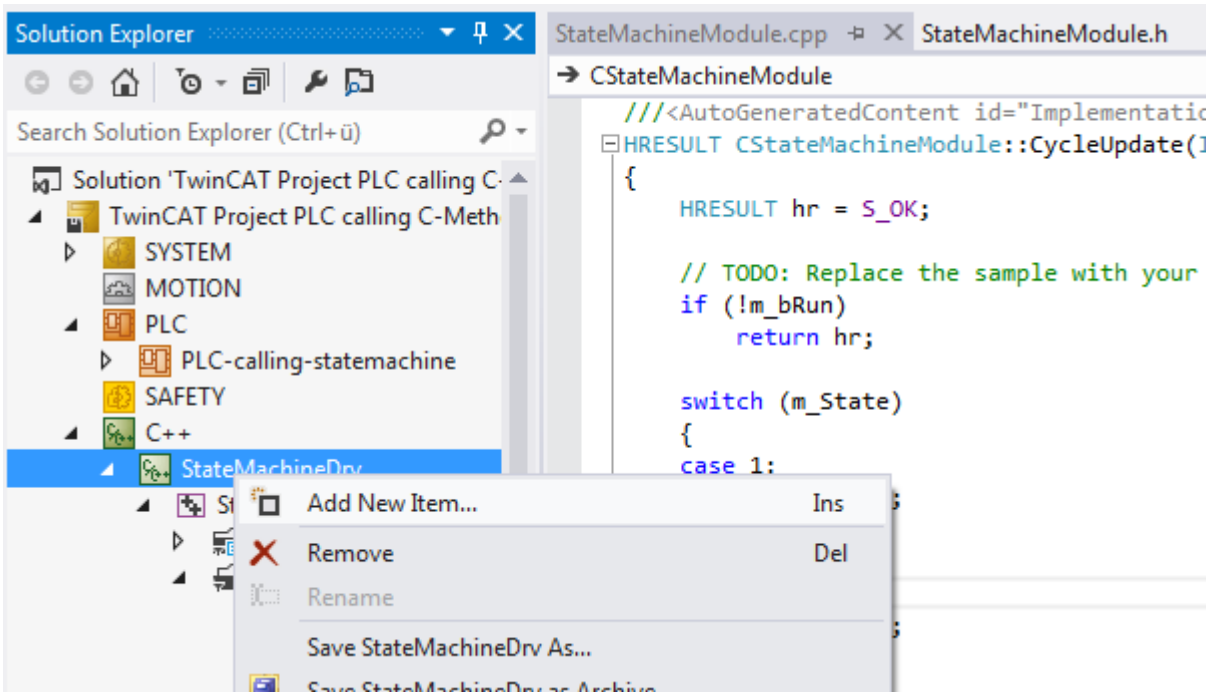


2. Repeat the compilation and optimize your code until the result looks like this:

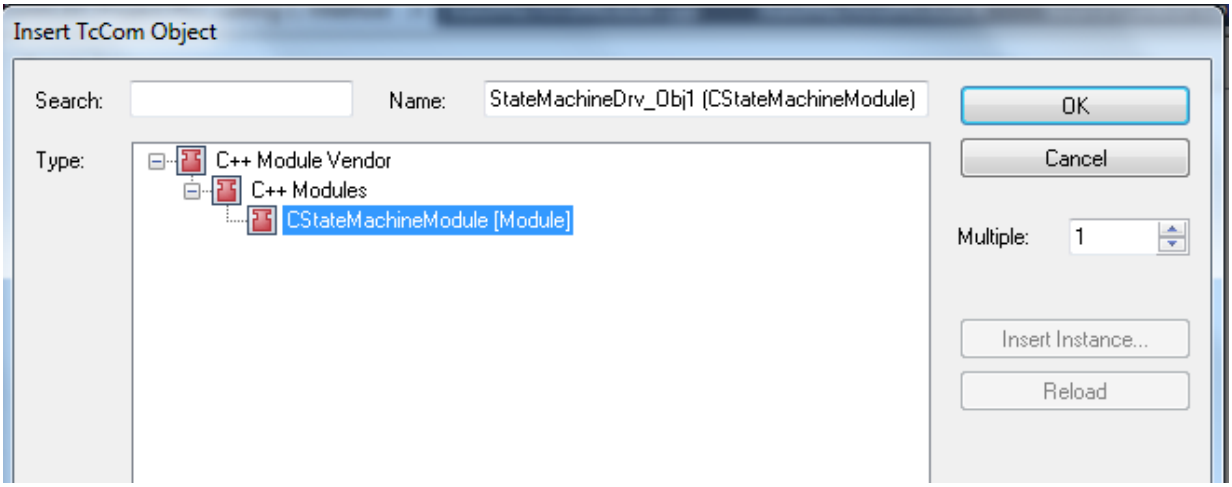


Step 11: creating an instance of the C++ module

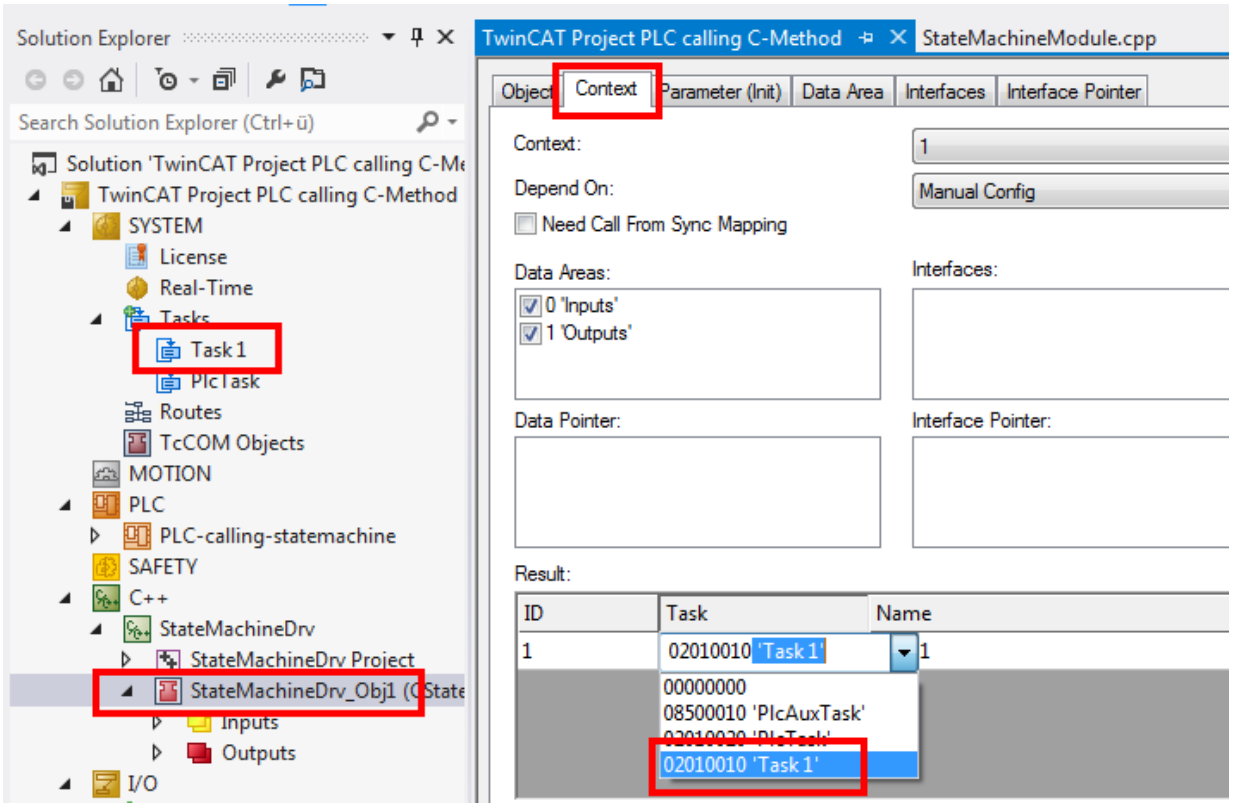
1. Right-click on the C++ project and select **Add New Item...** to create a new module instance.



2. Select the module that is to be added as a new instance – in this case CStateMachineModule.



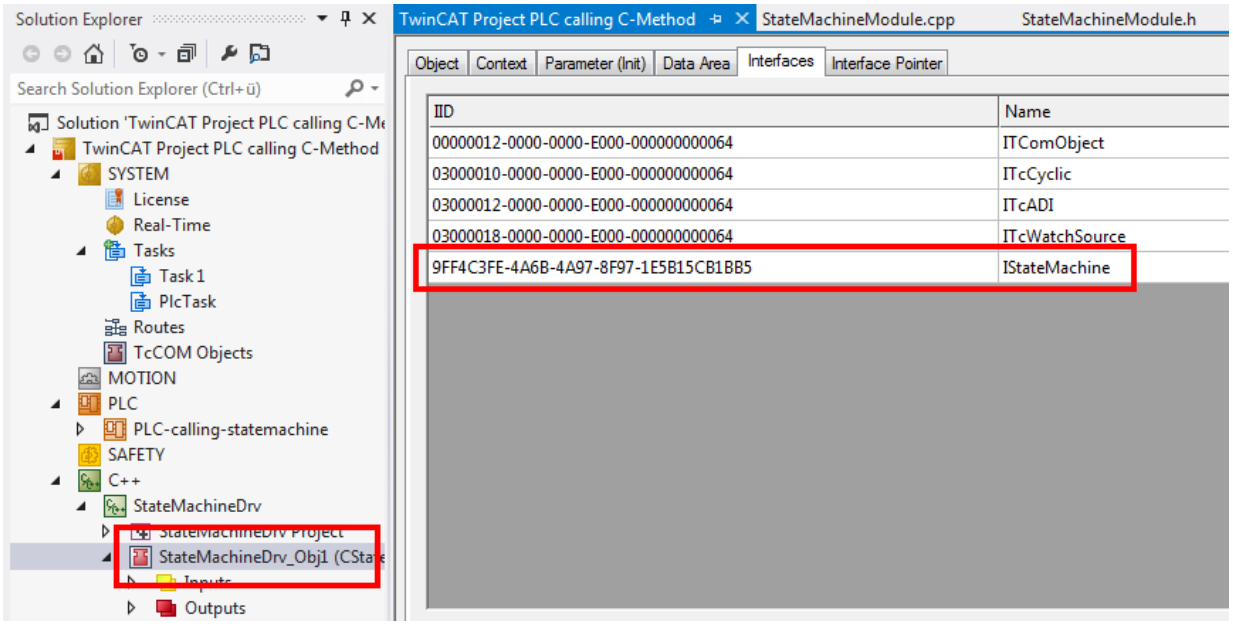
3. Assign the instance to a task:



Step 12: finished - check the result

1. Navigate to the module listed in the solution tree and select the **Interfaces** tab on the right-hand side.

⇨ The new interface IStateMachine is listed.



15.9.2 Calling methods offered by another module via PLC

This article describes how a PLC can call a method that is provided by another module; in this case: the previously defined C++ module.

- Step 1: check the available interfaces.
- Step 2: create a new PLC project.

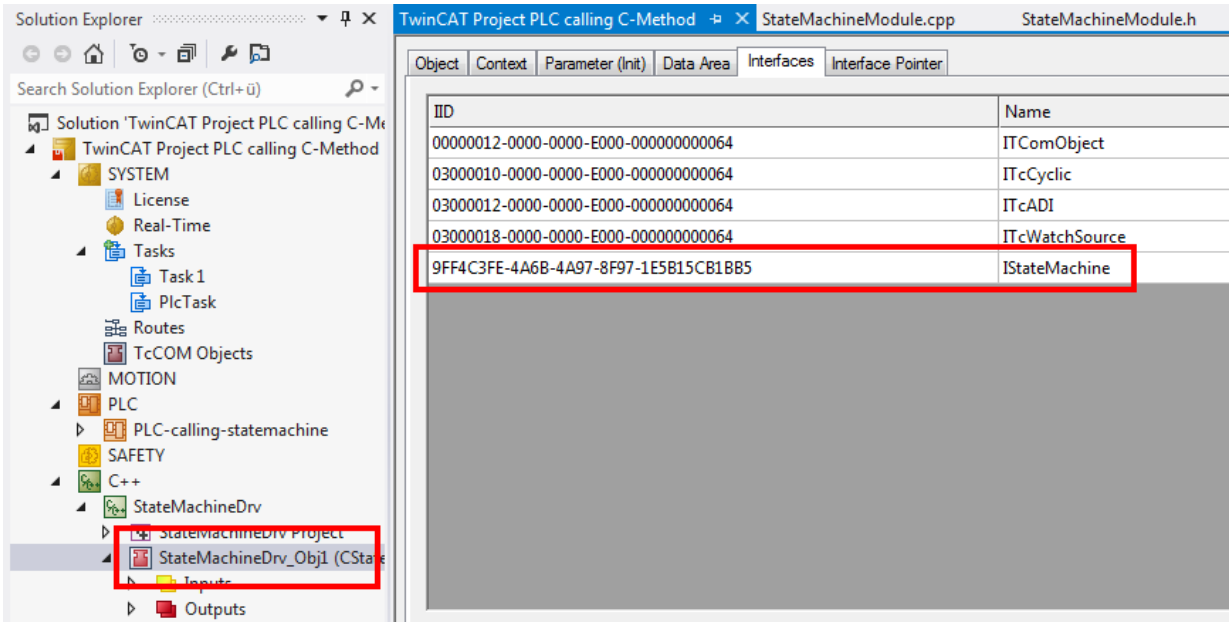
- Step 3: add a new FB state machine [▶ 274](which acts as the proxy that calls the C++ module methods).
- Step 4: clean up the function block interface.
- Step 5: add the FB methods "FB_init" and "exit". [▶ 278]
- Step 6: implement the FB methods.
- Step 7: call the FB state machine in the PLC. [▶ 282]
- Step 8: compile the PLC code.
- Step 9: link the PLC FB with the C++ instance.
- Step 10: observe the execution of both modules, PLC and C++.

Step 1: Check available interfaces

Option 1:

1. Navigate to the C++ module instance.
2. Select the **Interfaces** tab.

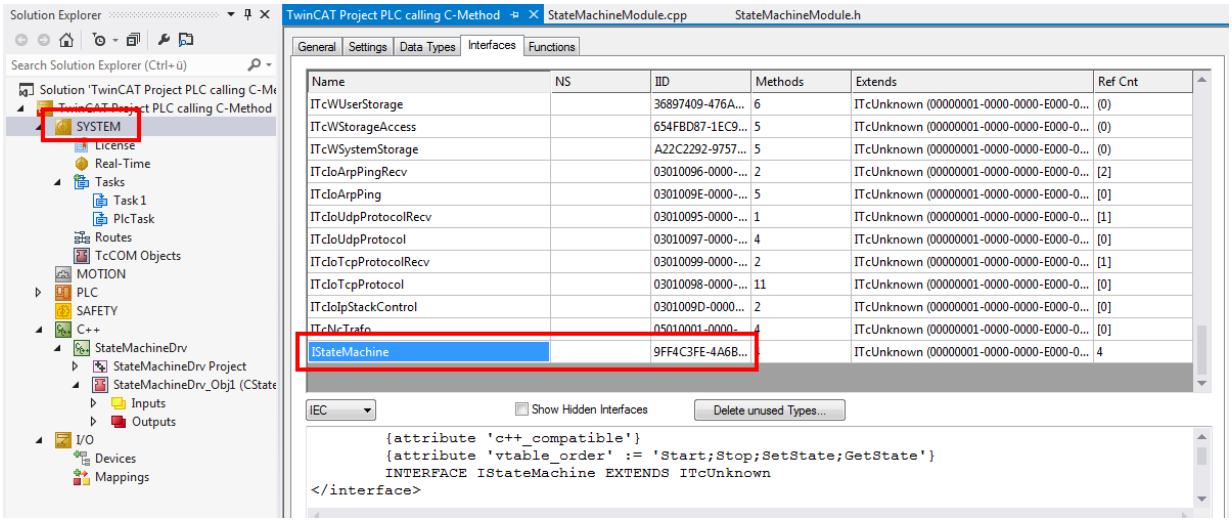
⇨ The **IStateMachine** interface is in the list with its specific IID (Interface ID).



Option 2:

1. Navigate to **System.**
2. Select the **Interfaces** tab.

⇨ The **IStateMachine** interface is in the list with its specific IID (Interface ID).

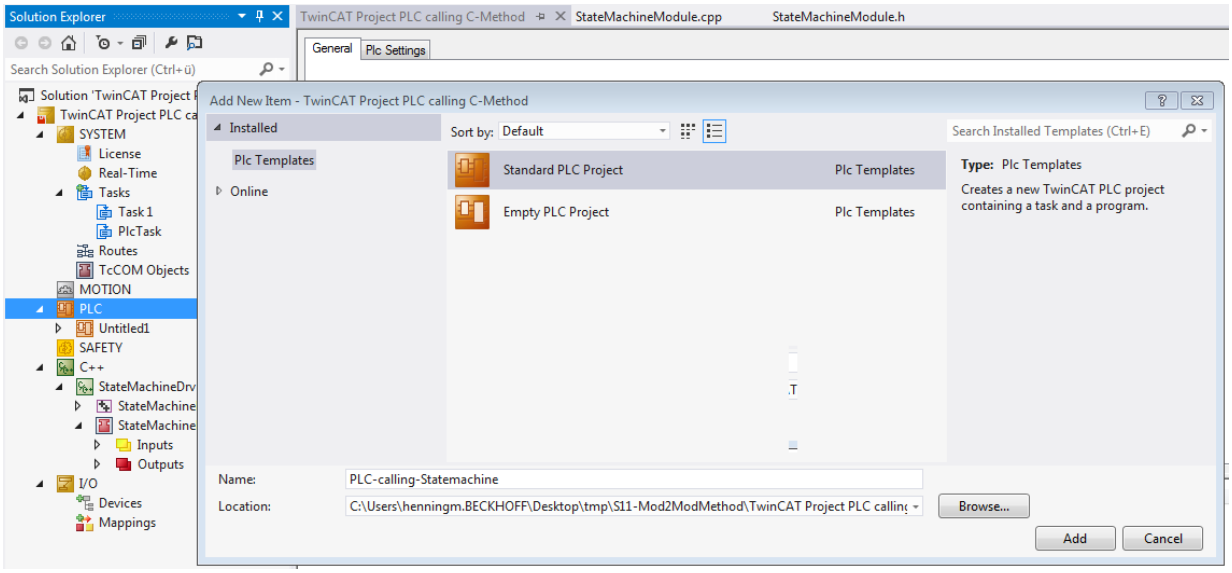


The lower section shows the stored code in different programming languages.

Step 2: Creating a new PLC project

A standard PLC project called "PLC-calling state machine" is created.

1. Right-click on the PLC node.
2. Select **Standard PLC Project**.
3. Adapt the name.

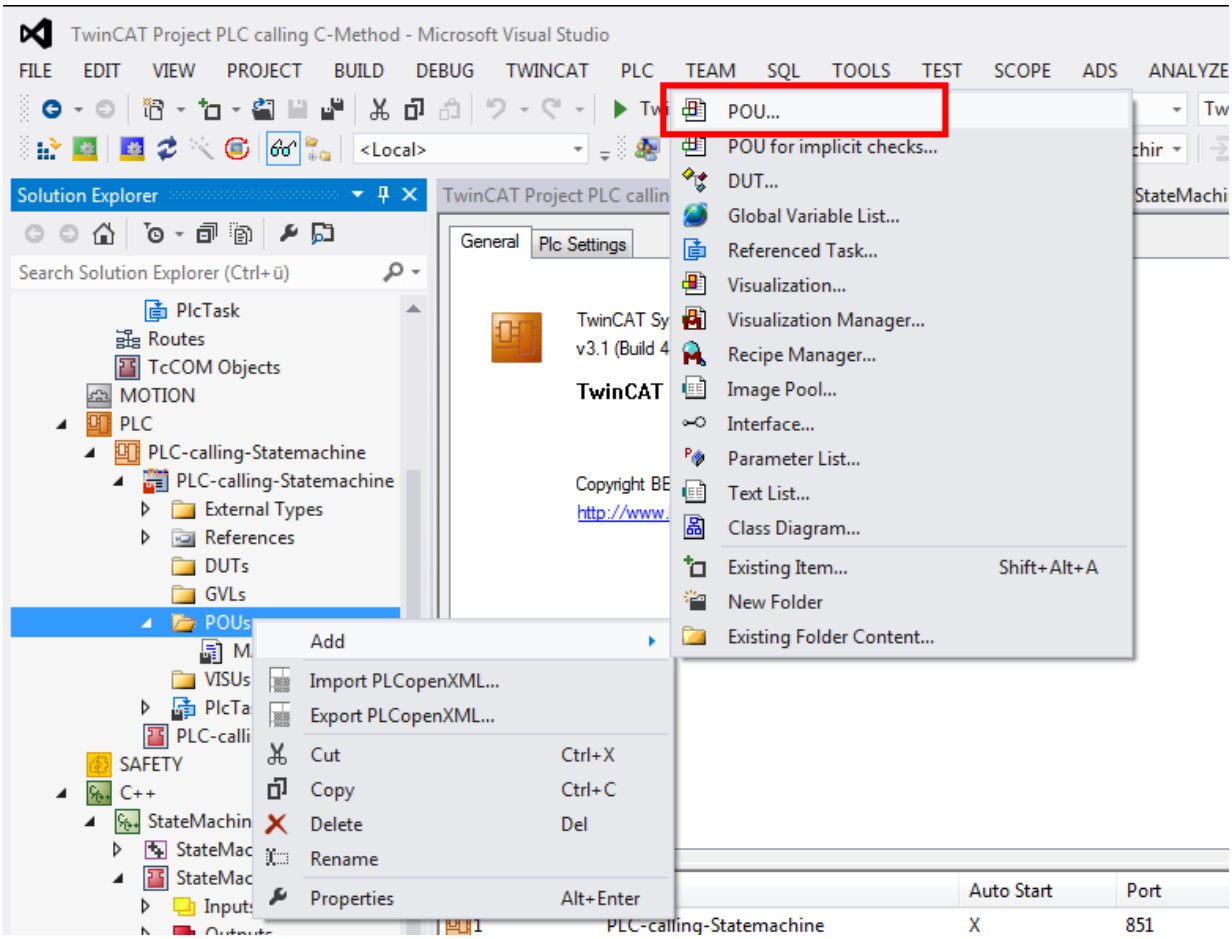


⇨ The project has been successfully created.

Step 3: add a function block (FB) (which serves as the proxy for calling the C++ module methods)

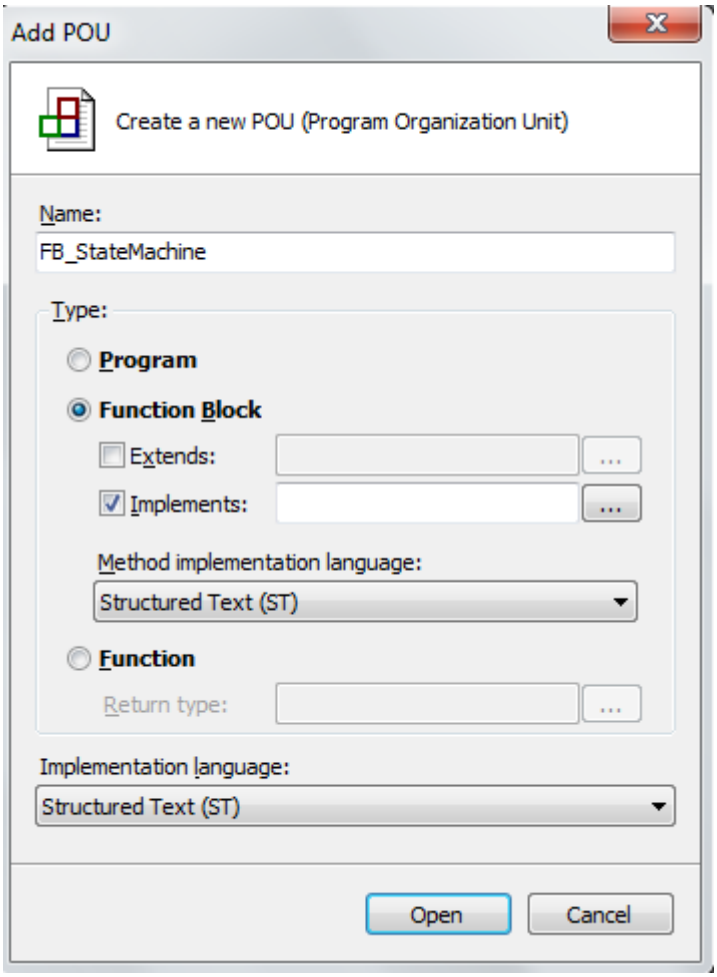
1. Right-click on **POUs**.

2. Select **Add->POU....**

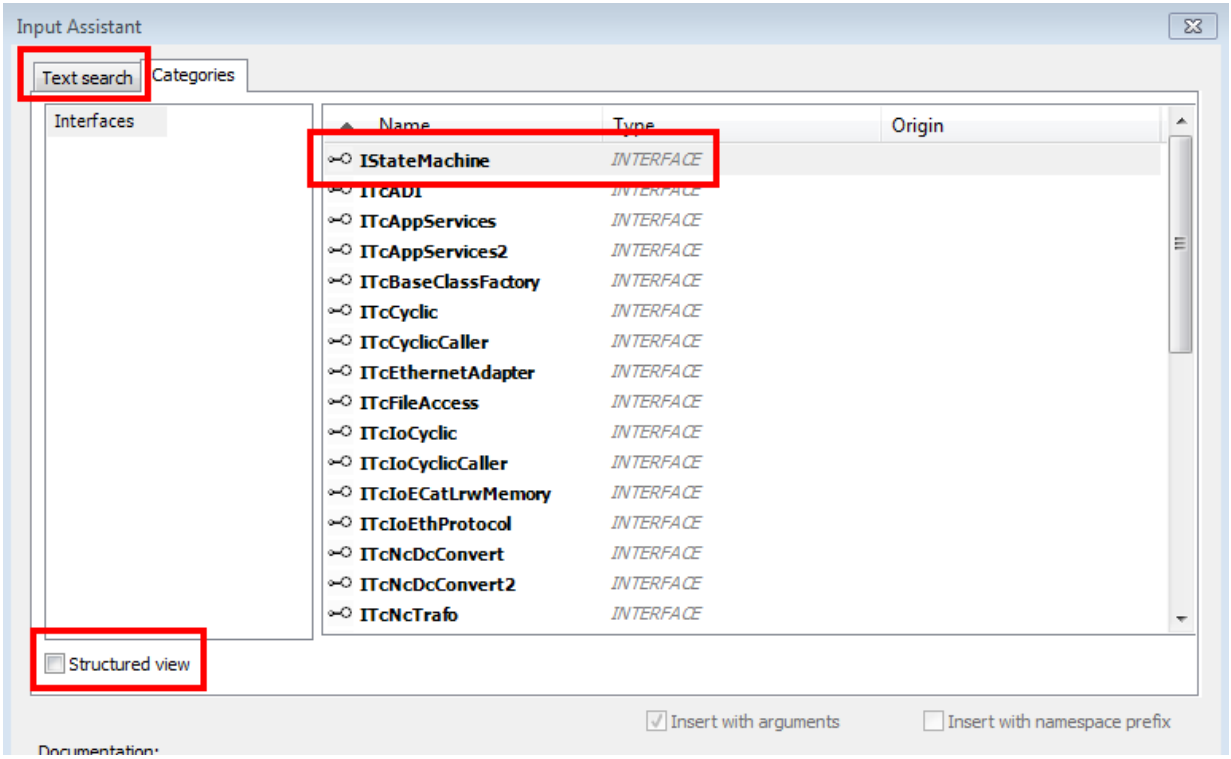


3. Define a new FB to be created, which will later act as a proxy for calling C++ classes: Enter the name of the new FB: "FB_StateMachine".

4. Select **Function Block**, then **Implements** and then click on the **...** button.



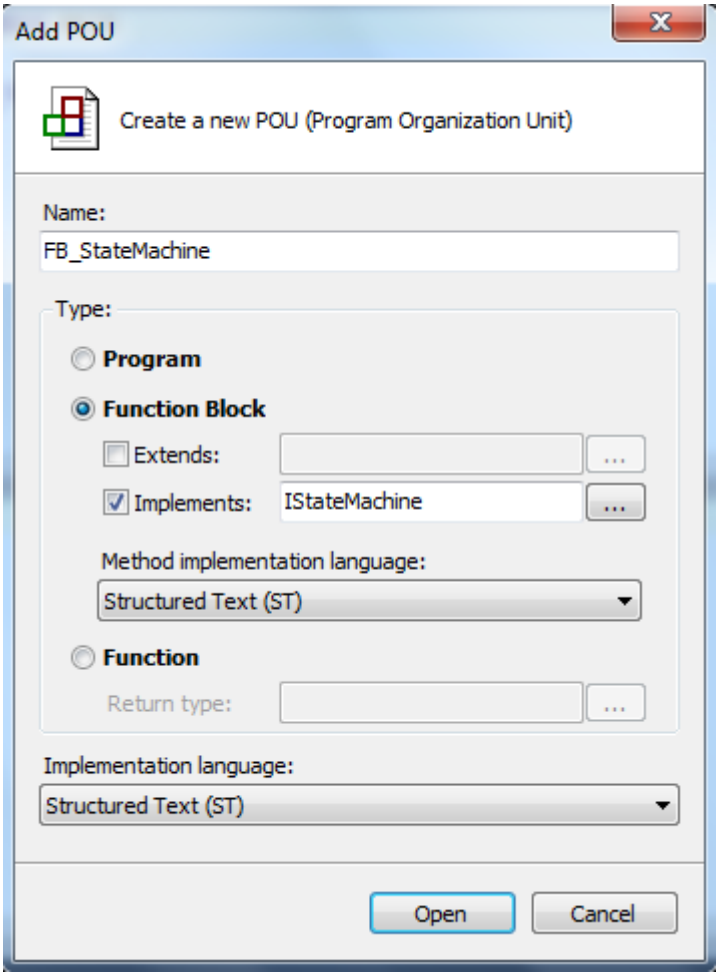
5. Select the interface either via the **Text Search** tab or the **Categories** tab by deselecting **Structured View**.



6. Select **IStateMachine** and click on **OK**.

⇨ The IStateMachine interface is then listed as the interface to be implemented.

7. Select **Structured Text (ST)** as **Method implementation language**.
8. Select **Structured Text (ST)** as implementation language.
9. End this dialog with **Open**.



⇨ You have successfully added the FB.

Step 4: Adjusting the function block interface

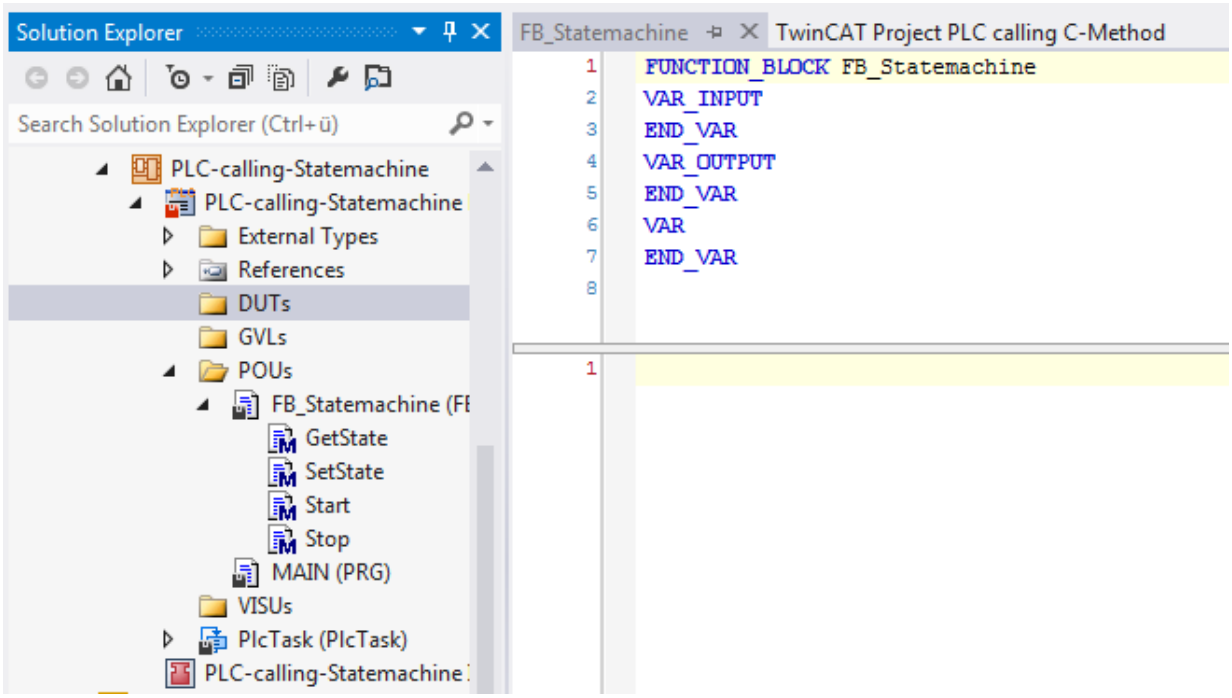
As a result of creating an FB that implements the IStateMachine interface, the wizard will create an FB with corresponding methods.

The FB_StateMachine makes 4 methods available:

- GetState
- SetState
- Start
- Stop

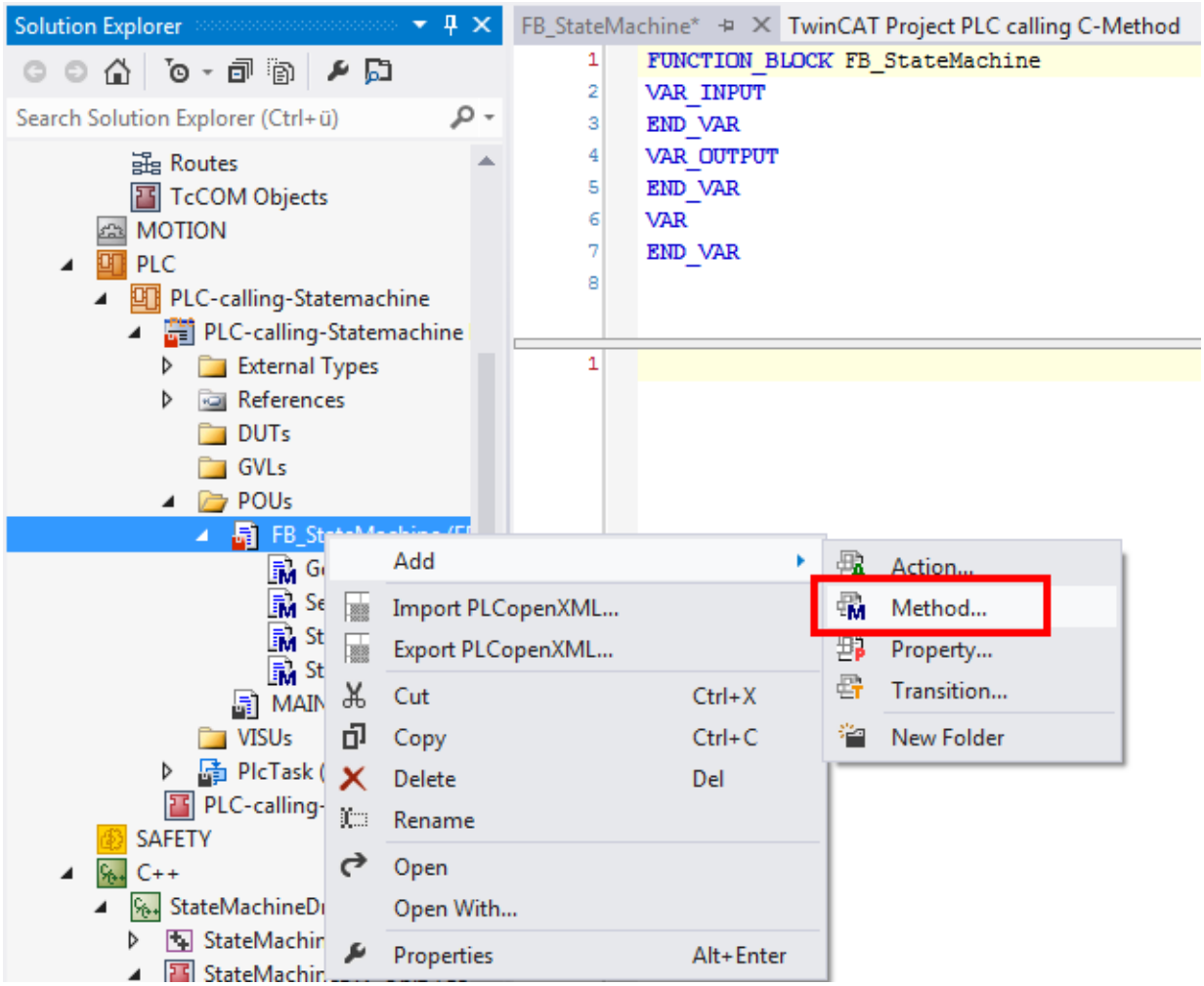
1. Delete Implements IStateMachine. Since the function block should act as proxy, it does not implement the interface itself. Therefore, it can be deleted.
2. Delete the methods TcAddRef, TcQueryInterface and TcRelease. They are not required for a proxy function block.

⇨ The result is:

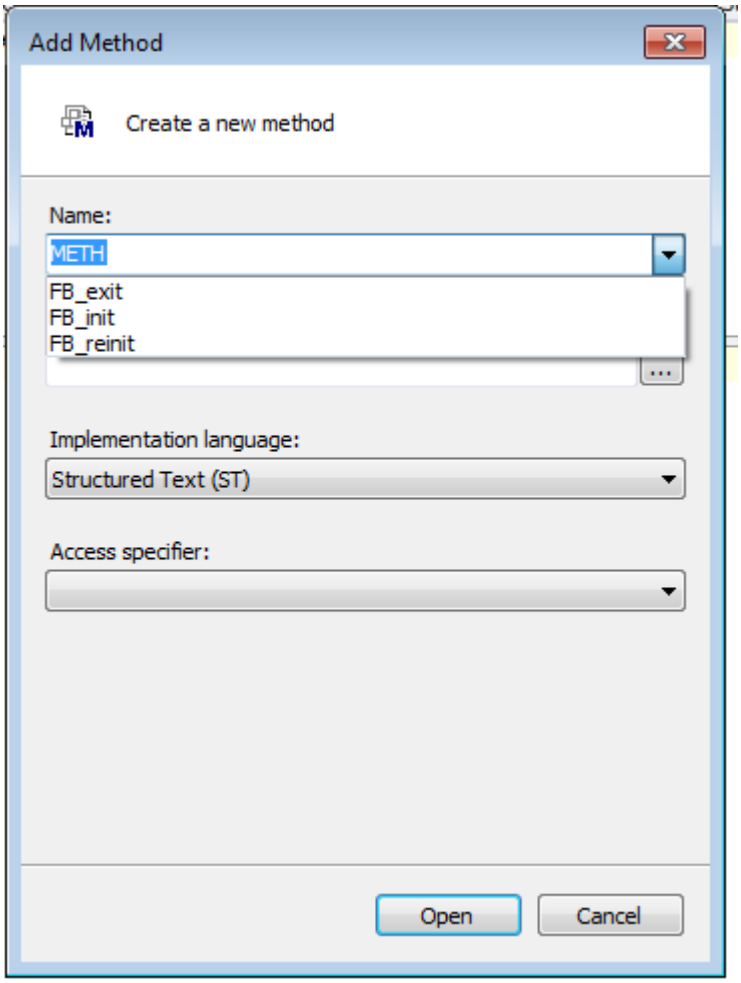


Step 5: add FB methods FB_init (Constructor) and FB_exit (Destructor)

1. Right-click on **FB_StateMachine** in the tree and select **Add / Method…**

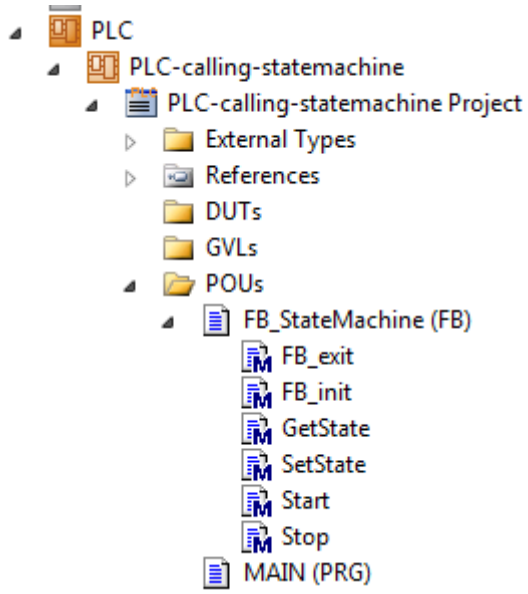


2. Add the methods FB_exit and FB_init - both with Structured Text (ST) as the implementation language. They are available as predefined name.



3. Exit the dialog in each case by clicking on **Open**.

⇨ In the end, all required methods are available:



Step 6: implement FB methods

Now all methods have to be filled with code.

NOTICE

Missing attributes lead to unexpected behavior

Attribute statements in brackets represent code to be added.

More precise information on the attributes is given in the PLC documentation.

1. Implement the variable declarations of the FB_Statemachine. The FB itself does not require cyclically executable code.

```
FB_StateMachine.FB_exit   FB_StateMachine.FB_init   FB_StateMachine*   TwinCAT Project PLC calling C-Method
    END_VAR
    VAR
        {attribute 'TcInitSymbol'}
        oidInstance : OTCID;
        ipStateMachine : IStateMachine; // interface pointer to the C++ StateMachine module instance
        hrInit : HRESULT;
    END_VAR
```

2. Implement the variable declarations and the code area of the method FB_exit.

```
FB_StateMachine.FB_exit*   FB_StateMachine.FB_init   FB_StateMachine*
    METHOD FB_exit : BOOL
    VAR_INPUT
        bInCopyCode : BOOL; // if TRUE, the exit method is called
    END_VAR

    IF NOT bInCopyCode THEN // no online change
        FW_SafeRelease(ADR(ipStateMachine));
    END_IF
```

3. Implement the variable declarations and the code area of the method FB_init.

```
FB_StateMachine.FB_exit*   FB_StateMachine.FB_init*   FB_StateMachine*   TwinCAT Project PLC calling C-Method
    METHOD FB_init : BOOL
    VAR_INPUT
        bInitRetains : BOOL; // if TRUE, the retain variables are initialized (warm start / cold start)
        bInCopyCode : BOOL;  // if TRUE, the instance afterwards gets moved into the copy code (online change)
    END_VAR

    IF NOT bInCopyCode THEN // no online change
        IF ipStateMachine = 0 THEN
            hrInit := FW_ObjMgr_GetObjectInstance(  oid:=oidInstance,
                                                    iid:=TC_GLOBAL_IID_LIST.IID_IStateMachine,
                                                    pipUnk:=ADR(ipStateMachine) );
        END_IF
    END_IF
```

4. Implement the variable declaration and the code area of the method GetState (the generated pragmas can be deleted as they are not required for a proxy FB).

FB_StateMachine.GetState* | FB_StateMachine.FB_exit* | FB_StateMachine.FB_init*

```
METHOD GetState : HRESULT
VAR_INPUT
    pState  : POINTER TO INT;
END_VAR
```

```
IF (ipStateMachine <> 0) THEN
    GetState:= ipStateMachine.GetState(pState);
END_IF
```

5. Implement the variable declaration and the code area of the method SetState (the generated pragmas can be deleted as they are not required for a proxy FB).

FB_StateMachine.SetState* | FB_StateMachine.GetState* | FB_StateMachine.FB_exit*

```
METHOD SetState : HRESULT
VAR_INPUT
    State   : INT;
END_VAR
```

```
IF (ipStateMachine <> 0) THEN
    SetState:= ipStateMachine.SetState(State);
END_IF
```

6. Implement the variable declaration and the code area of the method Start (the generated pragmas can be deleted as they are not required for a proxy FB).

FB_StateMachine.Start | FB_StateMachine.SetState* | FB_StateMachine.GetState*

```
METHOD Start : HRESULT
```

```
IF (ipStateMachine <> 0) THEN
    Start:= ipStateMachine.Start();
END_IF
```

7. Implement the variable declaration and the code area of the method Stop (the generated pragmas can be deleted as they are not required for a proxy FB).

FB_StateMachine.Stop | FB_StateMachine.Start | FB_StateMachine.SetState*

```
METHOD Stop : HRESULT
```

```
IF (ipStateMachine <> 0) THEN
    Stop:= ipStateMachine.Stop();
END_IF
```

⇨ The implementation of the FB_StateMachine, which acts as the proxy for calling the C++ module instance, is completed.

Step 7: Calling FB in PLC

The FB_StateMachine is now called in the POU MAIN.

This simple sample acts as follows:

- Cyclic incrementation of a PLC counter nCounter
- If nCounter = 500, the C++ StateMachine is started with the state "1" to increment its internal C++ counter. Then read the state of C++ using GetState().
- If nCounter = 1000, the C++ state machine is set to the state "2" in order to decrement its internal C++ counter. Then read the state of C++ using GetState().
- If nCounter = 1500, the C++ StateMachine is stopped. The PLC nCounter is also set to "0", so that everything starts again from the beginning.

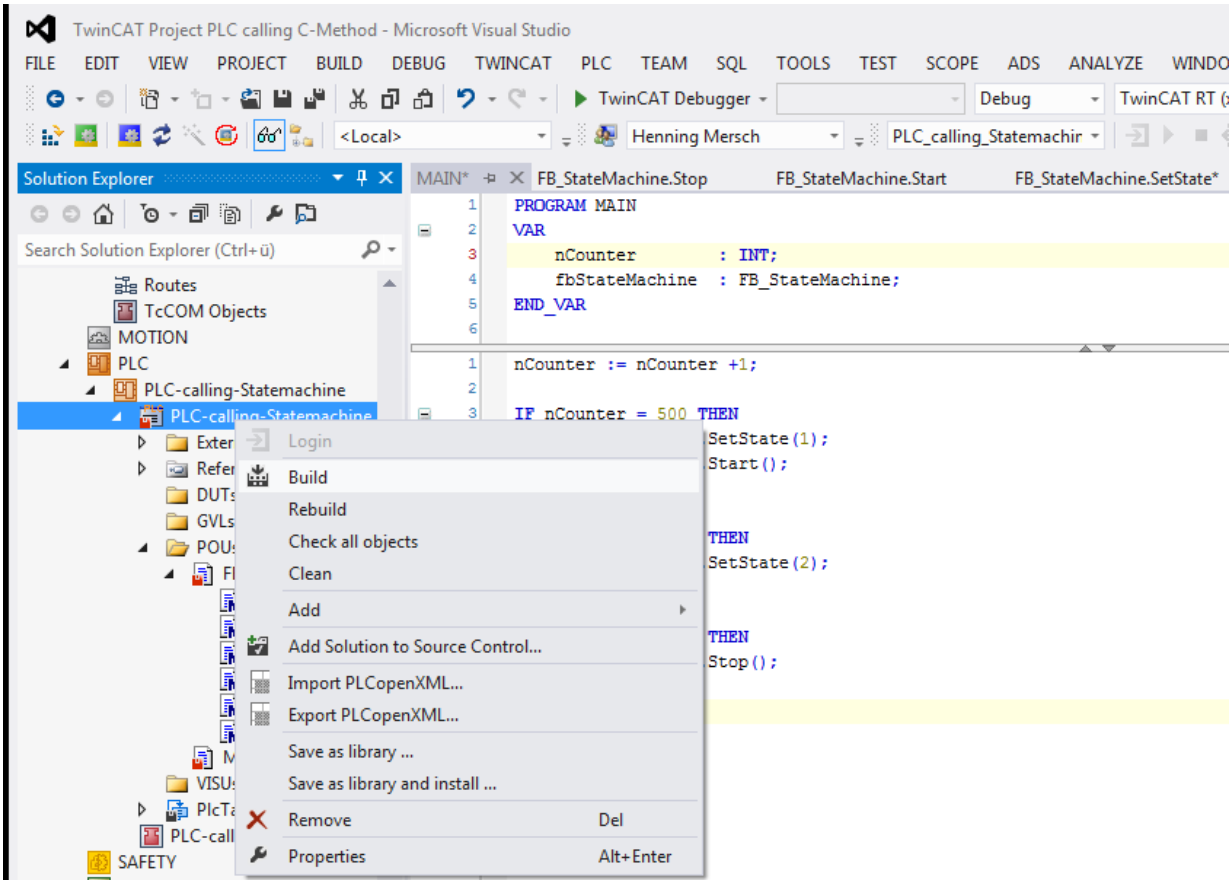
MAIN | TwinCAT Project PLC calling C-Method | FB_StateMachine.GetState

```
1  PROGRAM MAIN
2  VAR
3      nCounter        : INT;
4      nCurrentState   : INT;
5      fbStateMachine  : FB_StateMachine;
6  END_VAR
7
```

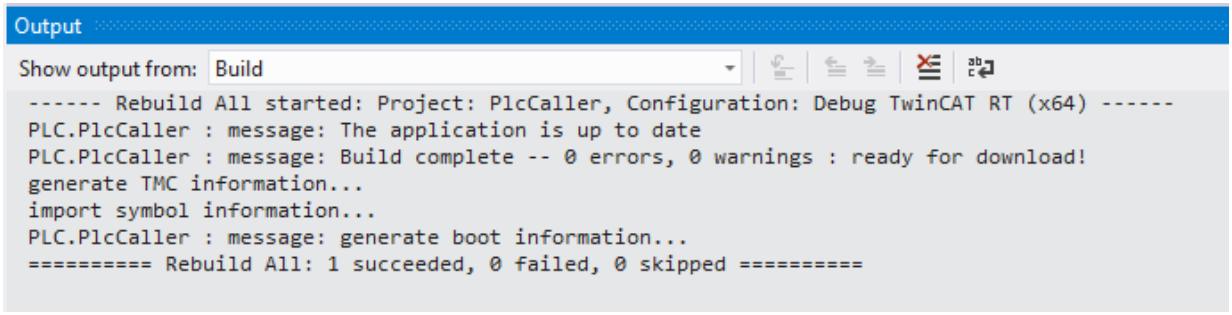
```
1  nCounter := nCounter +1;
2
3  IF nCounter = 500 THEN
4      fbStateMachine.SetState(1);
5      fbStateMachine.GetState(ADR(nCurrentState));
6      fbStateMachine.Start();
7  END_IF
8
9  IF nCounter = 1000 THEN
10     fbStateMachine.SetState(2);
11     fbStateMachine.GetState(ADR(nCurrentState));
12 END_IF
13
14 IF nCounter = 1500 THEN
15     fbStateMachine.Stop();
16     nCounter := 0;
17 END_IF
```

Step 8: compile PLC code

1. Right-click on the PLC project and click on **Build**.



⇨ The compilation result shows "1 succeeded - 0 failed".



Step 9: link PLC FB with C++ instance

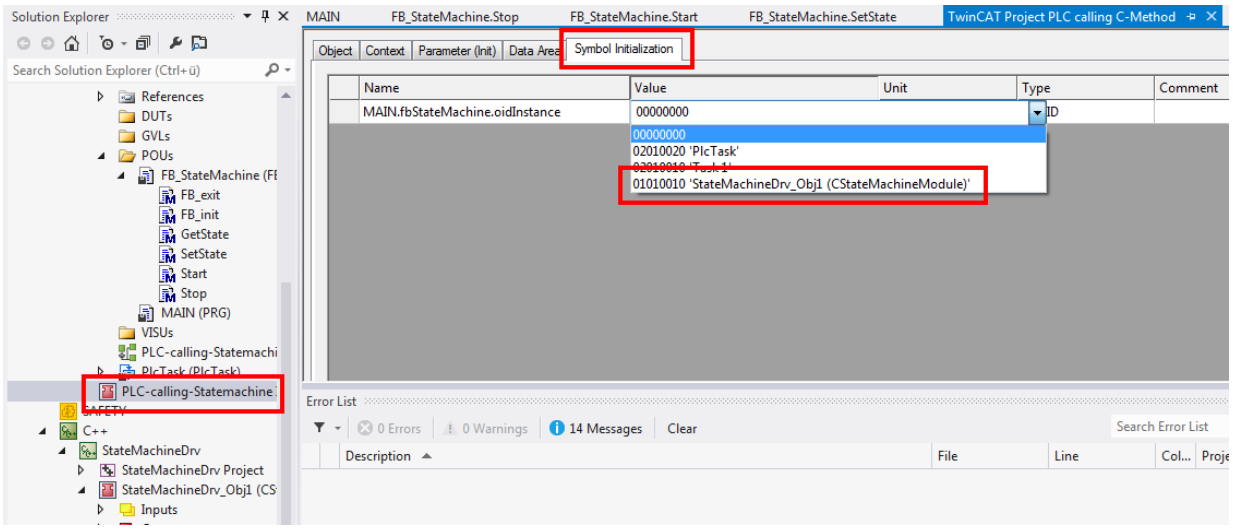
The benefits of all previous steps now become apparent:

The PLC FB FB_StateMachine can be configured with regard to linking with every instance of the C++ module StateMachine. This is a very flexible and powerful method of connecting PLC and C++ modules on the machine with each other.

1. Navigate to the instance of the PLC module in the left-hand tree and select the **Symbol Initialization** tab on the right-hand side.

   ⇨ All instances of FB_StateMachine are listed; in this sample we have only defined one FB instance in POU MAIN.

2. Select the drop-down field **Value** and then the C++ module instance that is to be linked to the FB instance.

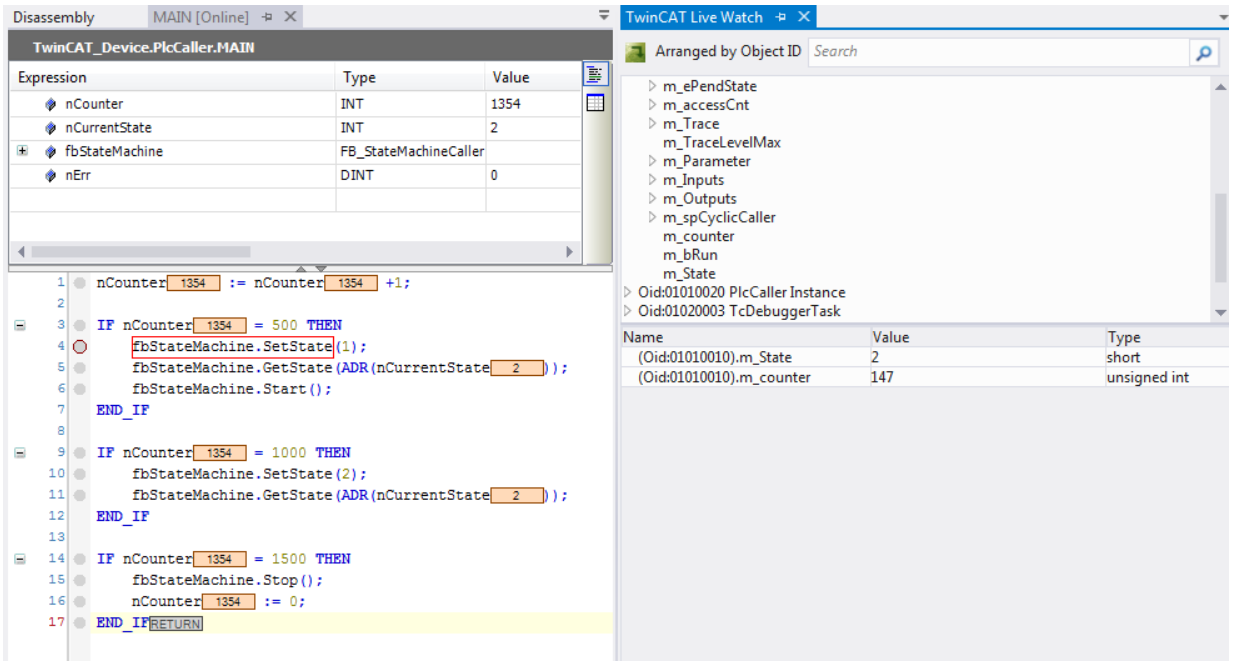


⇨ PLC and C++ module are connected to each other.

Step 10: observe the execution of the two modules, PLC and C++

Following the activation of the TwinCAT configuration and the downloading and starting of the PLC code, the execution of the two codes, PLC and C++, is simple to observe:

1. After the Login and Start of the PLC project, the editor is already in online mode (left-hand side – see following illustration).
2. In order to be able to access online variables of the C++ module, activate the C++ debugging [▶ 70] and follow the steps in the quick start [▶ 58] in order to start the debugging (right-hand side of the following illustration).



15.10 Sample11a: Module communication: Method call C++ module to C++ module

This article describes how TwinCAT 3 C++ modules could communicate via method calls. The method protects the data with a critical section thus the access could be initiated from different contexts / tasks.

Download

Here you can access the source code for this sample**:**

https://github.com/Beckhoff/TC1300_Samples/tree/main/S11a-Mod2ModCS

1. Using a Visual Studio with TwinCAT installed, open the project via **Open Project ....**
2. If you want to run the project on Windows or TwinCAT/BSD® (not required for Beckhoff RT Linux®): Configure signing for this project by right-clicking on the project on the page, selecting **Project -> Properties -> TC Sign**, enabling TwinCAT signing, and configuring your certificate and password if necessary.
   Further information on signing C++ projects can be found in the chapter TwinCAT [▶ 21].
3. Select your target system.
4. Build the sample (e.g. **Build->Build Solution**).
5. Activate the configuration by clicking  .

⇨ The sample is ready for operation.

Description

The project contains three modules:

- The instance of the CModuleDataProvider class hosts the data and protects against access via the Retrieve and Store methods through a Critical Section.
- The instance of the module class CModuleDataRead reads the data from the DataProvider by calling the Retrieve method.
- The instance of the module class CModuleDataWrite writes the data from the DataProvider by calling the Store method.

The read/write instances are configured for access to the DataProvider instance, which can be seen in the **Interface Pointer** menu on the instance configuration.
The context (task), in which the instances are to be executed, can also be configured there. In this sample two tasks are used, TaskRead and TaskWrite.
The DataWriteCounterModul parameters of CModuleDataWrite and DataReadCounterModulo (CModuleDataRead) enable the moment to be determined, at which the module instances initiate the access.

CriticalSections are described in the SDK in TcRtInterfaces.h and are therefore intended for the real-time context.

15.11 Sample12: module communication: Using IO mapping

This article describes how two TwinCAT 3 C++ modules could communicate via the usual IO mapping of TwinCAT 3: Two instances are linked via IO mapping and access the variable value periodically.

Download

Here you can access the source code for this sample:

https://github.com/Beckhoff/TC1300_Samples/tree/main/S12-Mod2ModIOMapping

1. Using a Visual Studio with TwinCAT installed, open the project via **Open Project ....**
2. If you want to run the project on Windows or TwinCAT/BSD® (not required for Beckhoff RT Linux®): Configure signing for this project by right-clicking on the project on the page, selecting **Project -> Properties -> TC Sign**, enabling TwinCAT signing, and configuring your certificate and password if necessary.
   Further information on signing C++ projects can be found in the chapter TwinCAT [▶ 21].
3. Select your target system.
4. Build the sample (e.g. **Build->Build Solution**).

5. Activate the configuration by clicking  .

⇨ The sample is ready for operation.

Description

Both instances are realized by means of a module class ModuleInToOut: The class cyclically copies its input data area Value to its output data area Value.
The Front instance acts as front end for the user. An input Value is transferred to the output Value via the method cycleupdate(). This output value of Front is assigned (linked) to the input "value" of the Back instance.
The Back instance copies the input Value to its output Value, which can be monitored by the user (see the following quick start steps to start debugging: Debugging a TwinCAT 3 C++ project)

Ultimately, the user can define the input value of the Front instance and observe the output value of Back.

15.12 Sample13: Module communication: Method call C++ module to PLC module

This article describes how a TwinCAT C++ module calls a methods of a PLC function block via the TcCOM interface.

Download

You can obtain the source code for this sample here: https://github.com/Beckhoff/TC1300_Samples/tree/main/S13-CppToPLC

1. Using a Visual Studio with TwinCAT installed, open the project via **Open Project ....**
2. If you want to run the project on Windows or TwinCAT/BSD® (not required for Beckhoff RT Linux®): Configure signing for this project by right-clicking on the project on the page, selecting **Project -> Properties -> TC Sign**, enabling TwinCAT signing, and configuring your certificate and password if necessary.
   Further information on signing C++ projects can be found in the chapter TwinCAT [▶ 21].
3. Select your target system.
4. Build the sample (e.g. **Build->Build Solution**).
5. Activate the configuration by clicking  .

⇨ The sample is ready for operation.

Description

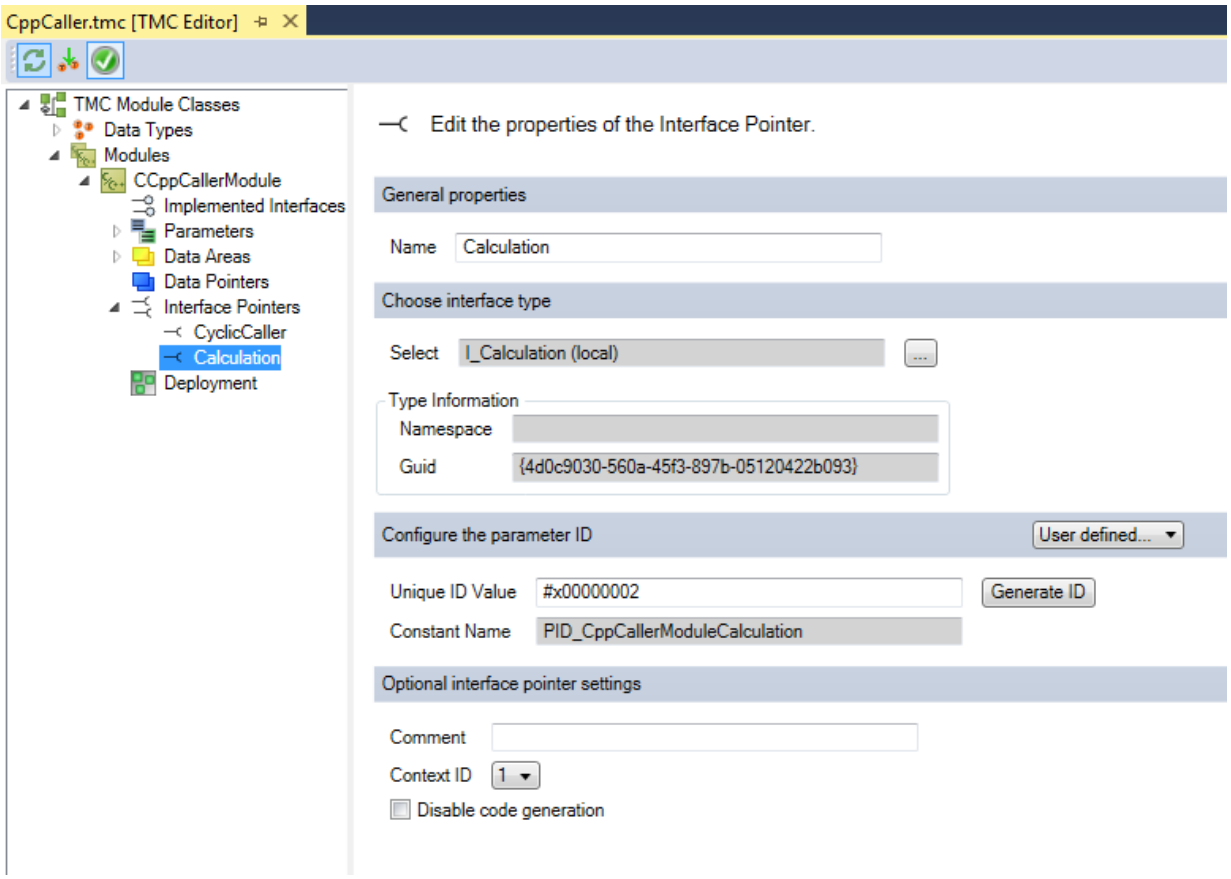
This sample provides for communication from a C++ module to a function block of a PLC by means of method call. To this end a TcCOM interface is defined that is offered by the PLC and used by the C++ module.

The PLC page as a provider in the process corresponds to the corresponding project of the TcCOM Sample 01 [▶ 304], where an PLC is considered after PLC communication. Here a Caller is now provided in C++, which uses the same interface.

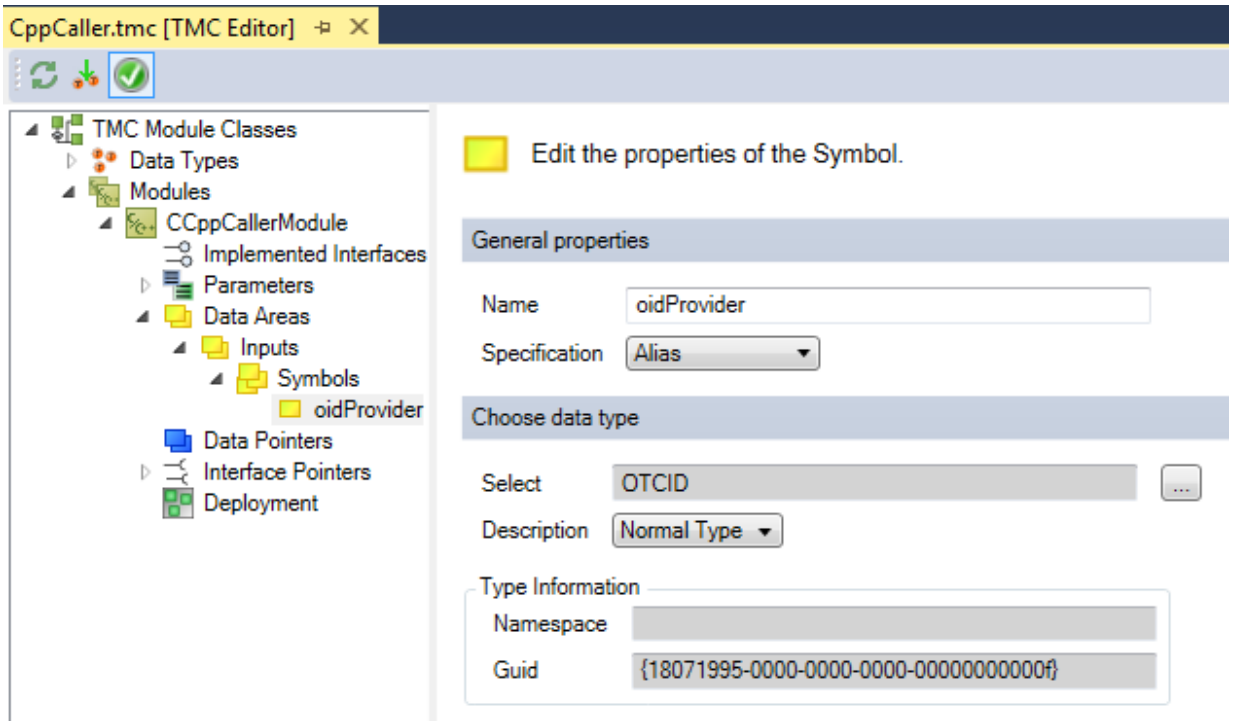
The PLC page adopted by TcCOM Sample 01 [▶ 304]. The function block registered there as TcCOM module offers the object ID allocated to it as an output variable.
It is the C++ module's task to make the offered interface of this function block accessible.

✓ A C++ project with a Cycle IO module is assumed.

1. In the TMC editor, create an interface pointer of the type I_Calculation with the name Calculationn). Later access occurs via this.

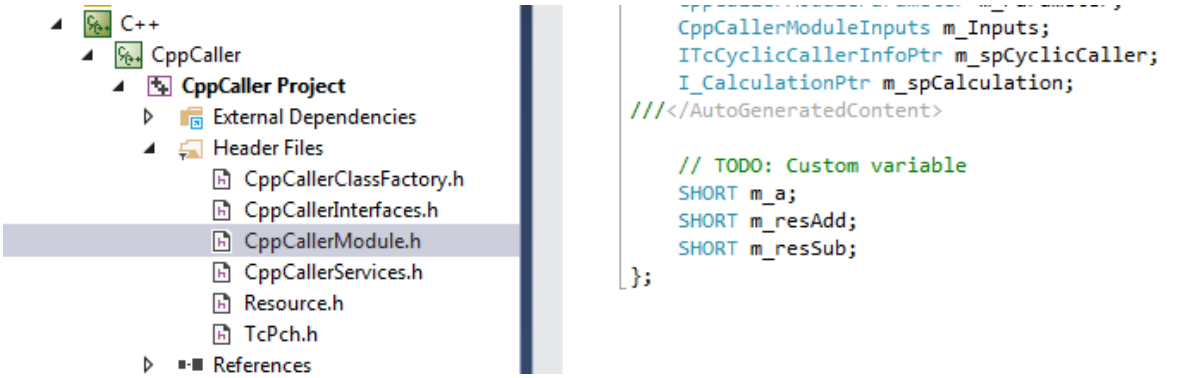


2. The Data Area Inputs have already been created by the module wizard with the type Input-Destination. Here in the TMC editor you create an input of the type OTCID with the name oidProvider, via which the Object ID will be linked from the PLC later.

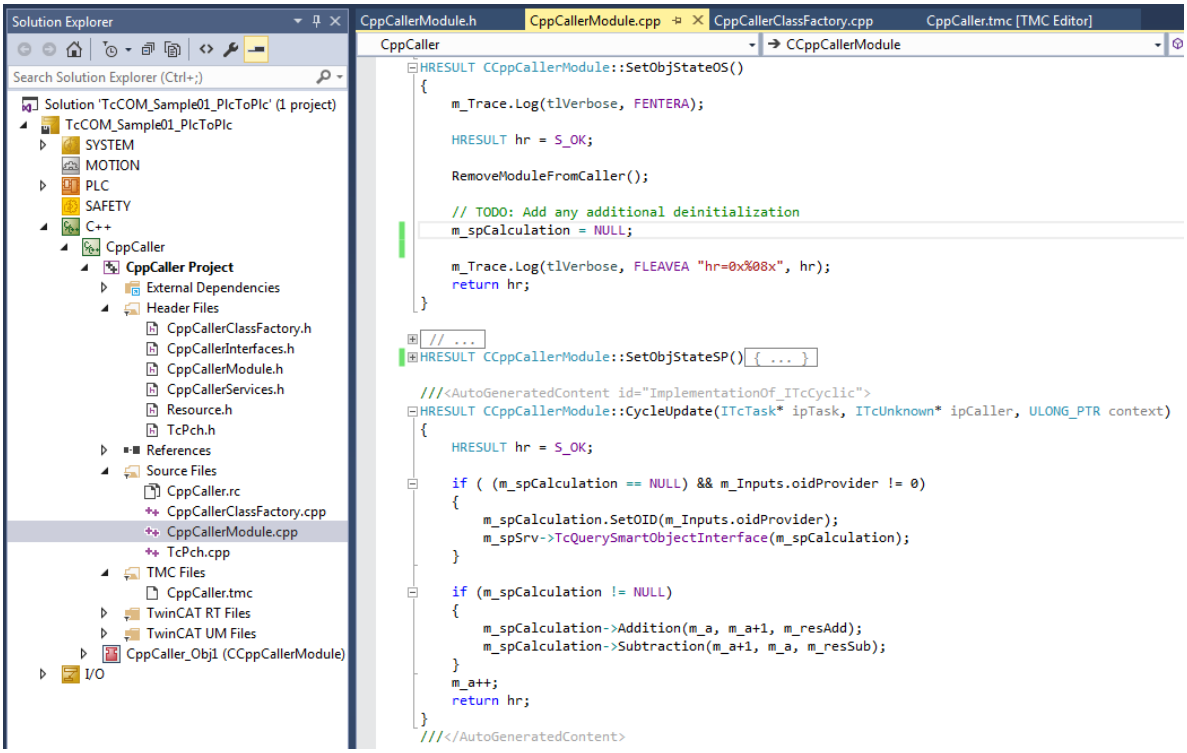


3. All other symbols are irrelevant for the sample and can be deleted.

⇨ The TMC-Code-Generator prepares the code accordingly.
In the header of the module some variables are created in order to carry out the methods calls later.



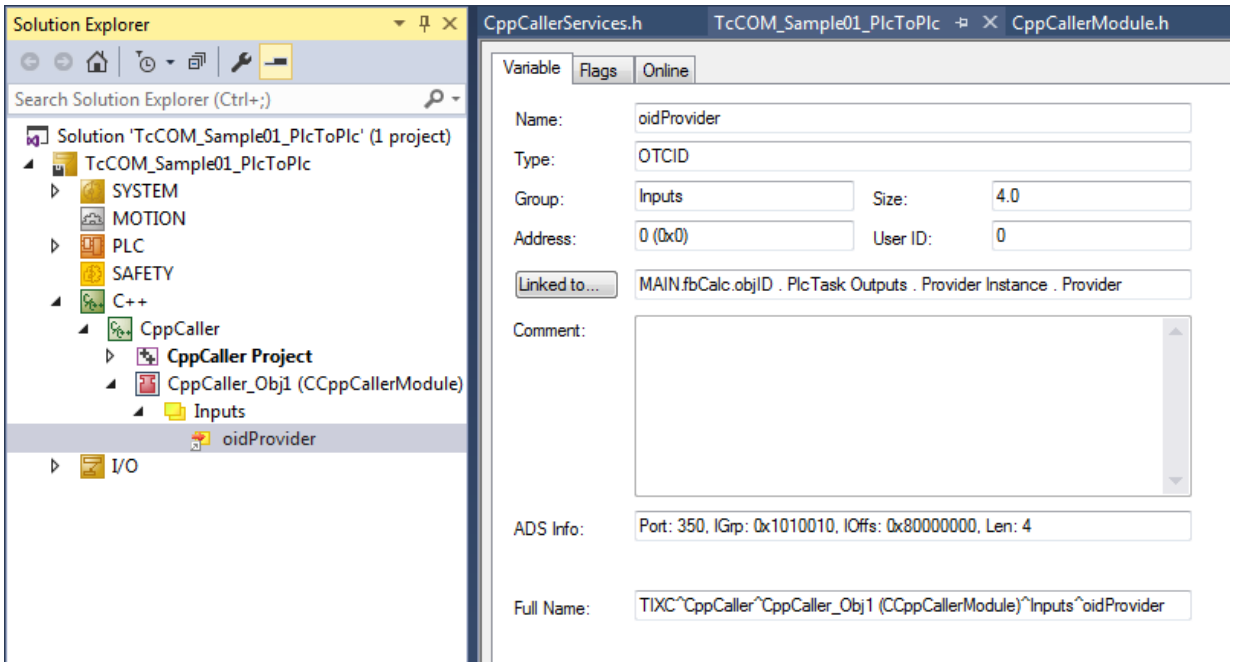
In the actual code of the module in CycleUpdate() the interface pointer is set using the object ID transmitted from the PLC. It is important that this happens in the CycleUpdate() and thus in real-time context, since the PLC must first provide the function block.
When this has taken place once, the methods can be called.



In addition, as can be seen above, the interface pointer is cleared when the program shuts down. This happens in the SetObjStateOS method.

4. Now build the C++ project.
5. Create an instance of the module.

6. Connect the input of the C++ module to the output of the PLC.



⇨ The project can be started. When the PLC is running, the OID is made known through the mapping to the C++ instance. Once this has occurred, the method can be called.

15.13 Sample19: Synchronous File Access

This article describes how to implement a TwinCAT 3 C++ module which accesses files on the hard disk within the startup of a module, thus within the real-time environment.

Download

You can obtain the source code for this sample here: https://github.com/Beckhoff/TC1300_Samples/tree/main/S19-FileIOSync

1. Using a Visual Studio with TwinCAT installed, open the project via **Open Project ....**
2. If you want to run the project on Windows or TwinCAT/BSD® (not required for Beckhoff RT Linux®): Configure signing for this project by right-clicking on the project on the page, selecting **Project -> Properties -> TC Sign**, enabling TwinCAT signing, and configuring your certificate and password if necessary.
   Further information on signing C++ projects can be found in the chapter TwinCAT [▶ 21].
3. Select your target system.
4. Build the sample (e.g. **Build->Build Solution**).
5. Activate the configuration by clicking .

⇨ The sample is ready for operation.

All source code that is not automatically generated by the wizard is marked with the comment start flag "// sample code" and the comment end flag "//sample code end".
This allows you to search for these strings in the files to get an idea of the details.

Description

This sample describes file access via the TwinCAT interface ITCFileAccess. The access is synchronous and can be used for reading a configuration during startup of a module, for example.

The sample contains a C++ module TcFileTestDrv with an instance of this module TcFileTestDrv_Obj1.
In this sample, the file access takes place during the transition PREOP to SAFEOP, i.e. in the SetObjStatePS() method.

Helper methods encapsulate file handling.
First, general file information and a directory list is printed to the log device of TwinCAT 3. Then, a file *%TC_TARGETPATH%DefaultConfig.xml* (normally *C:\TwinCAT\3.1\Target\DefaultConfig.xml*) is copied to %TC_TARGETPATH%DefaultConfig.xml.bak.

To access the log entries, see the **Error List** tab of the TwinCAT 3 output window.
You can set the amount of information by changing the TraceLevelMax variable in the TcFileTestDrv_obj1 instance in the **Parameters (Init)** tab.

15.14 Sample20: FileIO-Write

This article describes the implementation of TwinCAT 3 C++ modules, which write (process) values to a file.

The writing of the file is triggered by a deterministic cycle - the execution of File IO is decoupled (asynchronous), i.e.: the deterministic cycle continues to run and is not hindered by writing to the file.

Download

You can obtain the source code for this sample here: https://github.com/Beckhoff/TC1300_Samples/tree/main/S20-FileIOWrite

1. Using a Visual Studio with TwinCAT installed, open the project via **Open Project ....**
2. If you want to run the project on Windows or TwinCAT/BSD® (not required for Beckhoff RT Linux®): Configure signing for this project by right-clicking on the project on the page, selecting **Project -> Properties -> TC Sign**, enabling TwinCAT signing, and configuring your certificate and password if necessary.
   Further information on signing C++ projects can be found in the chapter TwinCAT [▶ 21].
3. Select your target system.
4. Build the sample (e.g. **Build->Build Solution**).
5. Activate the configuration by clicking  .

⇨ The sample is ready for operation.

Description

The sample includes an instance of TcAsyncWritingModule, which writes data to the AsyncTest.txt file in BOOTPRJPATH directory (Windows: *C:\TwinCAT\3.1\Boot; TwinCAT/BSD: /usr/local/etc/TwinCAT/3.1/Boot*).

TcAsyncBufferWritingModule has two buffers (m_Buffer1, m_Buffer2), which are alternately filled with current data. The member variable m_pBufferFill points to the buffer that is currently to be filled. Once a buffer is filled, the member variable m_pBufferWrite is set such that it points to the full buffer.

These data are written to a file with the aid of TcFsmFileWriter.

Note that the file has no human-readable content, such as ASCII characters; in this sample, but binary data are written to the file.

15.15 Sample20a: FileIO-Cyclic Read / Write

This article is a more comprehensive sample than S20 and S19. It demonstrates cyclic read and/or write access to files from a TwinCAT 3 C++ module.

Download

Here you can access the source code for this sample:

https://github.com/Beckhoff/TC1300_Samples/tree/main/S20a-FileIO-CyclicReadWrite

1. Using a Visual Studio with TwinCAT installed, open the project via **Open Project ....**

2. If you want to run the project on Windows or TwinCAT/BSD® (not required for Beckhoff RT Linux®): Configure signing for this project by right-clicking on the project on the page, selecting **Project -> Properties -> TC Sign**, enabling TwinCAT signing, and configuring your certificate and password if necessary.
   Further information on signing C++ projects can be found in the chapter TwinCAT [▶ 21].
3. Select your target system.
4. Build the sample (e.g. **Build->Build Solution**).
5. Activate the configuration by clicking  .

⇨ The sample is ready for operation.

Description

The sample describes how to access files for reading and/or writing using the CycleUpdate method, i.e. in a cyclic manner.

This sample contains the following projects and module instances.

- A static library (TcAsyncFileIo) offers the file access.
  The code for the file access can be shared, therefore this code is located in a static library that is used by the driver projects.
- A driver (TcAsyncBufferReadingDrv) provides two instances:
  - ReadingModule: uses the static library to read the AsyncTest.txt file.
  - WriteDetectModule: detects write operations and initiates read operations.
- A driver (TcAsyncBufferWritingDrv) provides one instance:
  - WriteModule: uses the static library to write the AsyncTest.txt file.

When starting the sample, the writing module begins to write data to the file located in the boot project path (Windows: *C:\TwinCAT\3.1\Boot\AsyncTest.txt; TwinCAT/BSD: /usr/local/etc/TwinCAT/3.1/Boot/AsyncTest.txt*). The input variable bDisableWriting can be used to prevent writing.

The objects are connected to one another: once the writing is complete, the WritingModule triggers the DetectModule of TcAsyncBufferReadingDrv. As a result of this, the ReadingModule initiates a reading procedure.

Observe the nBytesWritten / nBytesRead output variables of the WritingModule / ReadingModule. Over and above that, protocol messages are generated at verbose level. As before, these can be configured with the help of the TraceLevelMax parameter of the module.

- A driver (TcAsyncFileFindDrv) provides one instance
  - FileFindModule: list the files of a directory with the help of the static library.

Initiate the action with the help of the input variable bExecute. The FilePath parameter contains the directory whose files are to be listed (Windows: *c:\TwinCAT\3.1\Boot*; TwinCAT/BSD: /usr/local/etc/TwinCAT/3.1/Boot/**).

Observe the sequence tracking (verbose protocol level) with regard to the list of files found.

Understanding the sample

The project TcAsyncFileIO contains various classes in a static library. This library is used by driver projects for reading and writing.

Each class is intended for a file access operation such as Open / Read / Write / List / Close / .... Since execution takes place in a cyclic real-time context, each operation has a status, and the class encapsulates this state machine.

As an entry point for understanding the file access, begin with the classes TcFsmFileReader and TcFsmFileWriter.

If too many history tracking messages occur, which hamper understanding of the sample, you can disable modules!

See also

Sample S19 [▶ 289]

Sample S20 [▶ 290]

Sample S25 [▶ 296]

Interface ITcFileAccess [▶ 164] / Interface ITcFileAccessAsync [▶ 171]

15.16 Sample22: Automation Device Driver (ADD): Access DPRAM

This article describes how to implement a TwinCAT 3 C++ driver which acts as a TwinCAT Automation Device Driver (ADD) accessing the DPRAM.

Download

Here you can access the source code for this sample:

https://github.com/Beckhoff/TC1300_Samples/tree/main/S22-ADD

NOTICE

Configuration details

Read the configurations details below prior to the activation.

1. Open the zip file that it contains in TwinCAT 3 by clicking on **Open Project …**.
2. Select your target system.
3. Build the sample on your local machine (e.g. **Build->Build Solution**).
4. Note the actions listed on this page under **Configuration**.
5. Activate the configuration by clicking  .

⇨ The sample is ready for operation.

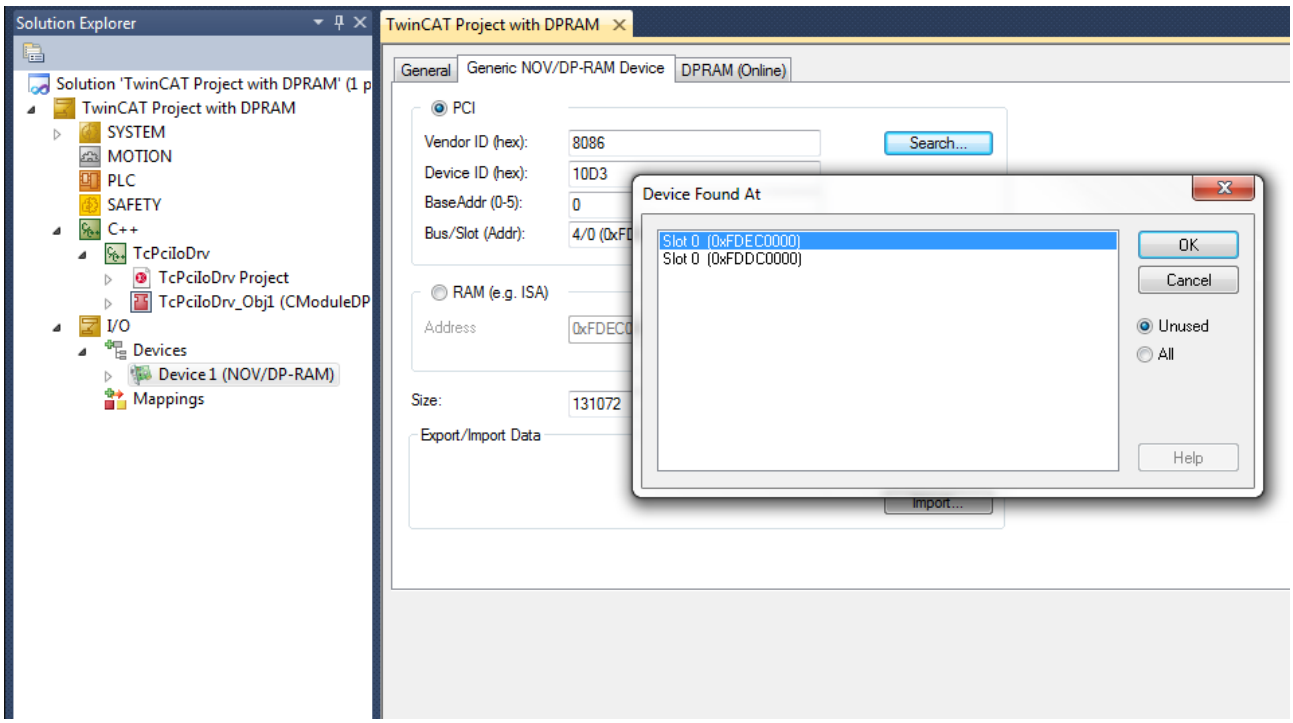
Description

This sample is intended to switch the Link Detect bit of the network adapter (i.e. of an CX5010) cyclically on and off.

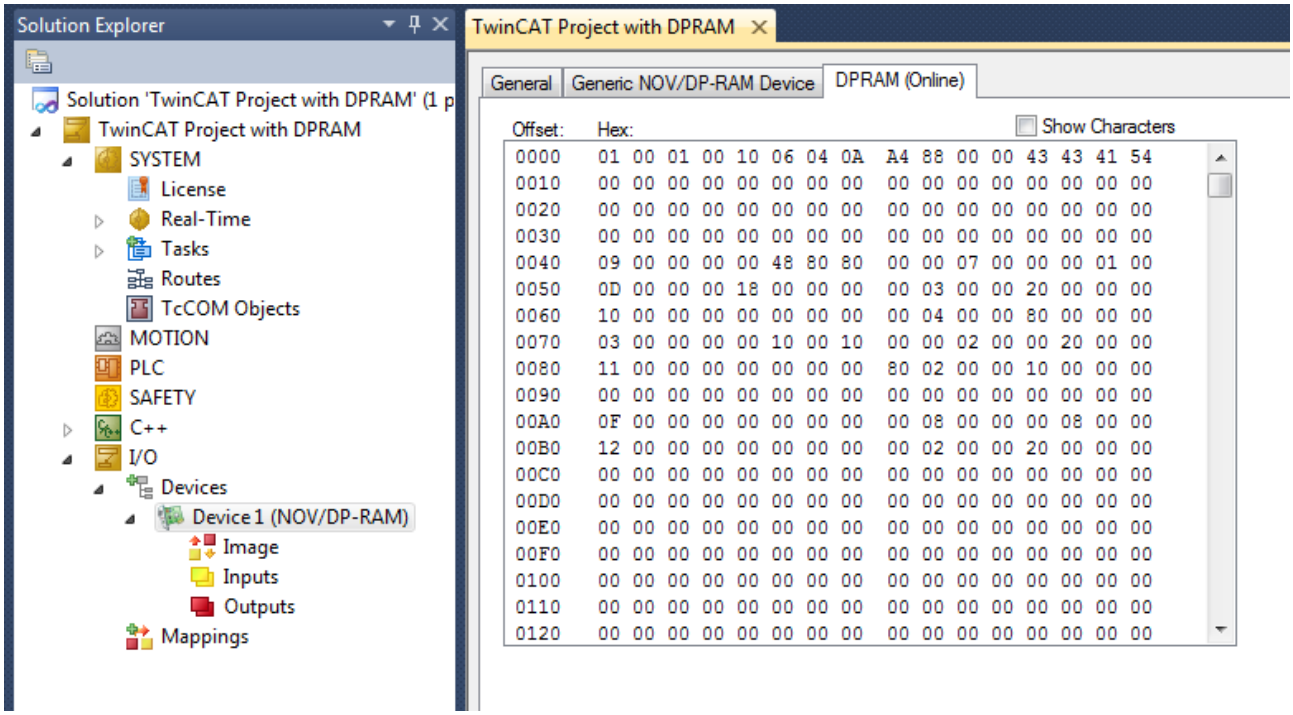
The C++ module is connected to the NOV/DP-RAM device via the PciDeviceAdi interface pointer of the C++ module.

Configuration

To make the sample work, the hardware addresses must be configured to match your own hardware. Check the PCI configuration:



To check whether the communication with NOV/DPRAM is set up correctly, use the DPRAM (Online) view:



15.17 Sample23: Structured Exception Handling (SEH)

This article describes the use of Structured Exception Handling (SEH) on the basis of five variants.

Download

Here you can access the source code for this sample:

https://github.com/Beckhoff/TC1300_Samples/tree/main/S23-SEH

1. Using a Visual Studio with TwinCAT installed, open the project via **Open Project ....**

2. If you want to run the project on Windows or TwinCAT/BSD® (not required for Beckhoff RT Linux®): Configure signing for this project by right-clicking on the project on the page, selecting **Project -> Properties -> TC Sign**, enabling TwinCAT signing, and configuring your certificate and password if necessary.
   Further information on signing C++ projects can be found in the chapter TwinCAT [▶ 21].
3. Select your target system.
4. Build the sample (e.g. **Build->Build Solution**).
5. Activate the configuration by clicking .

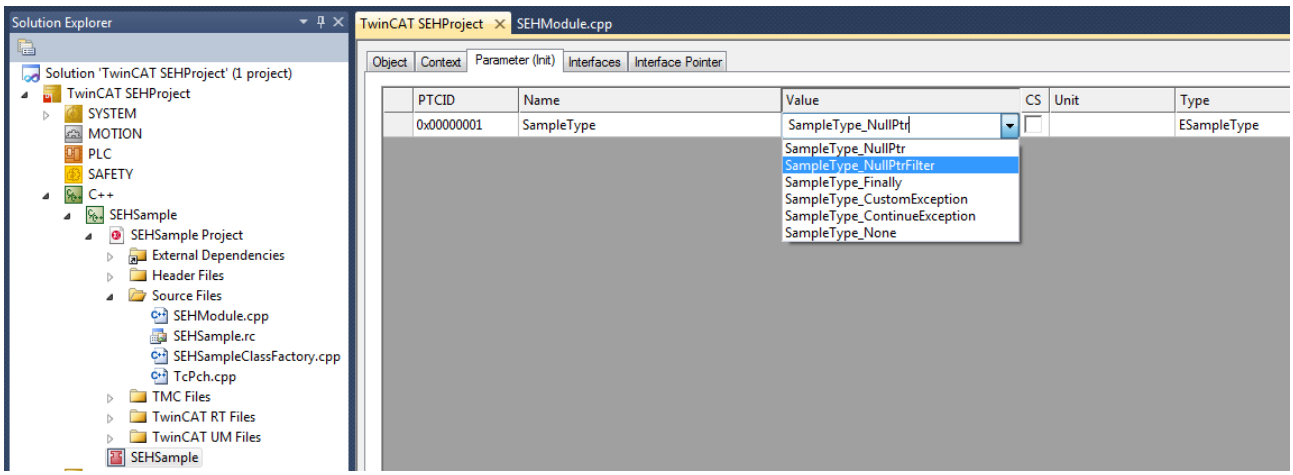
⇨ The sample is ready for operation.

Description

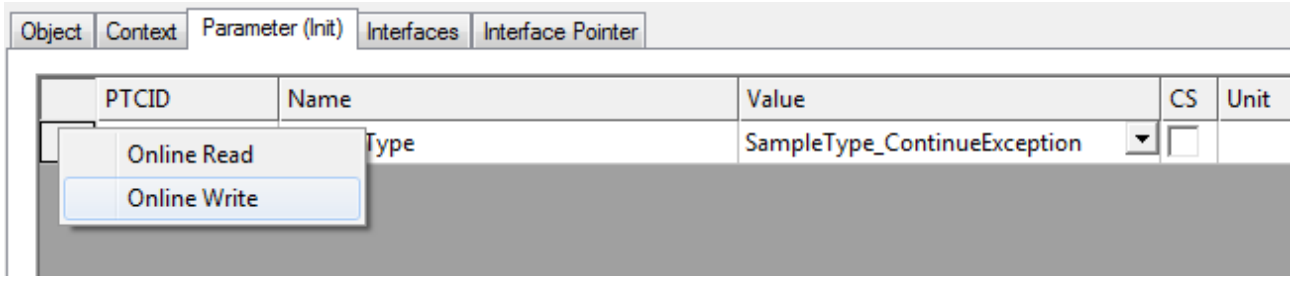
The sample contains five variants that demonstrate the use of SEH in TwinCAT C++:

1. Exception in the case of a NULL-pointer access
2. Exception in the case of a NULL-pointer access with a filter
3. Exception with Finally
4. A customer-specific structured exception
5. Exception with Continue block

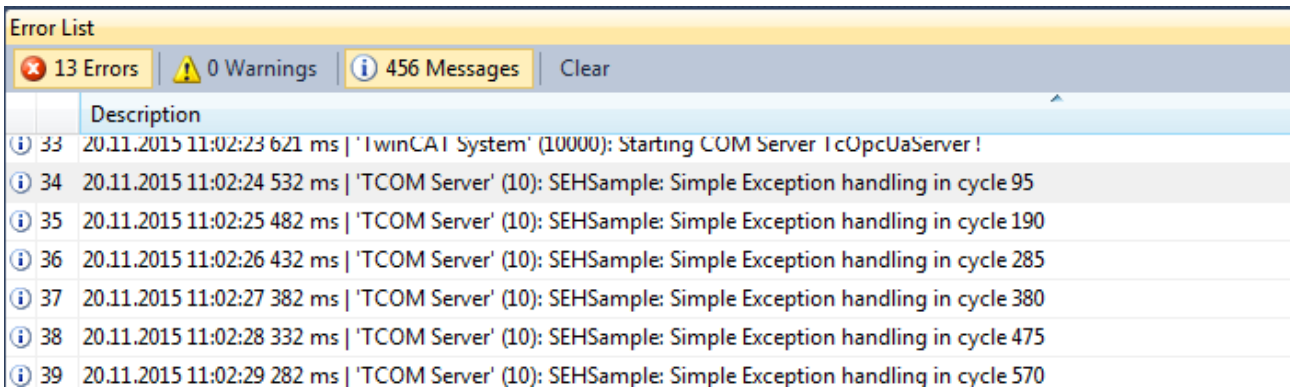
All of these variants can be selected via a drop-down box at the instance of the C++:



After selecting a variant you can also write the value at runtime by right-clicking on the first column:



All variants write trace messages to illustrate the behavior, so that messages appear in TwinCAT Engineering:



Understanding the sample

The selection in the drop-down box is an enumeration that is used in the CycleUpdate() of the module for selecting a case (switch case). As a result the variants can be considered independently of one another here:

1. Exception in the case of a NULL-pointer access
   Here, a PBYTE is created as NULL and used afterwards, which leads to an exception.
   This is intercepted by the TcTry{} block and an output generated.
2. Exception in the case of a NULL-pointer access with a filter
   This variant also accesses a NULL-pointer, but in TcExcept{} it uses a method, FilterException(), that is also defined in the module. Reactions take place to different exceptions within the method; in this case a message is merely output.
3. Exception with Finally
   A NULL-pointer access takes place once again here, but this time a TcFinally{} block is executed in every case.
4. A customer-specific structured exception
   By means of TcRaiseException() an exception is generated that is intercepted and processed by the FilterException() method. Since this is an exception defined in the module, the FilterException() method additionally outputs a further (specific) message.
5. Exception with Continue block
   Here too, a NULL-pointer access takes place once again with TcExcept{}; however, this time the exception is forwarded after handling in the FilterException() method so that the further TcExcept{} also handles the exception.

15.18 Sample24: Semaphores

This article describes the use of semaphores.

Download

Here you can access the source code for this sample:

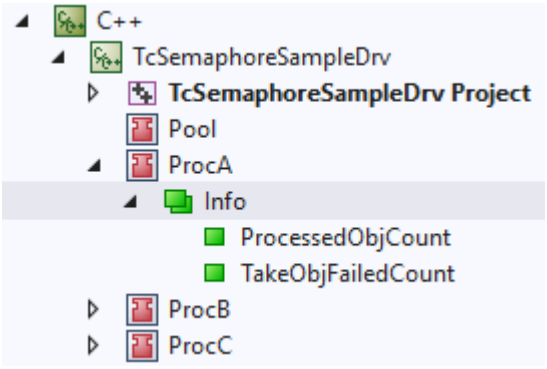
https://github.com/Beckhoff/TC1300_Samples/tree/main/S24-Semaphores

1. Open the zip file that it contains in TwinCAT 3 by clicking on **Open Project …**.
2. Select your target system.
3. Build the sample on your local machine (e.g. **Build->Build Solution**).
4. Note the actions listed on this page under **Configuration**.
5. Activate the configuration by clicking .

⇨ The sample is ready for operation.

Description

The sample contains a "LargeObjPool", which contains two "LargeObj" for processing by the "LargeObjProcessors".



The "LargeObjProcessors" are controlled by different tasks with different priorities that run on different CPU cores. In the info data area they count how often processing could be carried out ("Info->ProcessedObjCount") and how often no object was available for processing within the delay time ("Info->TakeObjFailedCount").

Understanding the sample

The LargeObjPool is initialized in the PS transition by the `InitPool()` method with two LargeObj (defined by the parameter "ObjCount"). The pool essentially offers two methods, which are used by the "LargeObjProcessors":

- `TakeObj()` returns a LargeObj if one is available. Based on a semaphore, the system waits until a timeout occurs to see if an object is made available. The objects themselves are stored in a map. Access to the map is protected by a CriticalSection.
- `ReturnObj()` provides a processed object to the pool for further processing.

In its `CycleUpdate()` LargeObjProcessor uses `TakeObj()` from the pool to obtain a LargeObj for processing. If this is successful, processing is simulated via the help function `ConsumeTime()` before the LargeObj is returned to the pool via `ReturnObj`.

Processing of all LargeObj must be complete before the system can be shut down. This is achieved by the LargeObjPool informing the TwinCAT system via `TcTryLockOpState()` and `TcReleaseOpState()` that an object is being processed. As long as the stored counter is not 0, the TcCOM object "Pool" is reset and only shut down later by the Op->SafeOp transition.

15.19 Sample25: Static Library

This article describes the implementation and use of a module of a static TwinCAT 3 C++ library.

Download

Here you can access the source code for this sample:

https://github.com/Beckhoff/TC1300_Samples/tree/main/S25-StaticLibrary

1. Using a Visual Studio with TwinCAT installed, open the project via **Open Project ....**
2. If you want to run the project on Windows or TwinCAT/BSD® (not required for Beckhoff RT Linux®): Configure signing for this project by right-clicking on the project on the page, selecting **Project -> Properties -> TC Sign**, enabling TwinCAT signing, and configuring your certificate and password if necessary.
   Further information on signing C++ projects can be found in the chapter TwinCAT [▶ 21].
3. Select your target system.
4. Build the sample (e.g. **Build->Build Solution**).
5. Activate the configuration by clicking  .

⇨ The sample is ready for operation.

Description

The sample contains two projects – the DriverUsingStaticLib project uses the static content of the StaticLib project.

StaticLib:
On the one hand, StaticLib offers the function ComputeSomething in the StaticFunction.h/.cpp.
On the other hand an interface ISampleInterface is defined (see TMCEditor) and implemented in the MultiplicationClass.

DriverUsingStaticLib:
In the CycleUpdate method of the ModuleUsingStaticLib, both the class and the function of StaticLib is used.

Understanding the sample

Follow the steps below to create and use a static library.

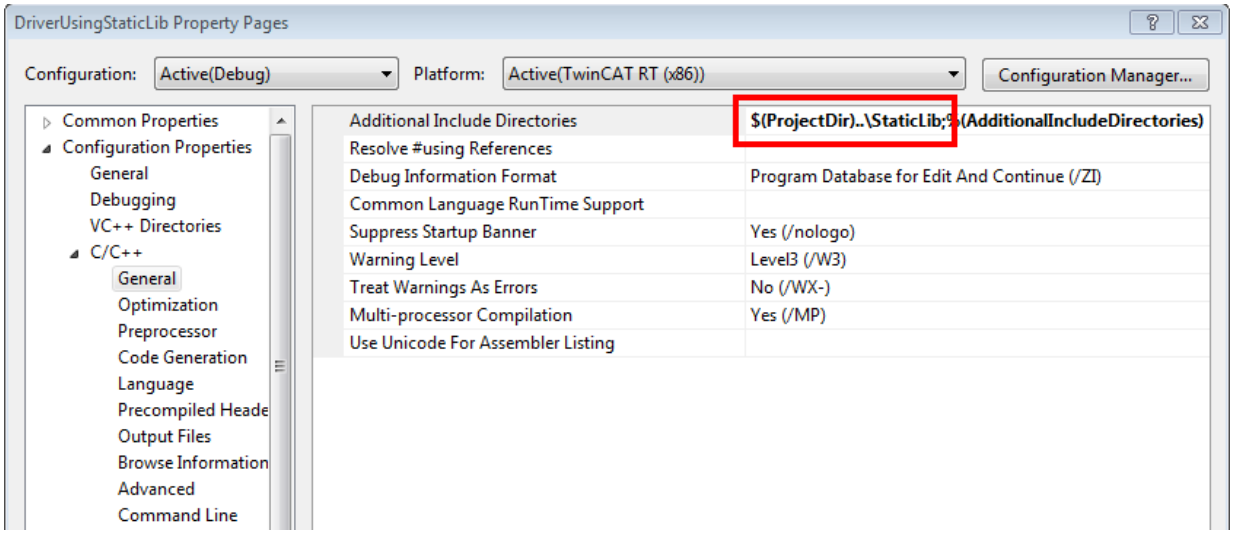
Manual recompilation

Note that Visual Studio does not automatically recompile the static library during driver development. Do that manually.

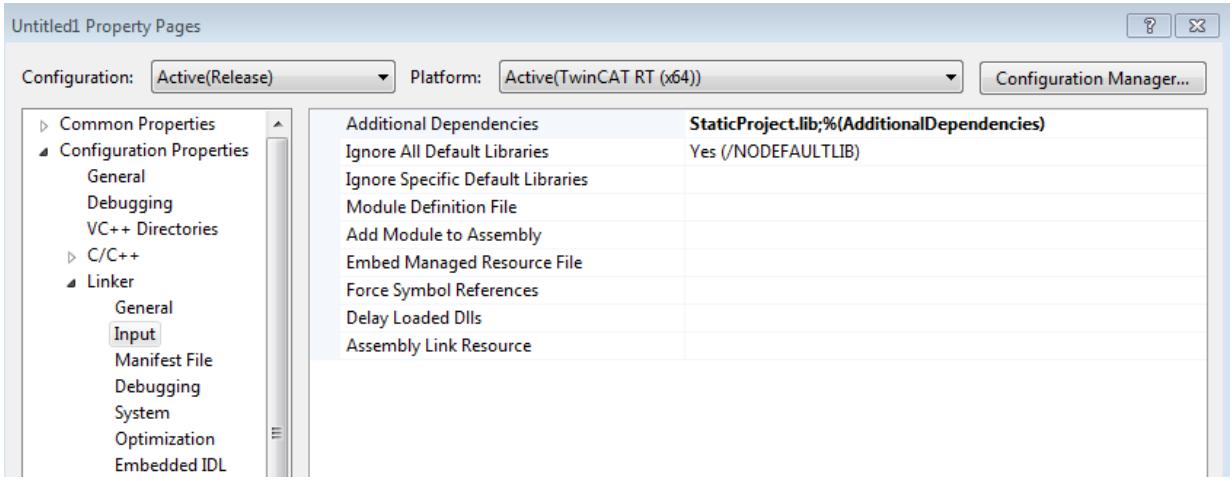
✓ During development of a C++ project use the TwinCAT Static Library Project template for creating a static library.

✓ For the following steps use the **Edit** dialog of VisualStudio, so that afterwards %(AddtitionalIncludeDirectories) or %(AdditionalDependencies) is used.

1. In the driver add the directory of the static library to the compiler under **Additional Include Directories**.



2. Add this as an additional dependency for the linker in the driver, which uses the static library. Open the project properties of the driver and add the static library:



15.20 Sample26: Order of execution in a task

This article describes the determination of the task execution order if more than one module is assigned to a task.

Download

Here you can access the source code for this sample:

https://github.com/Beckhoff/TC1300_Samples/tree/main/S26-SortOrder

1. Using a Visual Studio with TwinCAT installed, open the project via **Open Project ....**
2. If you want to run the project on Windows or TwinCAT/BSD® (not required for Beckhoff RT Linux®): Configure signing for this project by right-clicking on the project on the page, selecting **Project -> Properties -> TC Sign**, enabling TwinCAT signing, and configuring your certificate and password if necessary.
   Further information on signing C++ projects can be found in the chapter TwinCAT [▶ 21].
3. Select your target system.
4. Build the sample (e.g. **Build->Build Solution**).
5. Activate the configuration by clicking .

⇨ The sample is ready for operation.

Description

The sample contains the SortOrder module, which is instanced twice. The sort order determines the execution order, which can be configured via the TwinCAT Module Instance Configurator [▶ 129].

For example, the CycleUpdate method tracks the object name and ID along with the sort order of this module. On the console window you can see the order of execution:

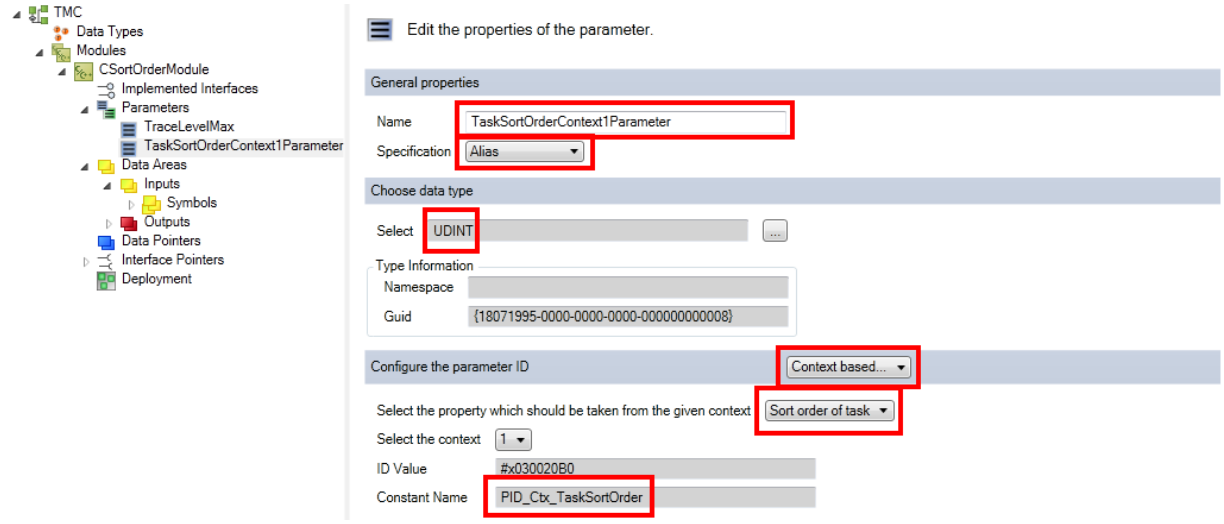
```
32 05.09.2014 13:01:14 343 ms | 'TCOM Server' (10): CSortOrderModule::CycleUpdate() I am 'SortOrder1' (0x01010010) w/ SortOrder 150
33 05.09.2014 13:01:14 343 ms | 'TCOM Server' (10): CSortOrderModule::CycleUpdate() I am 'SortOrder2' (0x01010020) w/ SortOrder 170
34 05.09.2014 13:01:15 343 ms | 'TCOM Server' (10): CSortOrderModule::CycleUpdate() I am 'SortOrder1' (0x01010010) w/ SortOrder 150
35 05.09.2014 13:01:15 343 ms | 'TCOM Server' (10): CSortOrderModule::CycleUpdate() I am 'SortOrder2' (0x01010020) w/ SortOrder 170
36 05.09.2014 13:01:16 343 ms | 'TCOM Server' (10): CSortOrderModule::CycleUpdate() I am 'SortOrder1' (0x01010010) w/ SortOrder 150
37 05.09.2014 13:01:16 343 ms | 'TCOM Server' (10): CSortOrderModule::CycleUpdate() I am 'SortOrder2' (0x01010020) w/ SortOrder 170
```

In the sample, one instance is configured with Sort Order 150 and one with 170, while both instances are assigned to a task.

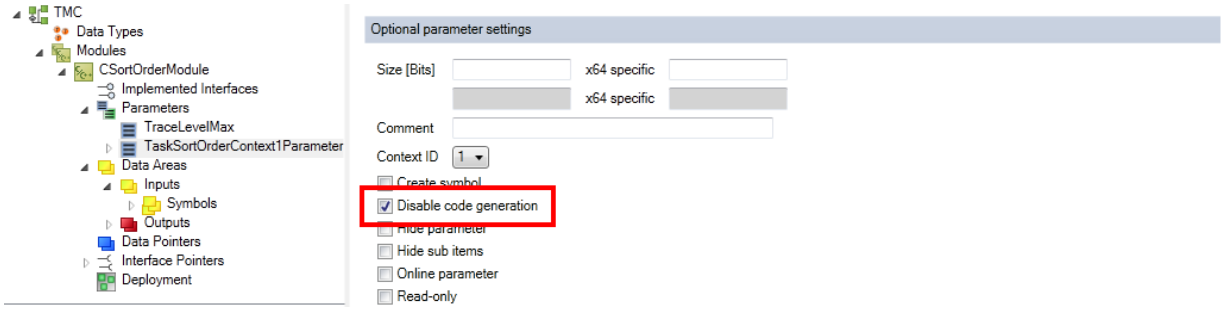
Understanding the sample

✓ A TcCOM C++ module with cyclic IO.

1. The module requires a context-based parameter Sort order of task, which will automatically select PID_Ctx_TaskSortOrder as name.
   Note that the parameter must be an alias (specification) of data type UDINT:



2. Start the TMC Code Generator in order to obtain the standard implementation.
3. Since the code is modified in the next step, disable the code generation for this parameter now.



4. Make sure you accept the changes before restarting the TMC Code Generator:
   Take a look at the CPP module (SortOrderModule.cpp in the sample). The instance of the smart pointer of the cyclic caller includes information data, including a field for the sorting order. The parameter value is stored in this field.

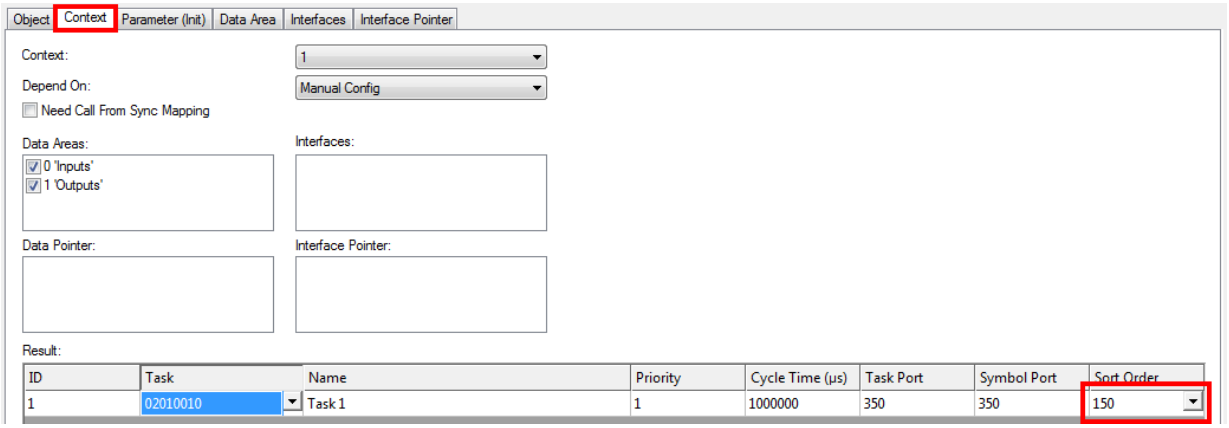
```
////////////////////////////////////////////////////////////////////////////////
// Set parameters of CSortOrderModule
BEGIN_SETOBJPARA_MAP(CSortOrderModule)
    SETOBJPARA_DATAAREA_MAP()
///<AutoGeneratedContent id="SetObjectParameterMap">
    SETOBJPARA_VALUE(PID_TcTraceLevel, m_TraceLevelMax)
    SETOBJPARA_ITFPTR(PID_Ctx_TaskOid, m_spCyclicCaller)
///</AutoGeneratedContent>
    SETOBJPARA_TYPE_CODE(PID_Ctx_TaskSortOrder, ULONG, m_spCyclicCaller.GetInfo()-
>sortOrder=*p) //ADDED
    //generated code: SETOBJPARA_VALUE(PID_Ctx_TaskSortOrder, m_TaskSortOrderContext1Parameter)
END_SETOBJPARA_MAP()

////////////////////////////////////////////////////////////////////////////////
// Get parameters of CSortOrderModule
BEGIN_GETOBJPARA_MAP(CSortOrderModule)
    GETOBJPARA_DATAAREA_MAP()
///<AutoGeneratedContent id="GetObjectParameterMap">
    GETOBJPARA_VALUE(PID_TcTraceLevel, m_TraceLevelMax)
    GETOBJPARA_ITFPTR(PID_Ctx_TaskOid, m_spCyclicCaller)
///</AutoGeneratedContent>
    GETOBJPARA_TYPE_CODE(PID_Ctx_TaskSortOrder, ULONG, *p=m_spCyclicCaller.GetInfo()-
>sortOrder) //ADDED
    //generated code: GETOBJPARA_VALUE(PID_Ctx_TaskSortOrder, m_TaskSortOrderContext1Parameter)
END_GETOBJPARA_MAP()
```

5. In this sample the object name, ID and sort order are tracked cyclically:

```
    // TODO: Add your cyclic code here
     m_counter+=m_Inputs.Value;
     m_Outputs.Value=m_counter;
    m_Trace.Log(tlAlways, FNAMEA "I am '%s' (0x%08x) w/ SortOrder %d ", this->TcGetObjectName(),
this->TcGetObjectId() , m_spCyclicCaller.GetInfo()->sortOrder); //ADDED
```

6. The sorting order can also be transferred as the fourth parameter of the method ITcCyclicCaller::AddModule(), which is used in CModuleA::AddModuleToCaller().
7. Allocate a task with a **long cycle interval** (e.g. 1000 ms) to the instances of this module, in order to limit the tracking messages sent to the TwinCAT Engineering system.
8. Assign a different sorting order to each instance via the TwinCAT Module Instance Configurator [▶ 129]:



15.21 Sample30: Timing Measurement

This article describes how to implement a TwinCAT 3 C++ module which contains time measurement functionalities.

Download

Here you can access the source code for this sample:

https://github.com/Beckhoff/TC1300_Samples/tree/main/S30-Timing

1. Using a Visual Studio with TwinCAT installed, open the project via **Open Project ....**
2. If you want to run the project on Windows or TwinCAT/BSD® (not required for Beckhoff RT Linux®): Configure signing for this project by right-clicking on the project on the page, selecting **Project -> Properties -> TC Sign**, enabling TwinCAT signing, and configuring your certificate and password if necessary.
   Further information on signing C++ projects can be found in the chapter TwinCAT [▶ 21].
3. Select your target system.
4. Build the sample (e.g. **Build->Build Solution**).
5. Activate the configuration by clicking  .

⇨ The sample is ready for operation.

Description

This sample exclusively deals with time measurement such as

- Querying the task cycle time in nanoseconds
- Querying the task priority
- Querying the time when the task cycle starts at intervals of 100 nanoseconds since January 1, 1601 (UTC)
- Querying the distributed clock time when the task cycle starts in nanoseconds since January 1, 2000

- Querying the time when the method is called at intervals of 100 nanoseconds since January 1, 1601 (UTC)

See also

ITcTask interface [▶ 183]

15.22 Sample31: Functionblock TON in TwinCAT3 C++

This article describes the implementation of a behavior in C++, which is comparable with a TON function block of PLC / IEC-61131-3.

Source

Here you can access the source code for this sample:

https://github.com/Beckhoff/TC1300_Samples/tree/main/S31-CTON

1. Using a Visual Studio with TwinCAT installed, open the project via **Open Project ....**
2. If you want to run the project on Windows or TwinCAT/BSD® (not required for Beckhoff RT Linux®): Configure signing for this project by right-clicking on the project on the page, selecting **Project -> Properties -> TC Sign**, enabling TwinCAT signing, and configuring your certificate and password if necessary.
   Further information on signing C++ projects can be found in the chapter TwinCAT [▶ 21].
3. Select your target system.
4. Build the sample (e.g. **Build->Build Solution**).
5. Activate the configuration by clicking  .

⇨ The sample is ready for operation.

Description

The behavior of this module is comparable with a module that was created with the Cyclic IO wizard. m_input.Value is added to m_Output.Value. As opposed to the Cyclic IO module, this module only adds m_input.Value to m_Output.Value if the defined time interval (1000 ms) has elapsed.

This is achieved with the help of a CTON class, which is comparable with the TON function block of PLC / 61131.

Understanding the sample

The C++ class CTON (TON.h/.cpp) provides the behavior of a TON function block of PLC / 61131. The Update() method is comparable with the rump of the function block, which has to be called regularly.

The Update() method contains two "in" parameters:

- IN1: Starts the timer switch on a rising edge and resets the timer switch on a falling edge.
- PT: Describes the time to wait before Q is set.

And two "out" parameters:

- Q: TRUE if PT exhibited a rising edge seconds after IN.
- ET: Designates the elapsed time.

Beyond that, ITcTask must be provided to query the time base.

See also

Sample30: Timing Measurement [▶ 300]

ITcTask interface [▶ 183]

15.23 Sample35: Access Ethernet

This article describes the implementation of TwinCAT 3 C++ modules that communicate directly via an Ethernet card. The sample code queries a hardware address (MAC) from a communication partner by means of the cyclic transmission and reception of ARP packets.

The sample illustrates the direct access to the Ethernet card. The TF6311 TCP/UDP RT function provides access to Ethernet cards on the basis of TCP and UDP, so that an implementation of a network stack is not necessary on the basis of this sample.

Download

Here you can access the source code for this sample:

https://github.com/Beckhoff/TC1300_Samples/tree/main/S35-AccessEthernet

1. Open the zip file that it contains in TwinCAT 3 by clicking on **Open Project …**.
2. Select your target system.
3. Build the sample on your local machine (e.g. **Build->Build Solution**).
4. Note the actions listed on this page under **Configuration**.
5. Activate the configuration by clicking  .

⇨ The sample is ready for operation.

Description

The sample contains an instance of the TcEthernetSample module, which sends and received ARP packets for the purpose of determining the remote hardware address (MAC).

The CycleUpdate method implements a rudimentary state machine for sending ARP packets and waiting for a response with a timeout.

The sample uses two Ethernet components from TwinCAT:

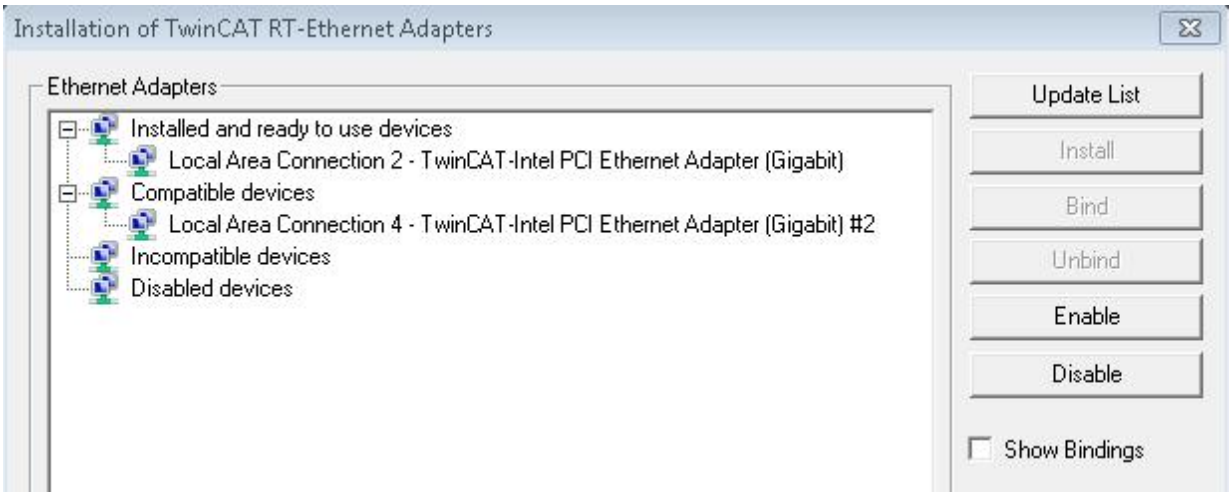
1. An **ITcEthernetAdapter** (instance name in the sample is m_spEthernetAdapter) represents an RT Ethernet adapter. It enables access to the adapter parameters such as hardware MAC address, link speed and link errors. It can be used to send Ethernet frames and enables a module instance to register itself as an ItcIoEthProtocol via the registerProtocol method.
2. The **ITcIOoEthProtocol** is extended by the sampling module, which ensures that a notification takes place via the **ITcEthernetAdapter** in case of Ethernet events.

Configuration

The downloaded TwinCAT project must be configured for execution in a network environment. Please carry out the following steps:

✓ This sample demands the use of the TwinCAT driver by the Ethernet card.

1. Start TcRteInstall.exe either from the XAE via the menu **TwinCAT->Show Realtime Ethernet compatible devices…** or from the hard disk on the XAR systems.



2. You may have to install and activate the driver with the help of the buttons.
3. TwinCAT must know which Ethernet card is to be used. Open the project in **XAE** and click on **Select I/O / Devices / Device 1 (RT-Ethernet Adapter)**.
4. Click on the **Adapter** tab and select the adapter with **Search**.
5. TcEthernetSample_Obj1 must be configured. Open the instance window and set the following values:
   Parameter (Init): SenderIpAddress (IP of the network adapter configured in step 2)
   Parameter (Init): TargetIpAddress (IP of target host)
   Interface pointer: EthernetAdapter must point to I/O / Devices / Device 1 (RT-Ethernet Adapter).

15.24 Sample37: Archive data

The sample TcCOM Object Archive describes restoration and saving of an object state during initialization and deinitialization.

TwinCAT supports retain data

TwinCAT also supports retain data, in order to utilize the NOVRAM of a device to make data persistent.

Download

Here you can access the source code for this sample:

https://github.com/Beckhoff/TC1300_Samples/tree/main/S37-ArchiveData

1. Using a Visual Studio with TwinCAT installed, open the project via **Open Project ....**
2. If you want to run the project on Windows or TwinCAT/BSD® (not required for Beckhoff RT Linux®): Configure signing for this project by right-clicking on the project on the page, selecting **Project -> Properties -> TC Sign**, enabling TwinCAT signing, and configuring your certificate and password if necessary.
   Further information on signing C++ projects can be found in the chapter TwinCAT [▶ 21].
3. Select your target system.
4. Build the sample (e.g. **Build->Build Solution**).
5. Activate the configuration by clicking  .

⇨ The sample is ready for operation.

Description

The sample TcCOM object archive describes restoration and saving of an object state during initialization and deinitialization. The state of the sample class CmoduleArchive corresponds to the value of the counter CModuleArchive::m_counter.

During the transition from PREOP to SAFEOP, i.e. when calling the method CModuleArchive::SetObjStatePS(), the object archive server (ITComObjArchiveServer) is used to create an object archive for reading, which is accessed via the interface ITComArchiveOp. This interface provides overloads from operator>>() in order to read them in the archive.

During the transition from SAFEOP to PREOP, i.e. when calling the method CModuleArchive::SetObjStateSP(), the TCOM object archive server is used to create an object archive for writing, which is accessed via the interface ITComArchiveOp. This interface provides overloads from operator<<() in order to write them in the archive.

The interface used here was not developed for the real-time context [▶ 44], therefore the interface can only be used in the non-real-time context.

15.25 TcCOM samples

Modules can communicate between PLC and C++. The description therefore covers handling of C++ modules on the PLC side and handling of the PLC on the C++ side.
The TcCOM samples for communication with the PLC are shown here.

The TcCOM_Sample01 sample [▶ 304] shows how TcCOM communication can take place between two PLCs. In the process functionalities from one PLC are directly called up from the other PLC.

The TcCOM_Sample02 sample [▶ 314] shows how a PLC application can use functionalities of an existing instance of a TwinCAT C++ class. Algorithms written in C++ (or MATLAB®) can thus be easily used in the PLC.
Although in the event of the use of an existing TwinCAT C++ module the TwinCAT C++ license is required on the target system, a C++ development environment is not necessary on the target system or on the development computer.

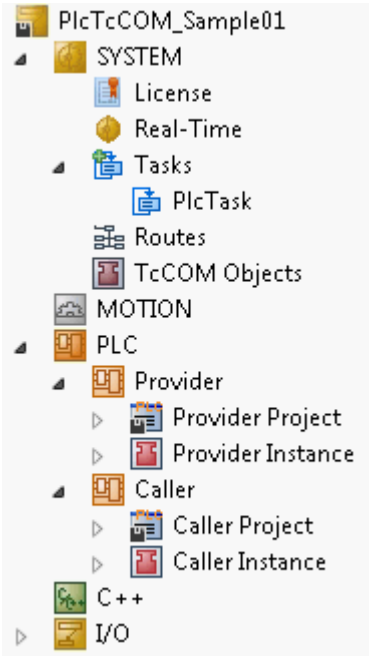
The TcCOM_Sample03 sample [▶ 318] shows how a PLC application uses functionalities of a TwinCAT C++ class by generating an instance of C++ class at the same time. In comparison to the previous sample this can offer increased flexibility.

15.25.1 TcCOM_Sample01_PlcToPlc

This sample describes a TcCOM communication between two PLCs.

Functionalities provided by a function block in the first PLC (also called "provider" in the sample), are called from the second PLC (also called "caller" in the sample). To this end it is not necessary for the function block or its program code to be copied. Instead the program works directly with the object instance in the first PLC.

Both PLCs must be in a TwinCAT runtime. In this connection a function block offers its methods system-wide via a globally defined interface and represents itself a TcCOM object. As is the case with every TcCOM object, such a function block is also listed at runtime in the **TcCOM Objects** node.



The procedure is explained in the following sub-chapters:

- Creating an FB in the first PLC that provides its functionality globally [▶ 305]
- Creating an FB in the second PLC that, as a simple proxy, also offers this functionality there [▶ 309]
- Execution of the sample project [▶ 312]

Downloading the sample: https://infosys.beckhoff.com/content/1033/TC3_C/Resources/2343046667.zip

Race Conditions in the case of Multi-Tasking (Multi-Threading) use

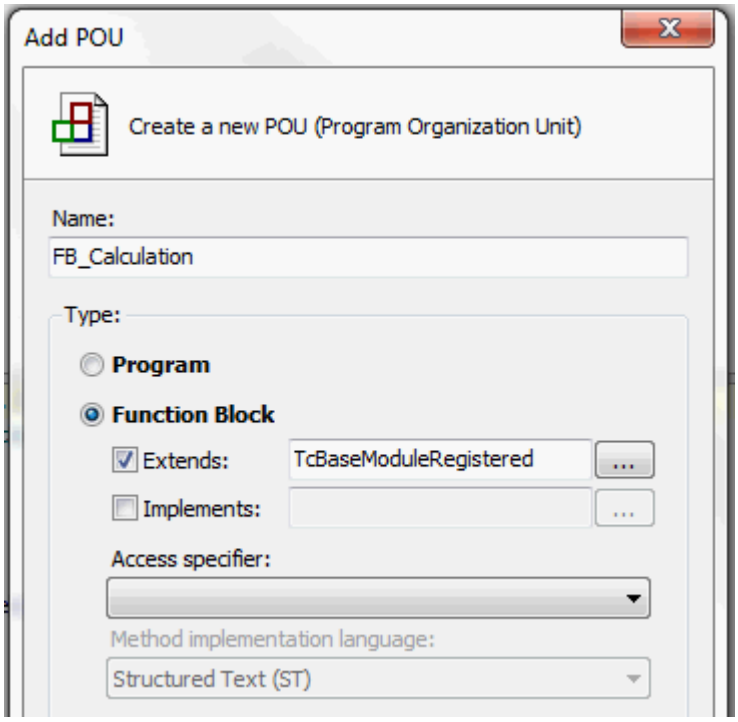
The function block that provides its functionality globally is instantiated in the first PLC. It can be used there like any function block. In addition, if it is used from a different PLC (or, for example, from a C++ module), make sure that the methods offered are thread-safe, as the various calls could take place simultaneously from different task contexts or mutually interrupt one another, depending on the system configuration. In this case the methods must not access member variables of the function block or global variables of the first PLC. If this should be absolutely necessary, prevent simultaneous access. Observe the function TestAndSet() from the Tc2_System library.

System requirements

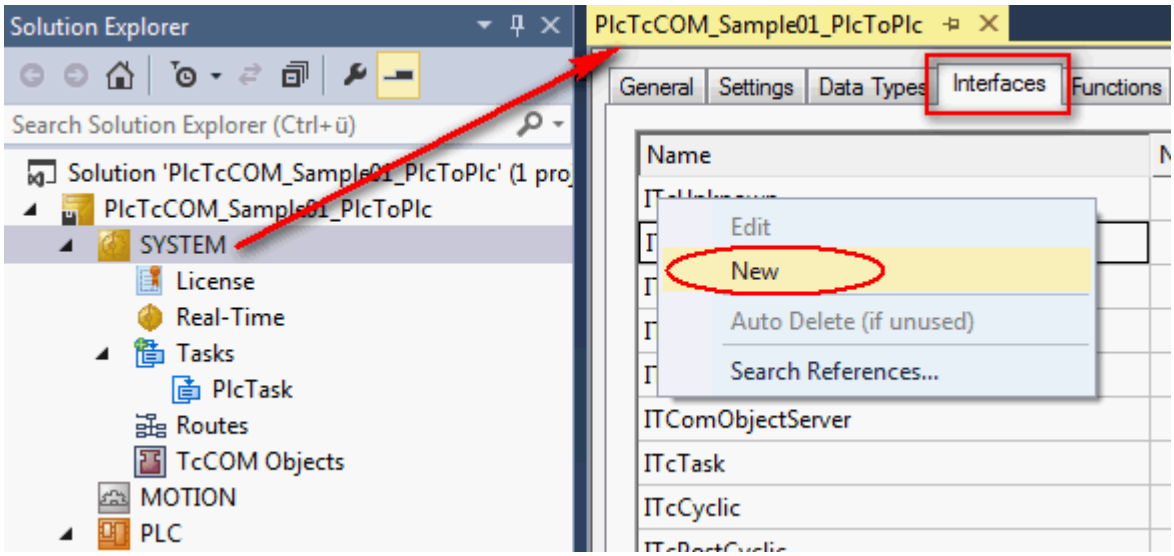
TwinCAT version	Hardware	Libraries to be integrated
TwinCAT 3.1, Build 4020	x86, x64, ARM	Tc3_Module

15.25.1.1 Creating an FB in the first PLC which provides its functionality globally

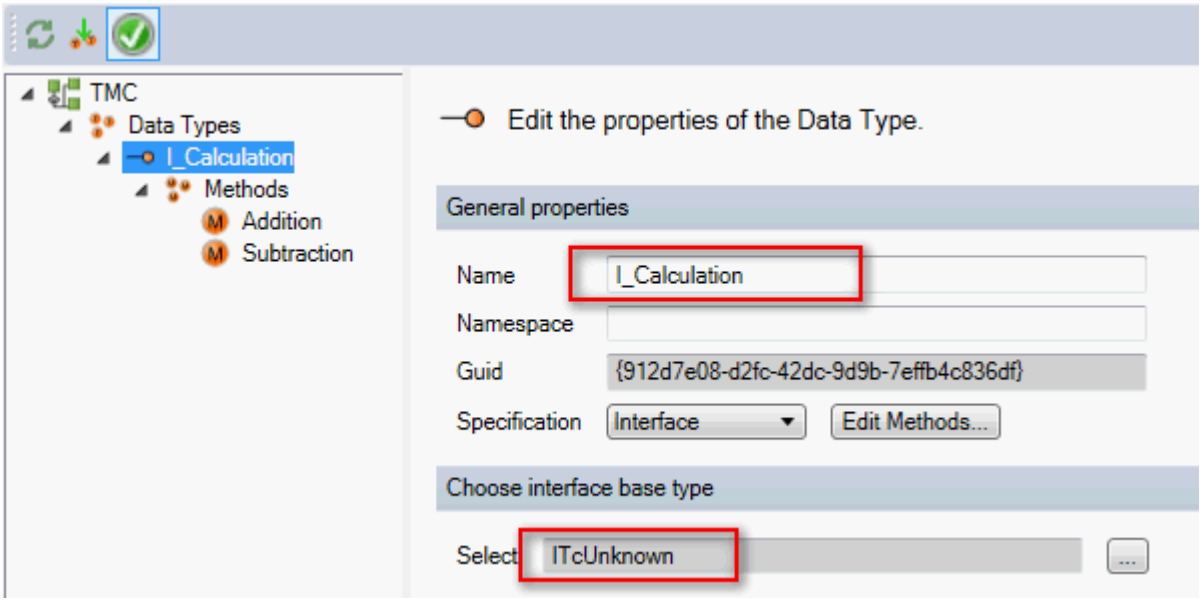
1. Create a PLC and a new function block (FB) (in this case: FB_Calculation). Derive the function block from the TcBaseModuleRegistered class so that an instance of the function block is available not only in the same PLC but also in a second one. As an alternative, you can modify an FB in an existing PLC.



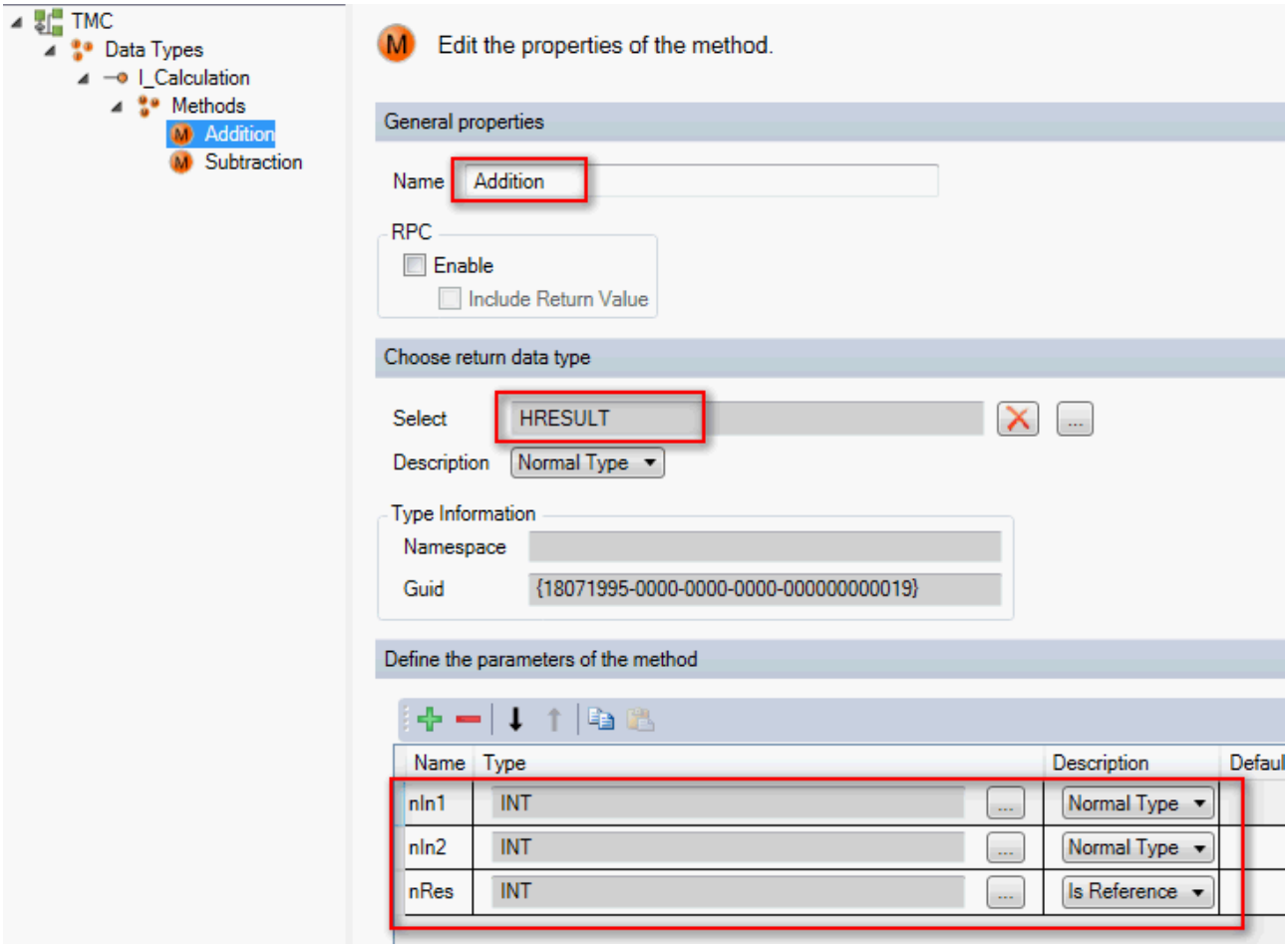
2. The function block must provide its functionality through methods. These are defined in a global interface whose type is known system-wide and regardless of the programming language. To create a global interface, open the context menu in the “Interface” tab of System Properties and select “New”.
   ⇨ The TMC Editor opens to help create a global interface.



3. Enter the name (in this case: I_Calculation) and append the desired methods. The interface is automatically derived from ITcUnknown to comply with the TwinCAT TcCOM module concept.



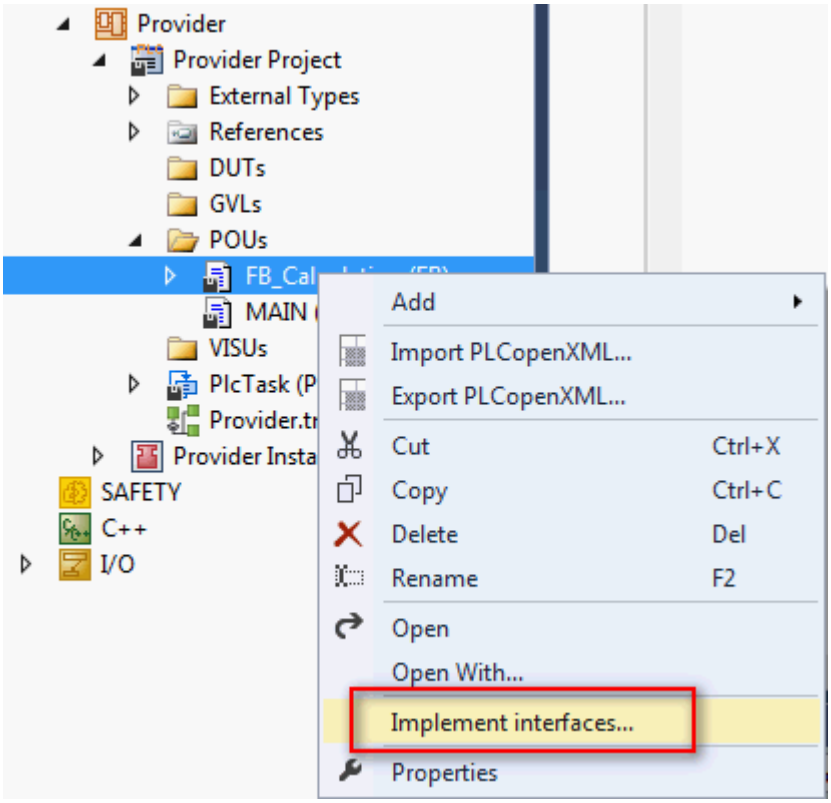
4. Enter the names of the methods (in this case: Addition() and Subtraction()) and select HRESULT as return data type. This return type is mandatory if this type of TcCOM communication is to be implemented.
5. Finally, specify the method parameters and close the TMC Editor.



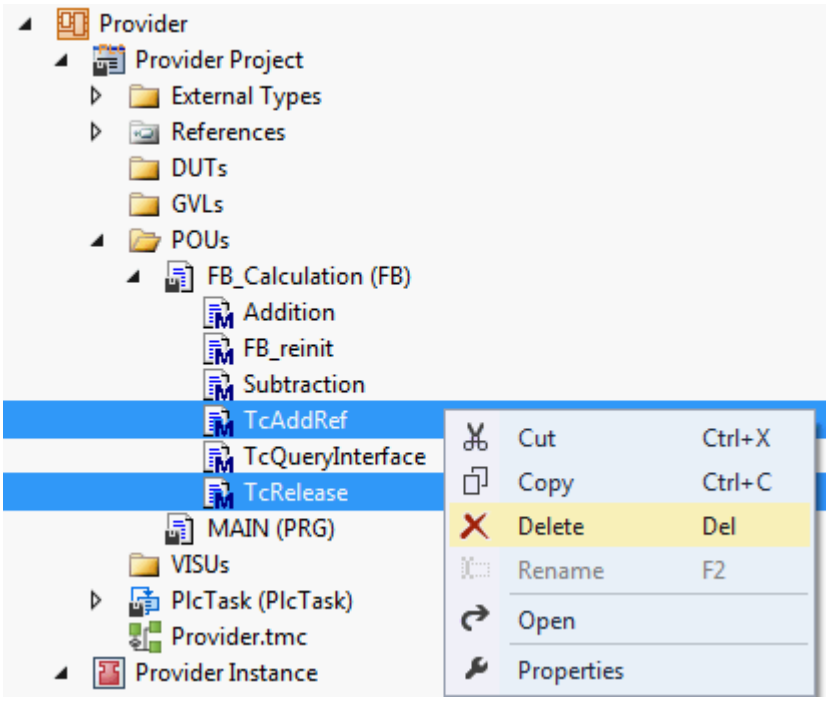
6. Now implement the I_Calculation interface in the FB_Calculation function block and append the “c++_compatible” attribute.

```
4   {attribute 'c++_compatible'}
5   FUNCTION_BLOCK FB_Calculation EXTENDS TcBaseModuleRegistered IMPLEMENTS I_Calculation
6
7   VAR
8   END_VAR
9
```

7. Choose the “Implement interfaces...” option in the context menu of the function block in order to obtain the methods belonging to this interface.



8. Delete the two methods TcAddRef() and TcRelease() because the existing implementation of the base class is to be used.



9. Create the FB_reinit() method for the FB_Calculation function block and call the basic implementation. This ensures that the FB_reinit() method of the base class will run during the online change. This is imperative.

```
FB_Calculation.FB_reinit
1  METHOD FB_reinit : BOOL
2  VAR_INPUT
3  END_VAR
4

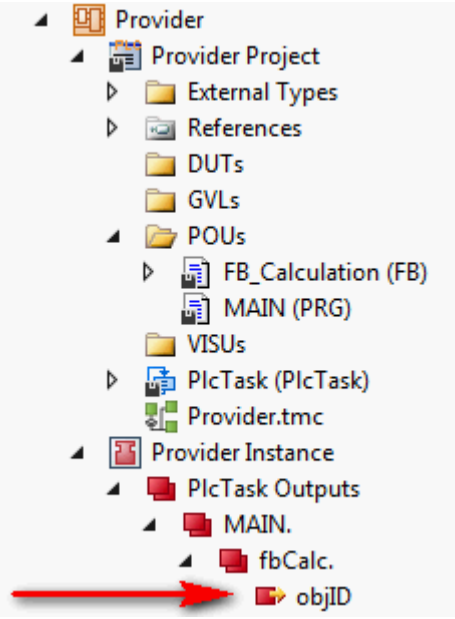
1  SUPER^.FB_reinit();
2
```

10. Implement the TcQueryInterface() method of the Interface ITcUnknown [▶ 187]. This method enables other TwinCAT components to obtain an interface pointer to an instance of this function block, allowing them to perform method calls. Calling TcQueryInterface is successful if the function block or its base class provides the interface requested by means of iid (Interface ID). In this case, the interface pointer that was handed over is allocated an address to the function block with its type changed and the reference counter is also incremented using TcAddRef().
11. Add the corresponding code to the Addition() and Subtraction() methods to provide the functionality: `nRes := nIn1 + nIn2` and `nRes := nIn1 - nIn2`
12. Add one or more instances of this function block to the MAIN program block or to a global variable list.

    ⇨ The implementation in the first PLC is complete.

```
MAIN*
1  PROGRAM MAIN
2  VAR
3      m : UDINT;
4
5      fbCalc : FB_Calculation('MAIN.fbCalc');
6  END_VAR
7
```

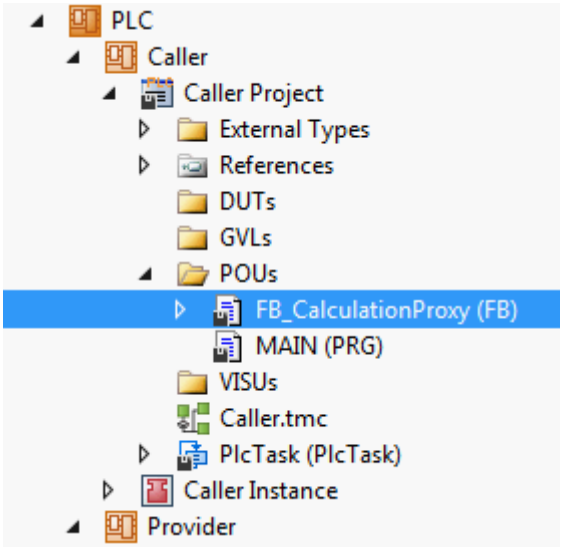
⇨ After compiling the PLC, the object ID of the TcCOM object representing the FB_Calculation instance is available as an outlet in the process image.



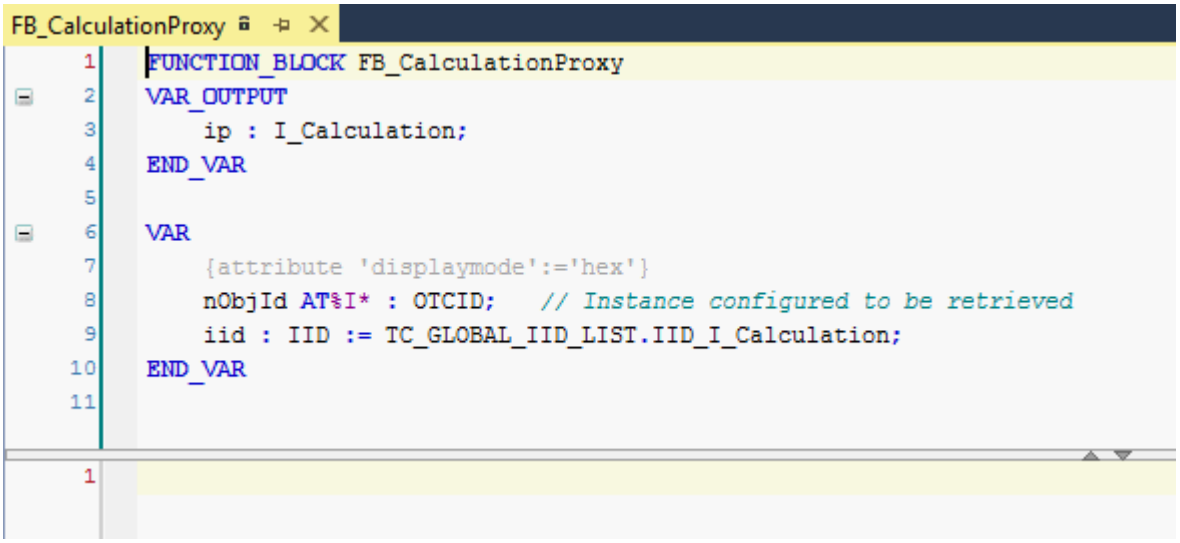
15.25.1.2 Creating an FB which likewise offers this functionality there as a simple proxy in the second PLC,

1. Create a PLC and append a new function block there.

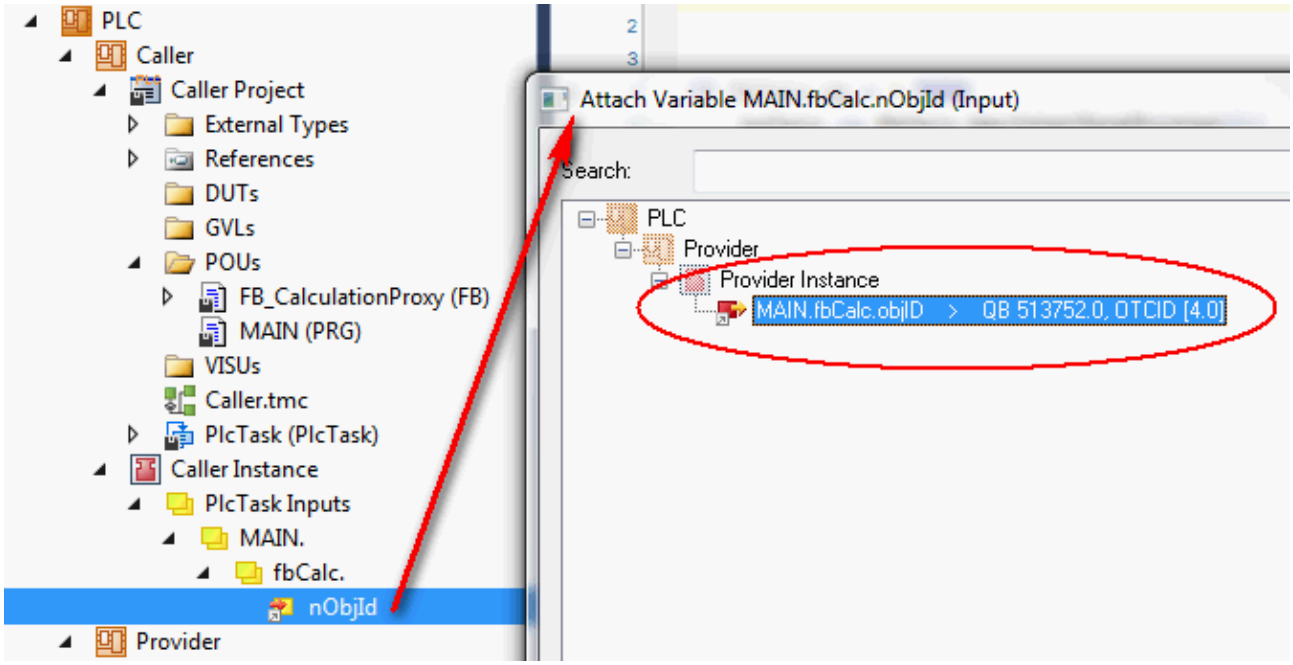
⇨ This proxy function block should provide the functionality which was programmed in the first PLC. It does this via an interface pointer of the type of the global interface I_Calculation.



2. In the declaration part of the function block declare as an output an interface pointer to the global interface which later provides the functionality outward.



3. In addition create the object ID and the interface ID as local member variables.
While the interface ID is already available via a global list, the object ID is assigned via a link in the process image.



4. Implement the PLC proxy function block. First add the GetInterfacePointer() method to the function block. The interface pointer is fetched to the specified interface of the specified TcCOM object with the help of the FW_ObjMgr_GetObjectInstance() function. This will only be executed if the object ID is valid and the interface pointer has not already been allocated. The object itself increments a reference counter.

FB_CalculationProxy.GetInterfacePointer

```
METHOD GetInterfacePointer : HRESULT
VAR
END_VAR
```

```
IF nObjID <> 0 THEN
    IF (ip = 0) THEN // only get interface pointer if it is not already existing
        GetInterfacePointer := FW_ObjMgr_GetObjectInstance(oid:=nObjID, iid:=iid, pipUnk:=ADR(ip));
    ELSE
        GetInterfacePointer := E_HRESULTAdsErr.EXISTS;
    END_IF
ELSE
    GetInterfacePointer := E_HRESULTAdsErr.INVALIDOBJID;
END_IF
```

5. It is imperative to release the used reference again. To this end call the FW_SafeRelease() function in the FB_exit destructor of the function block.

FB_CalculationProxy.FB_exit | FB_CalculationProxy.GetInterfacePointer

```
{attribute 'hide'}
METHOD FB_exit : BOOL
VAR_INPUT
    bInCopyCode : BOOL; // if TRUE, the exit method is c
END_VAR
```

```
IF NOT bInCopyCode THEN // if not online change
    FW_SafeRelease(ADR(ip));
END_IF
```

⇨ This completes the implementation of the Proxy function block.

6. Instantiate the Proxy function block FB_CalculationProxy in the application and call its method GetInterfacePointer() to get a valid interface pointer.
An instance of the proxy block is declared in the application to call the methods provided via the

interface. The calls themselves take all place over the interface pointer defined as output of the function block. As is typical for pointers a prior null check must be made. Then the methods can be called directly, also via Intellisense.

MAIN*

```
1  PROGRAM MAIN
2  VAR
3      fbCalc : FB_CalculationProxy;
4      hrCalc : HRESULT;
5      a : INT := 10;
6      b : INT := 7;
7      nSum : INT; // a + b
8      nDiff : INT; // a - b
9  END_VAR
10
```

```
1  IF fbCalc.ip = 0 THEN
2      hrCalc := fbCalc.GetInterfacePointer();
3  END_IF
4  IF fbCalc.ip <> 0 THEN
5      hrCalc := fbCalc.ip.Addition(a,b,nSum);
6      hrCalc := fbCalc.ip.Subtraction(a,b,nDiff);
7  END_IF
8
```

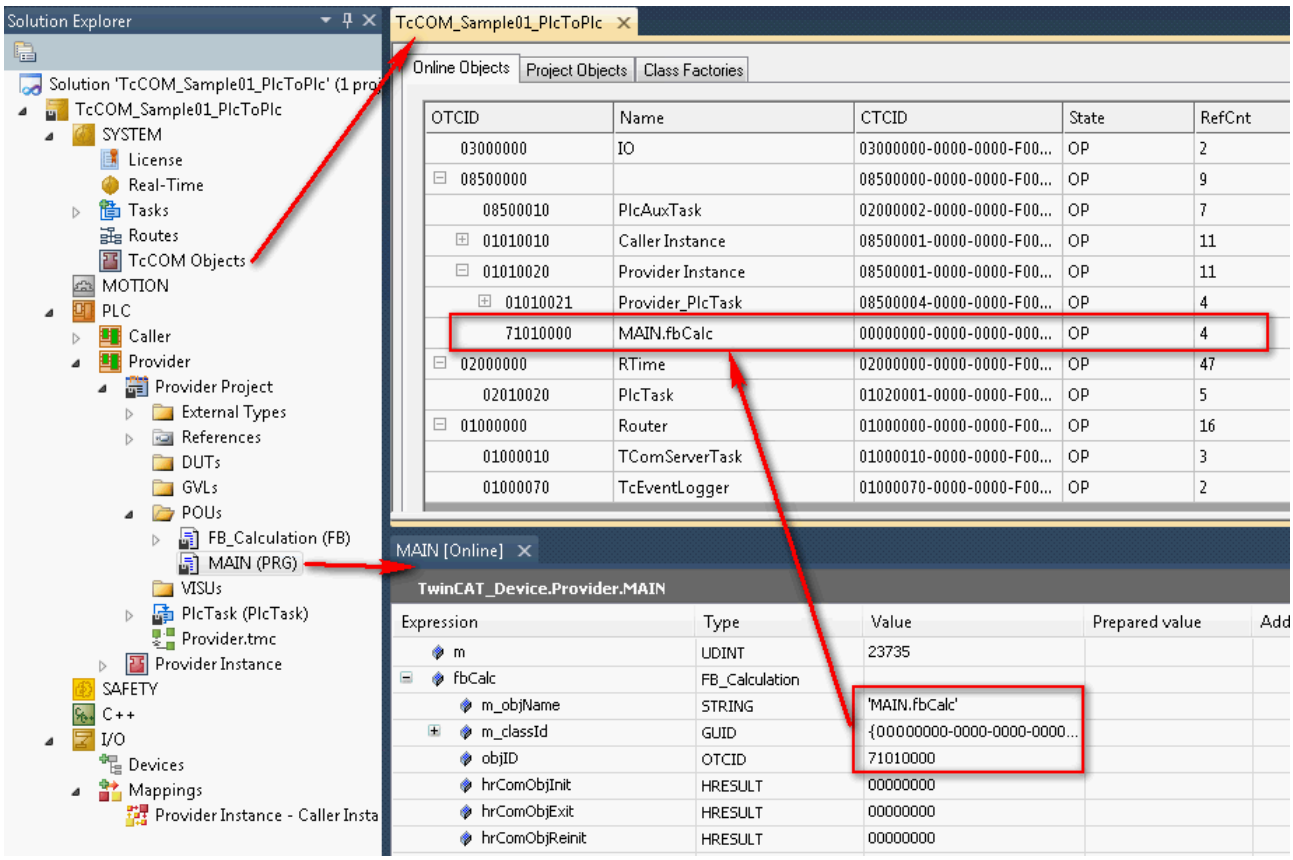
⇨ The sample is ready for testing.

Order irrelevant

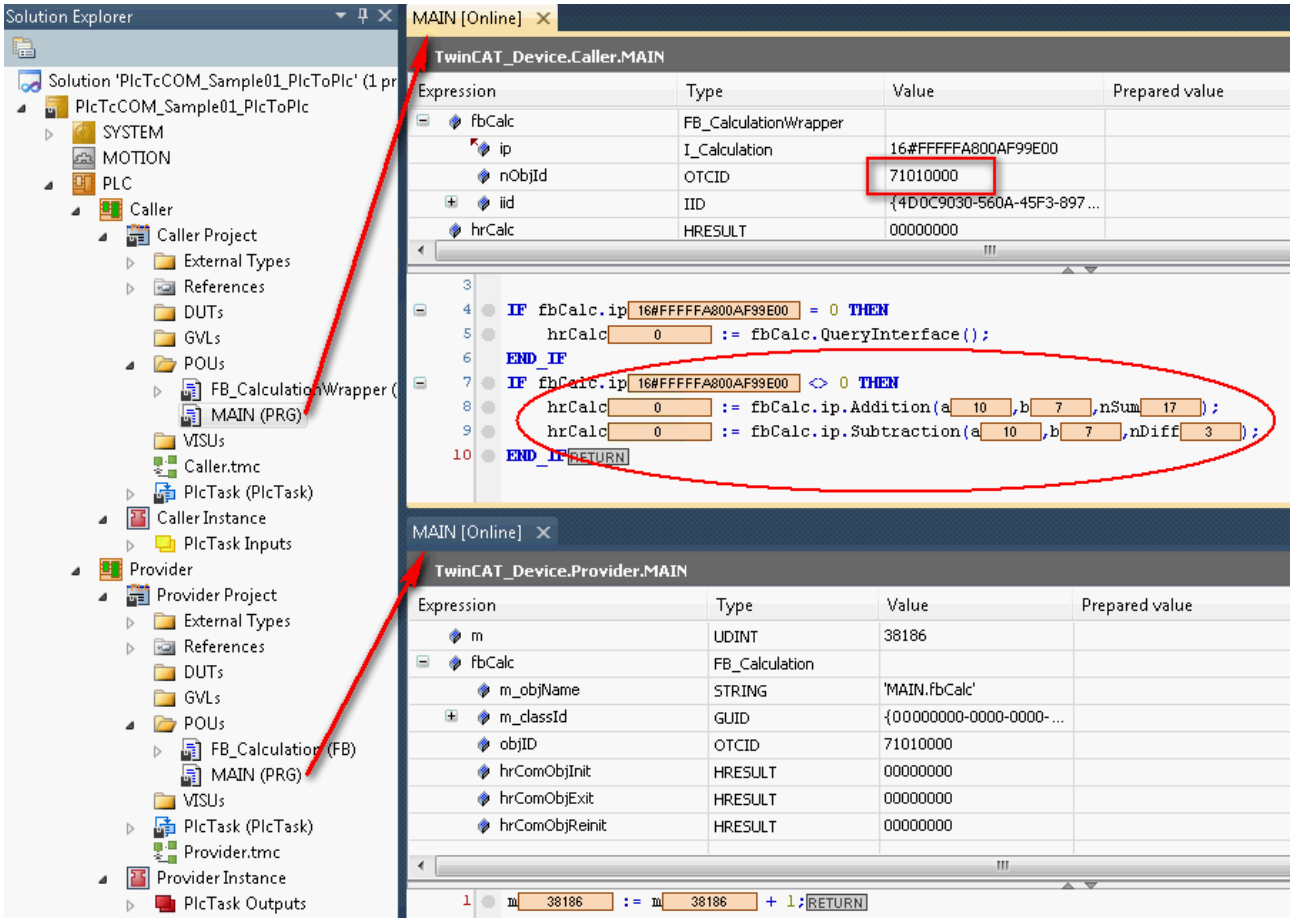
The sequence in which the two PLCs start later is irrelevant in this implementation.

15.25.1.3 Execution of the sample project

1. Select the destination system and compile the project.
2. Enable the TwinCAT configuration and execute a log-in and start both PLCs.
   ⇨ In the online view of the PLC application “Provider” the generated object ID of the C++ object can be seen in the PLC function block FB_Calculation. The project node “TcCOM Objects” keeps the generated object with its object ID and the selected name in its list.



⇨ In the online view of the PLC application “Caller” the Proxy function block has been allocated the same object ID via the process image. The interface pointer has a valid value and the methods are executed.



15.25.2 TcCOM_Sample02_PlcToCpp

This sample describes a TcCOM communication between PLC and C++. In this connection a PLC application uses functionalities of an existing instance of a TwinCAT C++ class. In this way own algorithms written in C++ can be used easily in the PLC.
Although in the event of the use of an existing TwinCAT C++ module the TwinCAT C++ license is required on the target system, a C++ development environment is not necessary on the target system or on the development computer.

An already built C++ module provides one or more classes whose interfaces are deposited in the TMC description file and thus are known in the PLC.

The procedure is explained in the following sub-chapters:

1. Instantiating a TwinCAT C++ class as a TwinCAT TcCOM Object [▶ 314]

2. Creating an FB in the PLC, which as a simple wrapper offers the functionality of the C++ object [▶ 315]

3. Execution of the sample project [▶ 317]

Download the sample: https://infosys.beckhoff.com/content/1033/TC3_C/Resources/2343048971.zip

System requirements

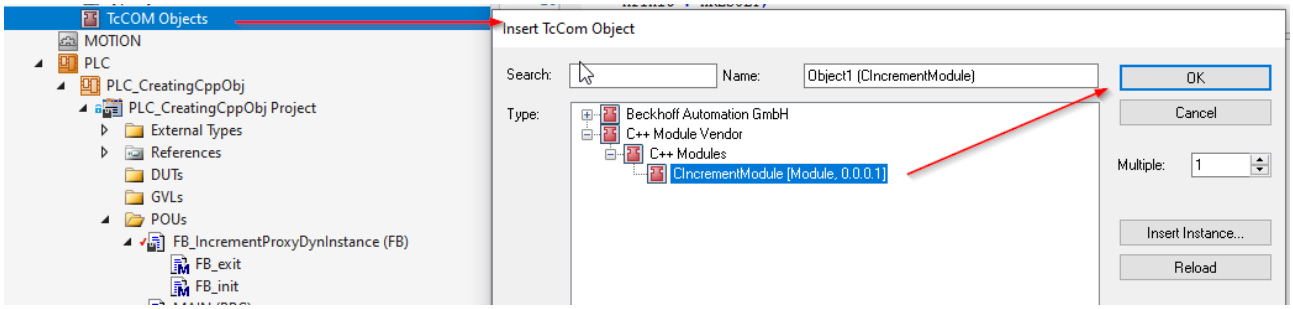
TwinCAT version	Hardware	Libraries to be integrated
TwinCAT 3.1, Build 4020	x86, x64	Tc3_Module

15.25.2.1 Instantiating a TwinCAT++ class as a TwinCAT TcCOM Object

The binary TwinCAT C++ project must be available on the engineering system so that it can be transferred to the target system together with the PLC project during activation.

TwinCAT offers a deployment mechanism for the distribution between engineering systems, which is described at Versioned C++ projects [▶ 49].
(This sample includes in the download the corresponding TMX, because TwinCAT places them automatically in the archive, if the Class Factory is used.)

1. Open a TwinCAT project or create a new project.
2. Add an instance of Class CIncrementModule in the solution under the **TcCOM Objects** node.

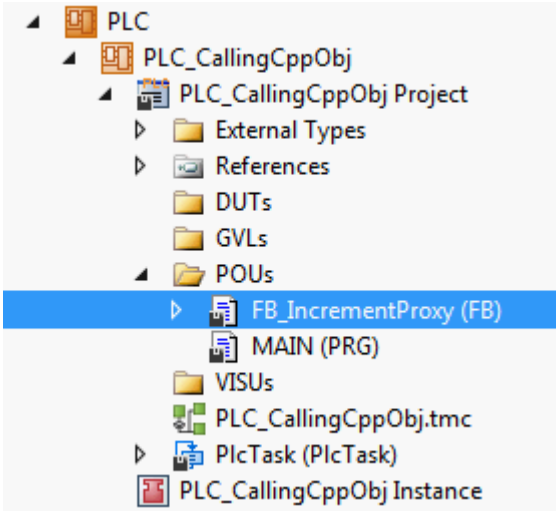


Creation of the C++ modules

In the documentation for TwinCAT C++ [▶ 10] there is a detailed explanation on how C++ modules for TwinCAT are created.

15.25.2.2 Creating an FB in the PLC that, as a simple proxy, offers the functionality of the C++ object

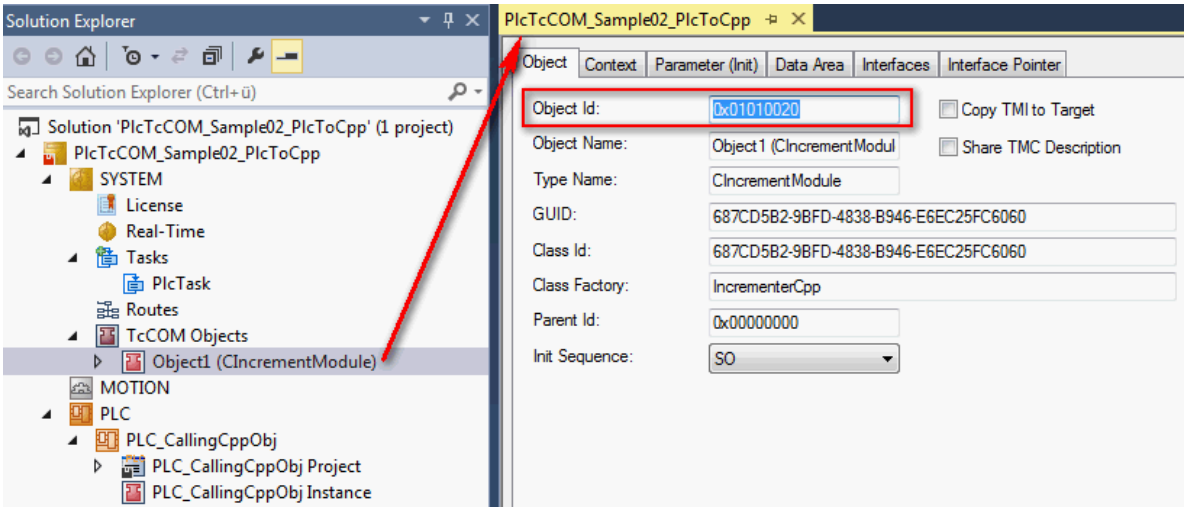
1. Create a PLC and append a new function block there.
   This Proxy function block should provide the functionality that was programmed in C++. It is able to do this via an interface pointer that was defined from the C++ class and is known in the PLC due to the TMC description file.



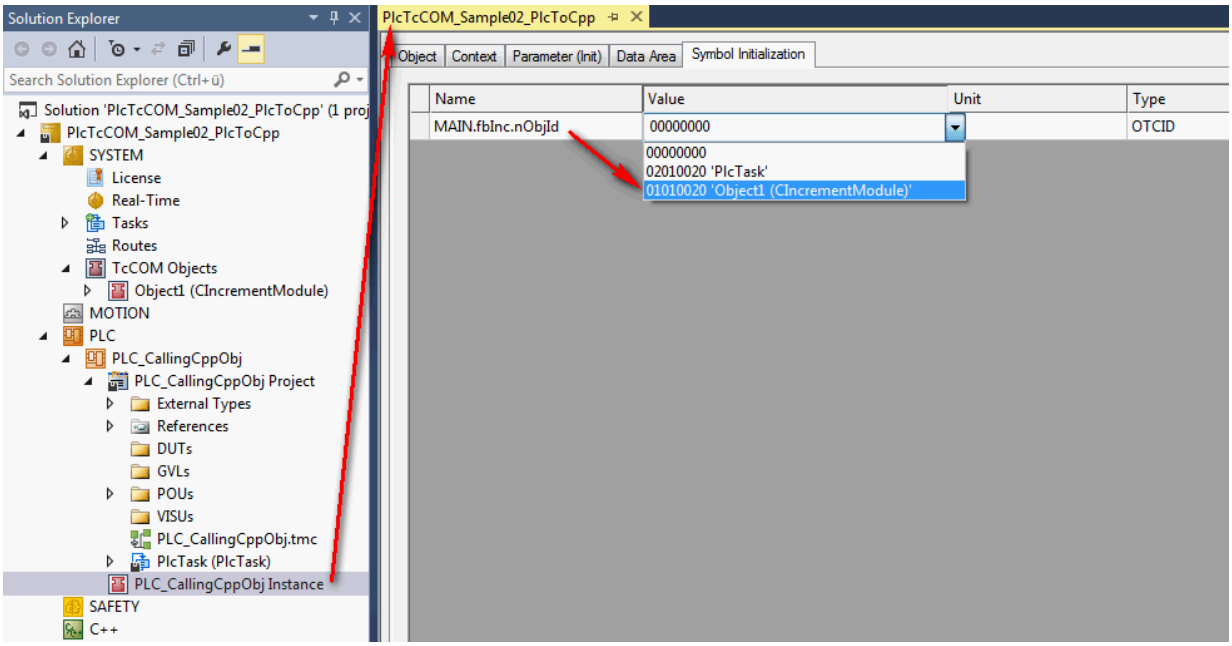
2. In the declaration part of the function block declare as an output an interface pointer to the interface which later provides the functionality outward.
3. Create the object ID and the interface ID as local member variables.
   While the interface ID is already available via a global list, the object ID is allocated via the TwinCAT symbol initialization. The TcInitSymbol attribute ensures that the variable appears in a list for external symbol initialization. The object ID of the created C++ object should be allocated.

```
FB_IncrementProxy
1  FUNCTION_BLOCK FB_IncrementProxy
2  VAR_OUTPUT
3      ip : IIncrement;
4  END_VAR
5
6  VAR
7      {attribute 'TcInitSymbol'}
8      {attribute 'displaymode':='hex'}
9      nObjId : OTCID;     // Instance configured to be retrieved
10     iid : IID := TC_GLOBAL_IID_LIST.IID_IIncrement;
11     hrInit : HRESULT;
12 END_VAR
13
1
```

⇨ The object ID is displayed upon selection of the object under the **TcCOM Objects** node. Provided the TcInitSymbol attribute was used, the list of symbol initializations is located in the node of the PLC instance in the **Symbol Initialization** tab.



4. Here, assign an existing object ID to the symbol name of the variable by drop-down. This value is assigned when the PLC is downloaded so it can be defined prior to the PLC run-time. New symbol initializations or changes are accordingly entered with a new download of the PLC.



As an alternative, the passing of the object ID could also be implemented by means of process image linking as implemented in the first sample (TcCOM_Sample01_PlcToPlc [▶ 304]).

5. Implement the PLC Proxy function block.
First the FB_init constructor method is added to the function block. For the case that it is no longer an OnlineChange but rather the initialization of the function block, the interface pointer to the specified

interface of the specified TcCOM object is obtained with the help of the function FW_ObjMgr_GetObjectInstance(). In this connection the object itself increments a reference counter.

```
FB_IncrementProxy.FB_init
{attribute 'hide'}
METHOD FB_init : BOOL
VAR_INPUT
    bInitRetains : BOOL; // if TRUE, the retain variables are initialized (warm start / cold start)
    bInCopyCode : BOOL;  // if TRUE, the instance afterwards gets moved into the copy code (online
END_VAR

IF NOT bInCopyCode THEN // if not online change
    IF nObjID <> 0 THEN
        hrInit := FW_ObjMgr_GetObjectInstance(oid:=nObjID, iid:=iid, pipUnk:=ADR(ip));
    ELSE
        hrInit := E_HRESULTAdsErr.INVALIDOBJID;
    END_IF
END_IF
```

6. It is imperative to release the used reference again. To this end call the FW_SafeRelease() function in the FB_exit destructor of the function block.

```
FB_IncrementProxy.FB_exit | FB_IncrementProxy.FB_init
{attribute 'hide'}
METHOD FB_exit : BOOL
VAR_INPUT
    bInCopyCode : BOOL; // if TRUE, the exit method is called for
END_VAR

IF NOT bInCopyCode THEN // if not online change
    FW_SafeRelease(ADR(ip));
END_IF
```

⇨ This completes the implementation of the Proxy function block.

7. Declare an instance of the Proxy function block to call the methods provided via the interface in the application.
   The calls themselves take all place over the interface pointer defined as output of the function block. As is typical for pointers a prior null check must be made. Then the methods can be called directly, also via Intellisense.

```
MAIN*
PROGRAM MAIN
VAR
    fbInc : FB_IncrementProxy;
    nValue : UDINT;
END_VAR

IF fbInc.ip <> 0 THEN
    fbInc.ip.doIncrement(4, ADR(nValue));
END_IF
```

⇨ The sample is ready for testing.

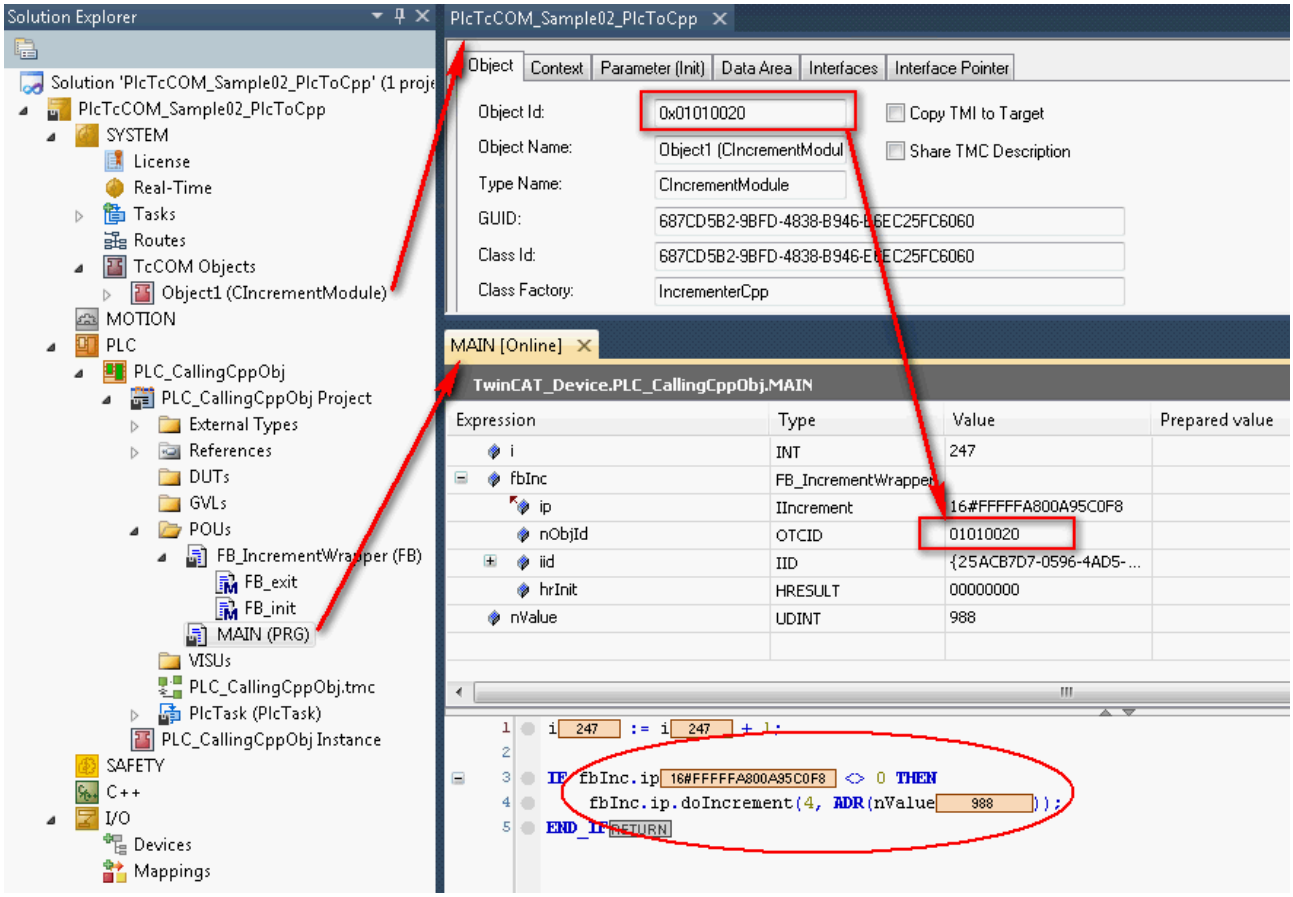
15.25.2.3 Execution of the sample project

1. Select the target system and compile the project.
2. Enable the TwinCAT configuration, then log in and start the PLC.

3. This C++ TcCOM module requires Windows Test Mode. Activate this to use the module and thus the example.
4. When activating the TwinCAT configuration for the first time, the test certificate must be accepted:



⇨ In the online view of the PLC application, the assigned object ID of the C++ object is indicated in the PLC Proxy function block. The interface pointer has a valid value and the method will be executed.



15.25.3 TcCOM_Sample03_PlcCreatesCpp

Just like Sample02, this sample describes a TcCOM communication between PLC and C++. To this end a PLC application uses functionalities of a TwinCAT C++ class. The required instances of this C++ class will be created by the PLC itself in this sample. In this way own algorithms written in C++ can be used easily in the PLC.
Although in the event of the use of an existing TwinCAT C++ driver the TwinCAT C++ license is required on the destination system, a C++ development environment is not necessary on the destination system or on the development computer.

An already built C++ driver provides one or more classes whose interfaces are deposited in the TMC description file and thus are known in the PLC.

The procedure is explained in the following sub-chapters:

1. Provision of a TwinCAT C++ driver and its classes [▶ 319]

2. Creating an FB in the PLC that creates the C++ object and offers its functionality [▶ 320]

3. Execution of the sample project [▶ 322]

Downloading the sample: https://infosys.beckhoff.com/content/1033/TC3_C/Resources/2343051531.zip

System requirements

TwinCAT version	Hardware	Libraries to be integrated
TwinCAT 3.1, Build 4020	x86, x64	Tc3_Module

15.25.3.1 Provision of the binary C++ project (TMX) and its classes

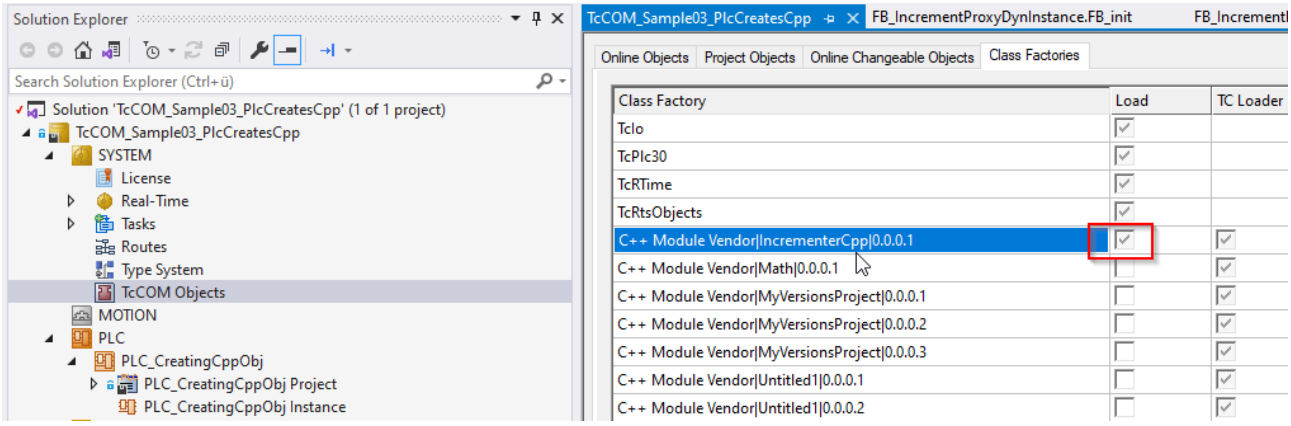
The binary TwinCAT C++ project must be available on the engineering system so that it can be transferred to the target system together with the PLC project during activation.

TwinCAT offers a deployment mechanism for distribution between engineering systems, which is described at Versioned C++ Projects [▶ 49].
(This sample includes in the download the corresponding TMX, because TwinCAT places them automatically in the archive, if the Class Factory is used.)

Open a TwinCAT project or create a new project.

1. Select the required C++ driver in the Solution under the **TcCOM Objects** node in the **Class Factories** tab. The checkbox can also be set automatically by TwinCAT if you implement this accordingly (as in the sample here).

⇨ This ensures that the driver is transferred and loaded on the target system when TwinCAT is started.

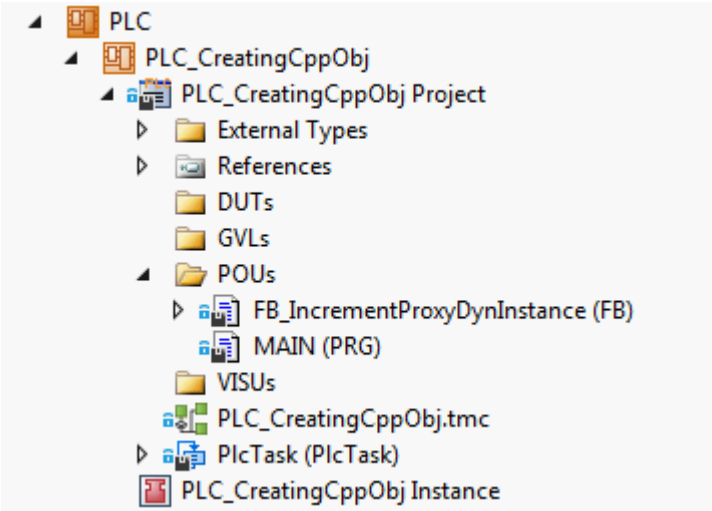


Creation of the binary C++ project

The documentation for TwinCAT C++ [▶ 10] explains in detail how to create the TMX.
For Sample03 it should be noted that TwinCAT C++ modules whose classes are to be dynamically instantiated must be defined as "TwinCAT Module Class for RT Context". The C++ wizard offers a special template for this purpose.
Furthermore, this sample uses a TwinCAT C++ class that does without TcCOM initialization data and without TcCOM parameters.

15.25.3.2 Creating an FB in the PLC that creates the C++ object and offers its functionality

1. Create a PLC and append a new function block there.
   This Proxy function block should provide the functionality that was programmed in C++. It manages this via an interface pointer that was defined by C++ and is known in the PLC due to the TMC description file.

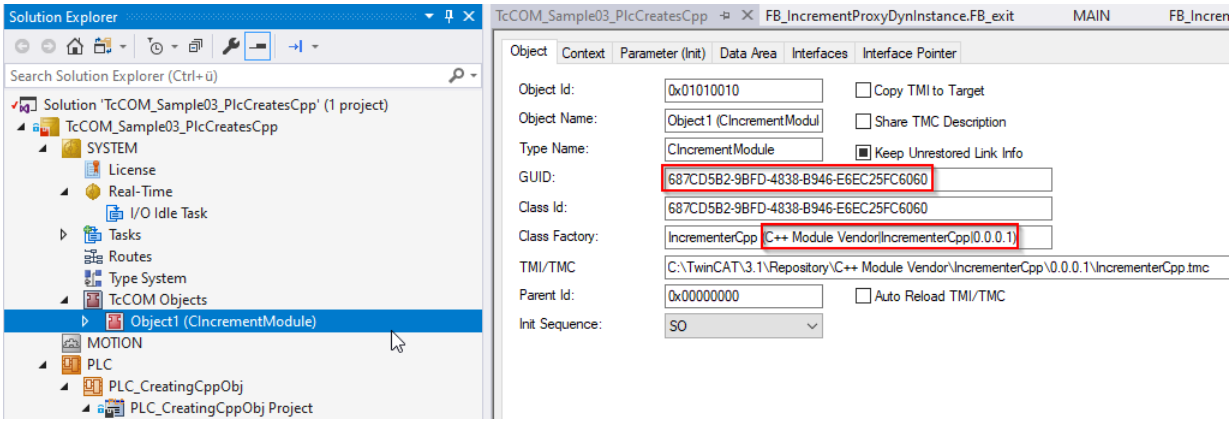


2. In the declaration part of the function block declare as an output an interface pointer to the interface (IIncrement) which later provides the functionality outward.

```
{attribute 'TcDependOnClassFactory' := 'C++ Module Vendor|IncrementerCpp|0.0.0.1'}
FUNCTION_BLOCK FB_IncrementProxyDynInstance
VAR_OUTPUT
    ip : IIncrement;
END_VAR

VAR
    objName : STRING;
    classId : CLSID := STRING_TO_GUID('687cd5b2-9bfd-4838-b946-e6ec25fc6060');
    sLibraryId : STRING := 'C++ Module Vendor|IncrementerCpp|0.0.0.1';
    classIdVersioned : CLSID;
    iid : IID := TC_GLOBAL_IID_LIST.IID_IIncrement;
    hrInit : HRESULT;
END_VAR
```

3. Create a library ID, class ID, and the interface ID as member variables, as shown in the previous step. While the interface ID is already available via a global list, the library ID and the class ID, provided they are not yet supposed to be known, are determined by other means. One possible way to do this is to create an instance temporarily and take the information from the dialog before it can be deleted again:



4. Add the FB_init constructor method to the PLC Proxy function block.
   For the case, that it is not an online change but rather the initialization of the function block, a new TcCOM object (Class instance of the specified class) is created and the interface pointer to the specified interface is obtained.
   First, the versioned class ID is calculated from the library ID and the class ID using the F_GetClassIdVersioned() method. Then the used FW_ObjMgr_CreateAndInitInstance() function is also given the name and the target state of the TcCOM object. These two parameters are declared here as input parameters of the FB_init method, whereby they are to be specified in the instantiation of the Proxy function block. The TwinCAT C++ class to be instantiated does without TcCOM initialization data and without TcCOM parameters.
   With this function call the object itself counts up a reference counter.

```
FB_IncrementProxyDynInstance.FB_init  FB_IncrementProxyDynInstance
METHOD FB_init : BOOL
VAR_INPUT
    bInitRetains : BOOL; // if TRUE, the retain variables are initialized (warm start / cold start)
    bInCopyCode : BOOL;  // if TRUE, the instance afterwards gets moved into the copy code (online change)

    sObjName : STRING;      // object name to be set for this instance (optional)
    eObjState : TCOM_STATE; // target object state (usually TCOM_STATE.TCOM_STATE_OP)
END_VAR

IF NOT bInCopyCode THEN // if not online change
    objName := sObjName;
    F_GetClassIdVersioned(sLibraryId:=sLibraryId, clsId:=classId, clsIdVersioned:=classIdVersioned);
    hrInit := FW_ObjMgr_CreateAndInitInstance( clsId       := classIdVersioned,
                                               iid         := iid,
                                               pipUnk      := ADR(ip),
                                               objId       := OTCID_CreateNewId,
                                               parentId    := TwinCAT_SystemInfoVarList._AppInfo.ObjId, // set PLC instance as parent
                                               name        := sObjName,
                                               state       := eObjState,
                                               pInitData   := 0 );
END_IF
```

5. It is imperative to release the used reference again and to delete the object, provided it is no longer being used. To this end call the FW_ObjMgr_DeleteInstance() function in the FB_exit destructor of the function block.

```
FB_IncrementProxyDynInstance.FB_exit  FB_IncrementProxyDynInstance.FB_init  FB_IncrementProxyDynInstance
{attribute 'hide'}
METHOD FB_exit : BOOL
VAR_INPUT
    bInCopyCode : BOOL; // if TRUE, the exit method is called for exiting an instance that is copied afterwards (online change).
END_VAR

IF NOT bInCopyCode THEN // if not online change
    FW_ObjMgr_DeleteInstance(ADR(ip));
END_IF
```

   ⇨ This completes the implementation of the Proxy function block.

6. Declare an instance of the Proxy function block to call the methods provided via the interface in the application. The calls themselves take all place over the interface pointer defined as output of the function block. As is typical for pointers a prior null check must be made. Then the methods can be called directly, also via Intellisense.

```
MAIN  FB_IncrementProxyDynInstance.FB_exit  FB_IncrementProxyDynInstance.FB_init  FB_IncrementProxyDynInstance
PROGRAM MAIN
VAR
    fbInc : FB_IncrementProxyDynInstance(   sObjName:='CIncrementModule:fbInc',
                                            eObjState:=TCOM_STATE.TCOM_STATE_OP);
    nValue : UDINT;
END_VAR


IF fbInc.ip <> 0 THEN
    fbInc.ip.doIncrement(100, ADR(nValue));
END_IF
```

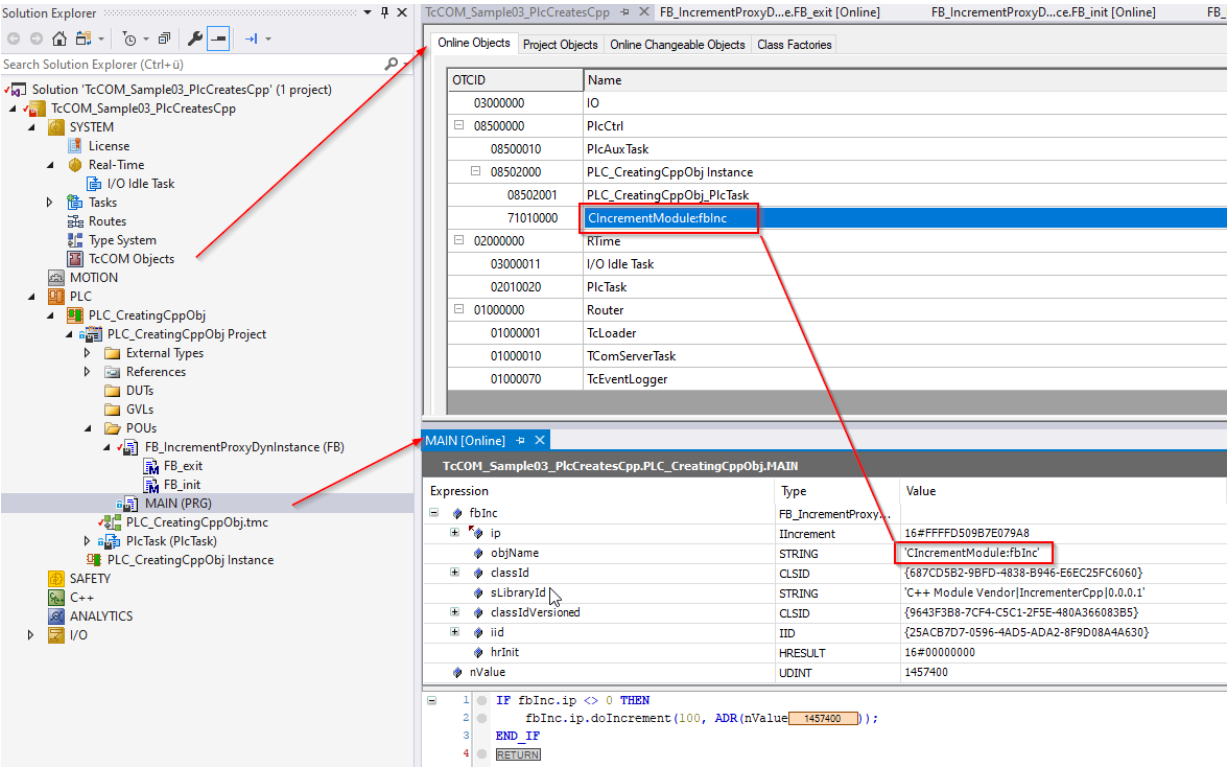
⇨ The sample is ready for testing.

15.25.3.3 Execution of the sample project

1. Select the target system and compile the project.
2. Enable the TwinCAT configuration, then log in and start the PLC.
3. This C++ TcCOM module requires Windows Test Mode. Activate this to use the module and thus the example.
4. When activating the TwinCAT configuration for the first time, the test certificate must be accepted:



⇨ In the online view of the PLC application, the desired TcCOM object name is indicated in the PLC Proxy function block. The project node **TcCOM objects** contains the generated object with the generated ID and the desired name in its list. The interface pointer has a valid value and the method will be executed.



16 Appendix

- The ADS Return Codes [▶ 323] are important across TwinCAT 3, particularly if ADS communication [▶ 189] itself is implemented.
- The Retain data [▶ 326] (in NOVRAM memory) can be used in a similar way from the PLC and also C++.
- In addition to the TcCOM Module [▶ 35] concept, the TwinCAT 3 type system is an important basis for understanding.
- The following pages originate from the documentation for the Automation Interface.
  When using the Automation Interface please refer to the dedicated documentation.
  - Creating and handling C++ projects and modules [▶ 329]
  - Creating and handling TcCOM modules [▶ 333]

16.1 ADS Return Codes

Grouping of error codes:
Global error codes: 0x0000 [▶ 323]... (0x9811_0000 ...)
Router error codes: 0x500 [▶ 324]... (0x9811_0500 ...)
General ADS errors: 0x700 [▶ 324]... (0x9811_0700 ...)
RTime error codes: 0x1000 [▶ 325]... (0x9811_1000 ...)

Global error codes

Hex	Dec	HRESULT	Name	Description
0x0	0	0x98110000	ERR_NOERROR	No error.
0x1	1	0x98110001	ERR_INTERNAL	Internal error.
0x2	2	0x98110002	ERR_NORTIME	No real time.
0x3	3	0x98110003	ERR_ALLOCLOCKEDMEM	Allocation locked – memory error.
0x4	4	0x98110004	ERR_INSERTMAILBOX	Mailbox full – the ADS message could not be sent. Reducing the number of ADS messages per cycle will help.
0x5	5	0x98110005	ERR_WRONGRECEIVEHMSG	Wrong HMSG.
0x6	6	0x98110006	ERR_TARGETPORTNOTFOUND	Target port not found – ADS server is not started, not reachable or not installed.
0x7	7	0x98110007	ERR_TARGETMACHINENOTFOUND	Target computer not found – AMS route was not found.
0x8	8	0x98110008	ERR_UNKNOWNCMDID	Unknown command ID.
0x9	9	0x98110009	ERR_BADTASKID	Invalid task ID.
0xA	10	0x9811000A	ERR_NOIO	No IO.
0xB	11	0x9811000B	ERR_UNKNOWNAMSCMD	Unknown AMS command.
0xC	12	0x9811000C	ERR_WIN32ERROR	Win32 error.
0xD	13	0x9811000D	ERR_PORTNOTCONNECTED	Port not connected.
0xE	14	0x9811000E	ERR_INVALIDAMSLENGTH	Invalid AMS length.
0xF	15	0x9811000F	ERR_INVALIDAMSNETID	Invalid AMS Net ID.
0x10	16	0x98110010	ERR_LOWINSTLEVEL	Installation level is too low –TwinCAT 2 license error.
0x11	17	0x98110011	ERR_NODEBUGINTAVAILABLE	No debugging available.
0x12	18	0x98110012	ERR_PORTDISABLED	Port disabled – TwinCAT system service not started.
0x13	19	0x98110013	ERR_PORTALREADYCONNECTED	Port already connected.
0x14	20	0x98110014	ERR_AMSSYNC_W32ERROR	AMS Sync Win32 error.
0x15	21	0x98110015	ERR_AMSSYNC_TIMEOUT	AMS Sync Timeout.
0x16	22	0x98110016	ERR_AMSSYNC_AMSERROR	AMS Sync error.
0x17	23	0x98110017	ERR_AMSSYNC_NOINDEXINMAP	No index map for AMS Sync available.
0x18	24	0x98110018	ERR_INVALIDAMSPORT	Invalid AMS port.
0x19	25	0x98110019	ERR_NOMEMORY	No memory.
0x1A	26	0x9811001A	ERR_TCPSEND	TCP send error.
0x1B	27	0x9811001B	ERR_HOSTUNREACHABLE	Host unreachable.
0x1C	28	0x9811001C	ERR_INVALIDAMSFRAGMENT	Invalid AMS fragment.

Hex	Dec	HRESULT	Name	Description
0x1D	29	0x9811001D	ERR_TLSSEND	TLS send error – secure ADS connection failed.
0x1E	30	0x9811001E	ERR_ACCESSDENIED	Access denied – secure ADS access denied.

Router error codes

Hex	Dec	HRESULT	Name	Description
0x500	1280	0x98110500	ROUTERERR_NOLOCKEDMEMORY	Locked memory cannot be allocated.
0x501	1281	0x98110501	ROUTERERR_RESIZEMEMORY	The router memory size could not be changed.
0x502	1282	0x98110502	ROUTERERR_MAILBOXFULL	The mailbox has reached the maximum number of possible messages.
0x503	1283	0x98110503	ROUTERERR_DEBUGBOXFULL	The Debug mailbox has reached the maximum number of possible messages.
0x504	1284	0x98110504	ROUTERERR_UNKNOWNPORTTYPE	The port type is unknown.
0x505	1285	0x98110505	ROUTERERR_NOTINITIALIZED	The router is not initialized.
0x506	1286	0x98110506	ROUTERERR_PORTALREADYINUSE	The port number is already assigned.
0x507	1287	0x98110507	ROUTERERR_NOTREGISTERED	The port is not registered.
0x508	1288	0x98110508	ROUTERERR_NOMOREQUEUES	The maximum number of ports has been reached.
0x509	1289	0x98110509	ROUTERERR_INVALIDPORT	The port is invalid.
0x50A	1290	0x9811050A	ROUTERERR_NOTACTIVATED	The router is not active.
0x50B	1291	0x9811050B	ROUTERERR_FRAGMENTBOXFULL	The mailbox has reached the maximum number for fragmented messages.
0x50C	1292	0x9811050C	ROUTERERR_FRAGMENTTIMEOUT	A fragment timeout has occurred.
0x50D	1293	0x9811050D	ROUTERERR_TOBEREMOVED	The port is removed.

General ADS error codes

Hex	Dec	HRESULT	Name	Description
0x700	1792	0x98110700	ADSERR_DEVICE_ERROR	General device error.
0x701	1793	0x98110701	ADSERR_DEVICE_SRVNOTSUPP	Service is not supported by the server.
0x702	1794	0x98110702	ADSERR_DEVICE_INVALIDGRP	Invalid index group.
0x703	1795	0x98110703	ADSERR_DEVICE_INVALIDOFFSET	Invalid index offset.
0x704	1796	0x98110704	ADSERR_DEVICE_INVALIDACCESS	Reading or writing not permitted. Several causes are possible. For example, an incorrect password was entered when creating routes.
0x705	1797	0x98110705	ADSERR_DEVICE_INVALIDSIZE	Parameter size not correct.
0x706	1798	0x98110706	ADSERR_DEVICE_INVALIDDATA	Invalid data values.
0x707	1799	0x98110707	ADSERR_DEVICE_NOTREADY	Device is not ready to operate.
0x708	1800	0x98110708	ADSERR_DEVICE_BUSY	Device is busy.
0x709	1801	0x98110709	ADSERR_DEVICE_INVALIDCONTEXT	Invalid operating system context. This can result from use of ADS blocks in different tasks. It may be possible to resolve this through multitasking synchronization in the PLC.
0x70A	1802	0x9811070A	ADSERR_DEVICE_NOMEMORY	Insufficient memory.
0x70B	1803	0x9811070B	ADSERR_DEVICE_INVALIDPARM	Invalid parameter values.
0x70C	1804	0x9811070C	ADSERR_DEVICE_NOTFOUND	Not found (files, ...).
0x70D	1805	0x9811070D	ADSERR_DEVICE_SYNTAX	Syntax error in file or command.
0x70E	1806	0x9811070E	ADSERR_DEVICE_INCOMPATIBLE	Objects do not match.
0x70F	1807	0x9811070F	ADSERR_DEVICE_EXISTS	Object already exists.
0x710	1808	0x98110710	ADSERR_DEVICE_SYMBOLNOTFOUND	Symbol not found.
0x711	1809	0x98110711	ADSERR_DEVICE_SYMBOLVERSIONINVALID	Invalid symbol version. This can occur due to an online change. Create a new handle.
0x712	1810	0x98110712	ADSERR_DEVICE_INVALIDSTATE	Device (server) is in invalid state.
0x713	1811	0x98110713	ADSERR_DEVICE_TRANSMODENOTSUPP	AdsTransMode not supported.
0x714	1812	0x98110714	ADSERR_DEVICE_NOTIFYHNDINVALID	Notification handle is invalid.
0x715	1813	0x98110715	ADSERR_DEVICE_CLIENTUNKNOWN	Notification client not registered.
0x716	1814	0x98110716	ADSERR_DEVICE_NOMOREHDLS	No further handle available.
0x717	1815	0x98110717	ADSERR_DEVICE_INVALIDWATCHSIZE	Notification size too large.
0x718	1816	0x98110718	ADSERR_DEVICE_NOTINIT	Device not initialized.

Hex	Dec	HRESULT	Name	Description
0x719	1817	0x98110719	ADSERR_DEVICE_TIMEOUT	Device has a timeout.
0x71A	1818	0x9811071A	ADSERR_DEVICE_NOINTERFACE	Interface query failed.
0x71B	1819	0x9811071B	ADSERR_DEVICE_INVALIDINTERFACE	Wrong interface requested.
0x71C	1820	0x9811071C	ADSERR_DEVICE_INVALIDCLSID	Class ID is invalid.
0x71D	1821	0x9811071D	ADSERR_DEVICE_INVALIDOBJID	Object ID is invalid.
0x71E	1822	0x9811071E	ADSERR_DEVICE_PENDING	Request pending.
0x71F	1823	0x9811071F	ADSERR_DEVICE_ABORTED	Request is aborted.
0x720	1824	0x98110720	ADSERR_DEVICE_WARNING	Signal warning.
0x721	1825	0x98110721	ADSERR_DEVICE_INVALIDARRAYIDX	Invalid array index.
0x722	1826	0x98110722	ADSERR_DEVICE_SYMBOLNOTACTIVE	Symbol not active.
0x723	1827	0x98110723	ADSERR_DEVICE_ACCESSDENIED	Access denied. Several causes are possible. For example, a unidirectional ADS route is used in the opposite direction.
0x724	1828	0x98110724	ADSERR_DEVICE_LICENSENOTFOUND	Missing license.
0x725	1829	0x98110725	ADSERR_DEVICE_LICENSEEXPIRED	License expired.
0x726	1830	0x98110726	ADSERR_DEVICE_LICENSEEXCEEDED	License exceeded.
0x727	1831	0x98110727	ADSERR_DEVICE_LICENSEINVALID	Invalid license.
0x728	1832	0x98110728	ADSERR_DEVICE_LICENSESYSTEMID	License problem: System ID is invalid.
0x729	1833	0x98110729	ADSERR_DEVICE_LICENSENOTIMELIMIT	License not limited in time.
0x72A	1834	0x9811072A	ADSERR_DEVICE_LICENSEFUTUREISSUE	Licensing problem: time in the future.
0x72B	1835	0x9811072B	ADSERR_DEVICE_LICENSETIMETOLONG	License period too long.
0x72C	1836	0x9811072C	ADSERR_DEVICE_EXCEPTION	Exception at system startup.
0x72D	1837	0x9811072D	ADSERR_DEVICE_LICENSEDUPLICATED	License file read twice.
0x72E	1838	0x9811072E	ADSERR_DEVICE_SIGNATUREINVALID	Invalid signature.
0x72F	1839	0x9811072F	ADSERR_DEVICE_CERTIFICATEINVALID	Invalid certificate.
0x730	1840	0x98110730	ADSERR_DEVICE_LICENSEOEMNOTFOUND	Public key not known from OEM.
0x731	1841	0x98110731	ADSERR_DEVICE_LICENSERESTRICTED	License not valid for this system ID.
0x732	1842	0x98110732	ADSERR_DEVICE_LICENSEDEMODENIED	Demo license prohibited.
0x733	1843	0x98110733	ADSERR_DEVICE_INVALIDFNCID	Invalid function ID.
0x734	1844	0x98110734	ADSERR_DEVICE_OUTOFRANGE	Outside the valid range.
0x735	1845	0x98110735	ADSERR_DEVICE_INVALIDALIGNMENT	Invalid alignment.
0x736	1846	0x98110736	ADSERR_DEVICE_LICENSEPLATFORM	Invalid platform level.
0x737	1847	0x98110737	ADSERR_DEVICE_FORWARD_PL	Context – forward to passive level.
0x738	1848	0x98110738	ADSERR_DEVICE_FORWARD_DL	Context – forward to dispatch level.
0x739	1849	0x98110739	ADSERR_DEVICE_FORWARD_RT	Context – forward to real-time.
0x740	1856	0x98110740	ADSERR_CLIENT_ERROR	Client error.
0x741	1857	0x98110741	ADSERR_CLIENT_INVALIDPARM	Service contains an invalid parameter.
0x742	1858	0x98110742	ADSERR_CLIENT_LISTEMPTY	Polling list is empty.
0x743	1859	0x98110743	ADSERR_CLIENT_VARUSED	Var connection already in use.
0x744	1860	0x98110744	ADSERR_CLIENT_DUPLINVOKEID	The called ID is already in use.
0x745	1861	0x98110745	ADSERR_CLIENT_SYNCTIMEOUT	Timeout has occurred – the remote terminal is not responding in the specified ADS timeout. The route setting of the remote terminal may be configured incorrectly.
0x746	1862	0x98110746	ADSERR_CLIENT_W32ERROR	Error in Win32 subsystem.
0x747	1863	0x98110747	ADSERR_CLIENT_TIMEOUTINVALID	Invalid client timeout value.
0x748	1864	0x98110748	ADSERR_CLIENT_PORTNOTOPEN	Port not open.
0x749	1865	0x98110749	ADSERR_CLIENT_NOAMSADDR	No AMS address.
0x750	1872	0x98110750	ADSERR_CLIENT_SYNCINTERNAL	Internal error in Ads sync.
0x751	1873	0x98110751	ADSERR_CLIENT_ADDHASH	Hash table overflow.
0x752	1874	0x98110752	ADSERR_CLIENT_REMOVEHASH	Key not found in the table.
0x753	1875	0x98110753	ADSERR_CLIENT_NOMORESYM	No symbols in the cache.
0x754	1876	0x98110754	ADSERR_CLIENT_SYNCRESINVALID	Invalid response received.
0x755	1877	0x98110755	ADSERR_CLIENT_SYNCPORTLOCKED	Sync Port is locked.
0x756	1878	0x98110756	ADSERR_CLIENT_REQUESTCANCELLED	The request was canceled.

RTime error codes

Hex	Dec	HRESULT	Name	Description
0x1000	4096	0x98111000	RTERR_INTERNAL	Internal error in the real-time system.
0x1001	4097	0x98111001	RTERR_BADTIMERPERIODS	Timer value is not valid.
0x1002	4098	0x98111002	RTERR_INVALIDTASKPTR	Task pointer has the invalid value 0 (zero).
0x1003	4099	0x98111003	RTERR_INVALIDSTACKPTR	Stack pointer has the invalid value 0 (zero).
0x1004	4100	0x98111004	RTERR_PRIOEXISTS	The request task priority is already assigned.
0x1005	4101	0x98111005	RTERR_NOMORETCB	No free TCB (Task Control Block) available. The maximum number of TCBs is 64.
0x1006	4102	0x98111006	RTERR_NOMORESEMAS	No free semaphores available. The maximum number of semaphores is 64.
0x1007	4103	0x98111007	RTERR_NOMOREQUEUES	No free space available in the queue. The maximum number of positions in the queue is 64.
0x100D	4109	0x9811100D	RTERR_EXTIRQALREADYDEF	An external synchronization interrupt is already applied.
0x100E	4110	0x9811100E	RTERR_EXTIRQNOTDEF	No external sync interrupt applied.
0x100F	4111	0x9811100F	RTERR_EXTIRQINSTALLFAILED	Application of the external synchronization interrupt has failed.
0x1010	4112	0x98111010	RTERR_IRQLNOTLESSOREQUAL	Call of a service function in the wrong context
0x1017	4119	0x98111017	RTERR_VMXNOTSUPPORTED	Intel® VT-x extension is not supported.
0x1018	4120	0x98111018	RTERR_VMXDISABLED	Intel® VT-x extension is not enabled in the BIOS.
0x1019	4121	0x98111019	RTERR_VMXCONTROLSMISSING	Missing function in Intel® VT-x extension.
0x101A	4122	0x9811101A	RTERR_VMXENABLEFAILS	Activation of Intel® VT-x fails.

Specific positive HRESULT Return Codes:

HRESULT	Name	Description
0x0000_0000	S_OK	No error.
0x0000_0001	S_FALSE	No error. Example: successful processing, but with a negative or incomplete result.
0x0000_0203	S_PENDING	No error. Example: successful processing, but no result is available yet.
0x0000_0256	S_WATCHDOG_TIMEOUT	No error. Example: successful processing, but a timeout occurred.

TCP Winsock error codes

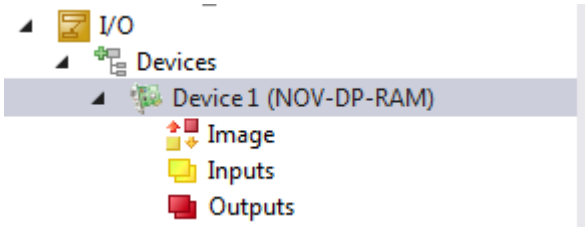
Hex	Dec	Name	Description
0x274C	10060	WSAETIMEDOUT	A connection timeout has occurred - error while establishing the connection, because the remote terminal did not respond properly after a certain period of time, or the established connection could not be maintained because the connected host did not respond.
0x274D	10061	WSAECONNREFUSED	Connection refused - no connection could be established because the target computer has explicitly rejected it. This error usually results from an attempt to connect to a service that is inactive on the external host, that is, a service for which no server application is running.
0x2751	10065	WSAEHOSTUNREACH	No route to host - a socket operation referred to an unavailable host.
More Winsock error codes: Win32 error codes			

16.2 Retain data

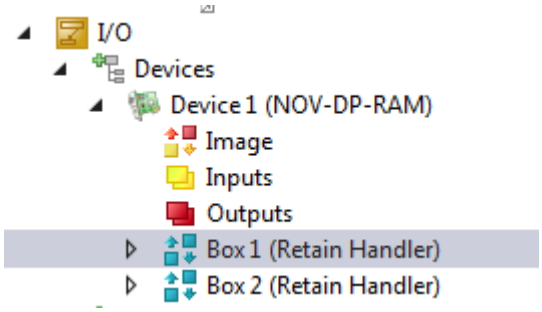
This section describes the option to make data available even after an ordered or spontaneous system restart. The NOV-RAM of a device is used for this purpose. The EL6080 cannot be used for these retain data, because the corresponding data must first be transferred, which leads to corresponding runtimes. The following section describes the retain handler, which stores data and makes them available again, and the application of the different TwinCAT 3 programming languages.

Configuring a retain device

1. The retain data are stored and made available by a retain handler, which is part of the NOV-DP-RAM device in the IO section of the TwinCAT solution. Create a NOV-RAM DP Device in the IO area of the Solution.

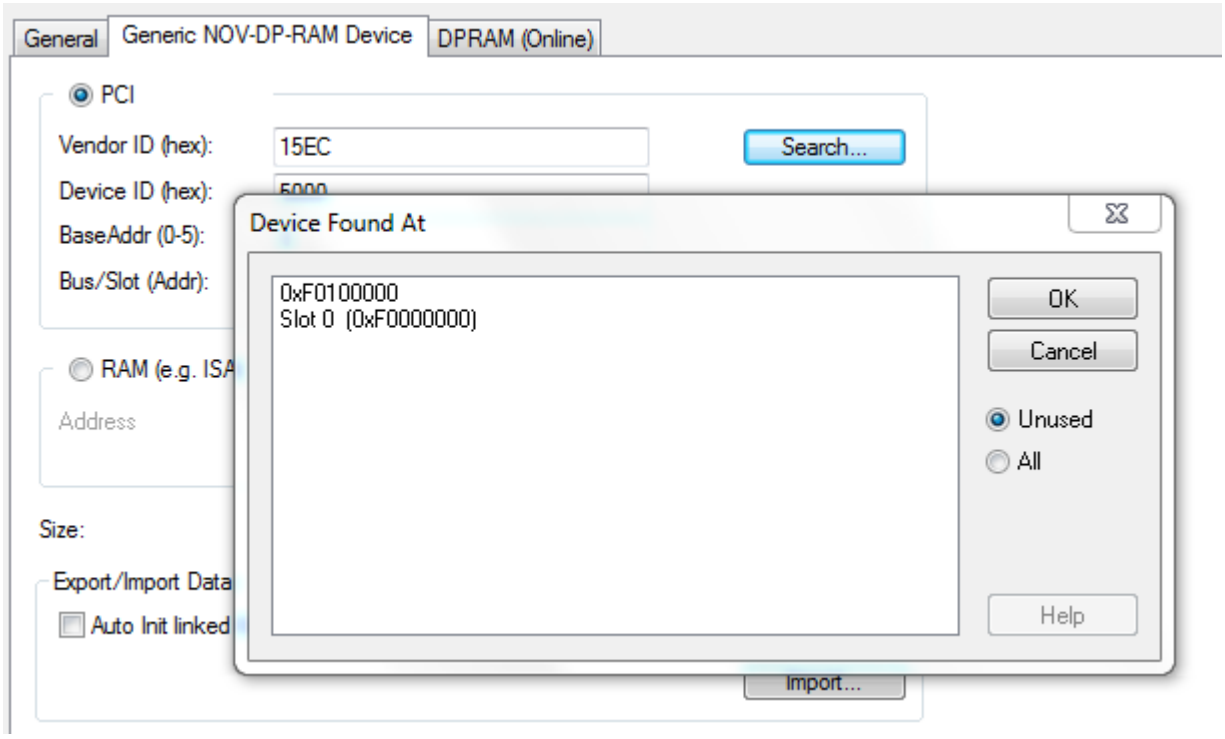


2. Create one or more Retain Handler below this device.



Storage location: NOVRAM

3. Configure the NOV-DP RAM device. In the **Generic NOV-DP-RAM Device** tab, use **Search...** to define the area to be used.



4. An additional retain directory for the symbols is created in the TwinCAT boot directory.

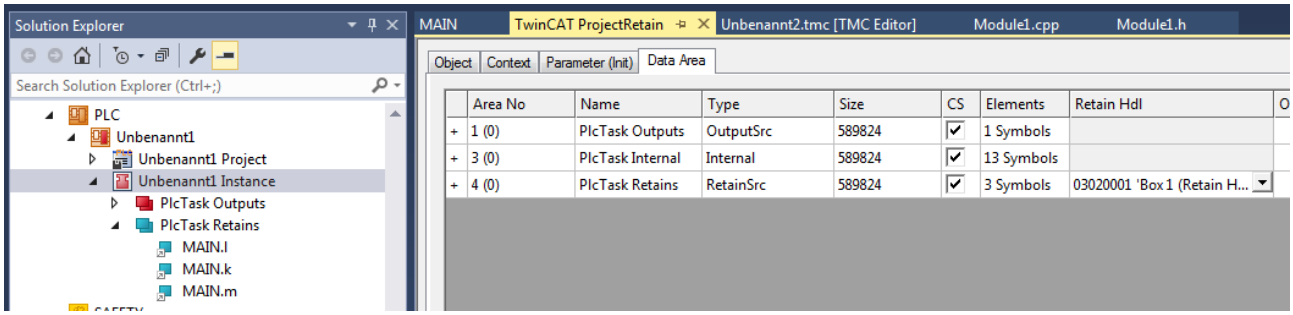
Using the retain handler with a PLC project

In a PLC project the variables are either created in a VAR RETAIN area or identified with the attribute TcRetain.

```
PROGRAM MAIN
VAR RETAIN
  l: UINT;
  k: UINT;
END_VAR
VAR
```

```
  {attribute 'TcRetain':='1'}
  m: UINT;
  x: UINT;
END_VAR
```

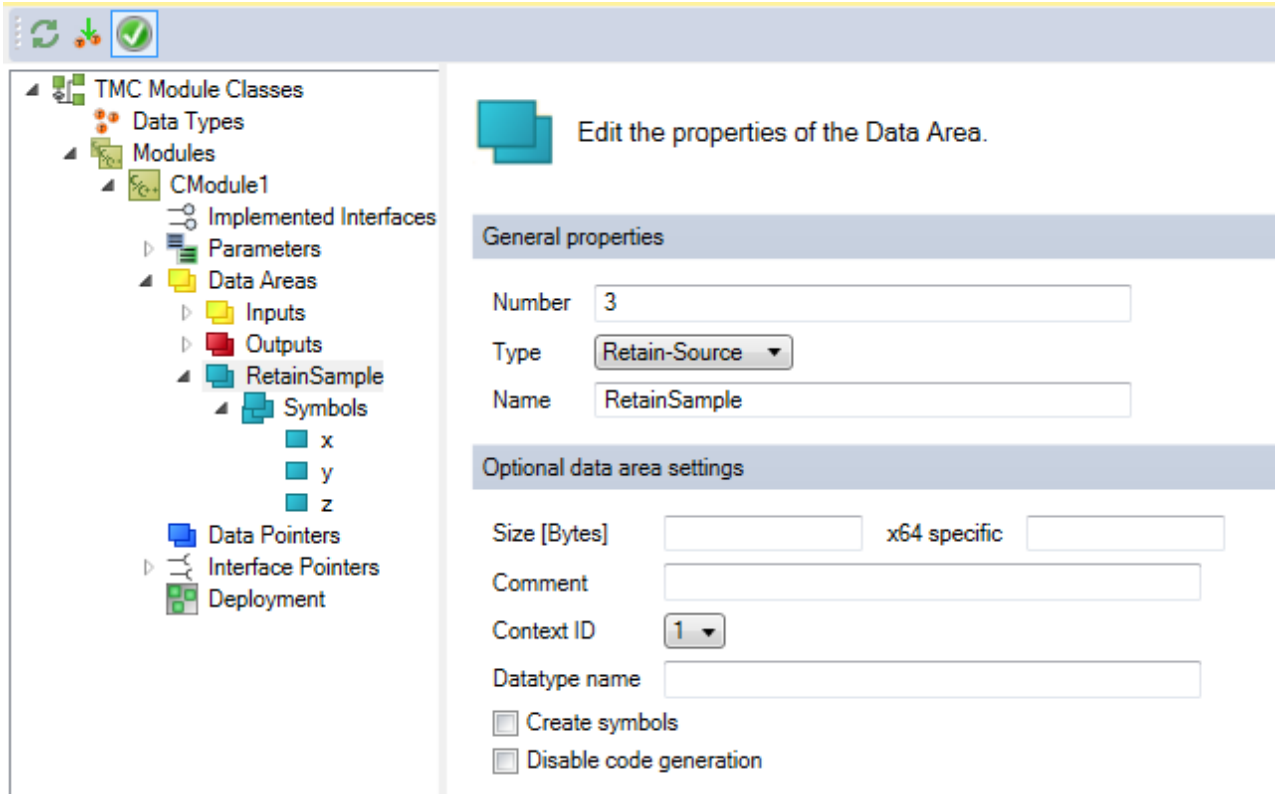
Corresponding symbols are created after a "Build".
The assignment to the retain handler of the NOV-DP-RAM device is done in column **Retain Hdl**.



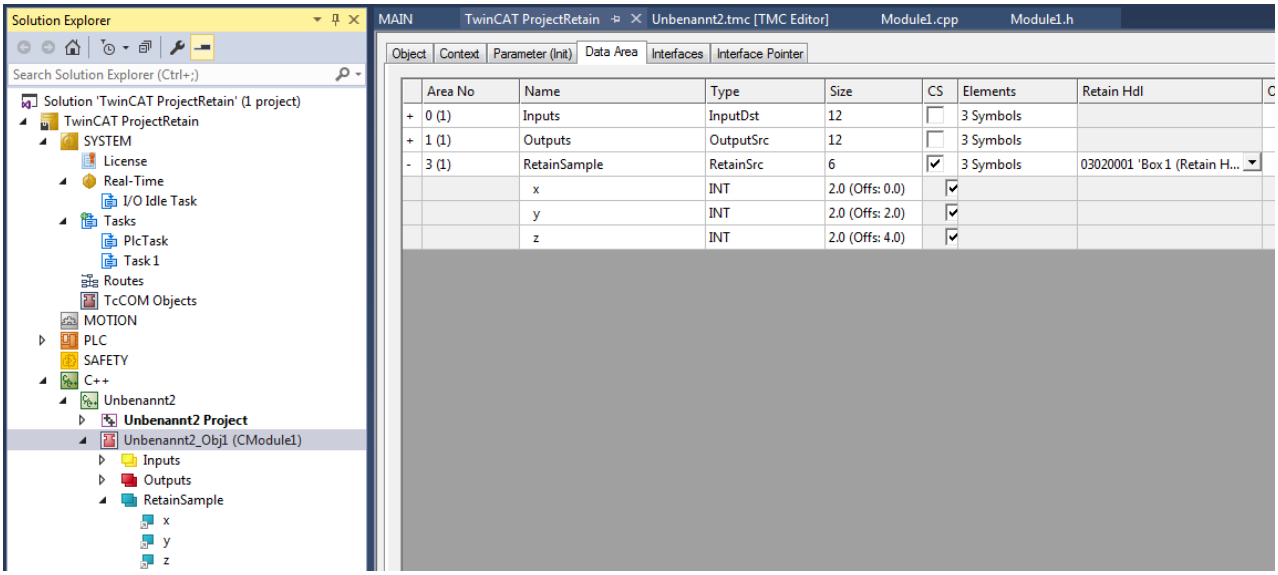
If self-defined data types (DUTs) are used as retain, the data types must be available in the TwinCAT type system. You can either use the option **Convert to Global Type** or you can create structures directly as `STRUCT RETAIN` . However, the Retain Handler then handles all occurrences of the structure.
Retain data cannot be used for POUs (function blocks) as a whole. However, individual elements of a POU can be used.

Using the retain handler with a C++ module

In a C++ module a data area of type Retain Source is created, which contains the corresponding symbols.

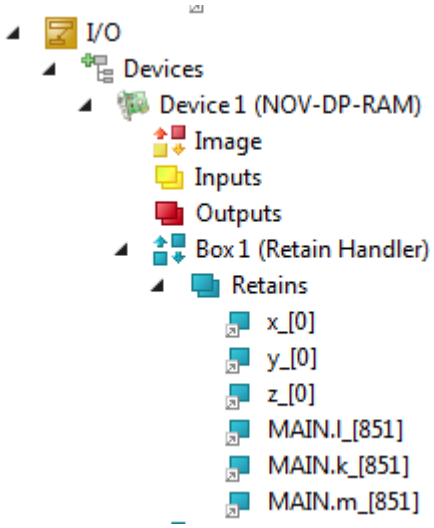


At the instances of the C++ module, a retain handler of the NOV-DP-RAM device to be used for this data area is defined in column **Retain Hdl**.



Conclusions

When a retain handler is selected as target in the respective project, the symbols under retain handler and a mapping are created automatically after a "Build".



16.3 Creating and handling C++ projects and modules

This chapter explains in-depth how to create, access and handle TwinCAT C++ projects. The following list shows all chapters in this article:

- General information about C++ projects
- Creating new C++ projects
- Creating new module within a C++ project
- Opening existing C++ projects
- Creating module instances
- Calling TMC Code Generator
- Calling Publish Modules command
- Setting C++ Project Properties
- Building project

General information about C++ projects

C++ projects are specified by their so-called project template, which are used by the “TwinCAT C++ Project Wizard”. Inside a project multiple modules could be defined by module templates, which are used by the “TwinCAT Class Wizard”.

TwinCAT-defined templates are documented in the Section C++ / Wizards [▶ 85].

The customer could define own templates, which is documented at the corresponding sub-section if C++ Section / Wizards [▶ 133].

Create C++ projects

To create a new C++ project using the Automation Interface, you must navigate to the C++ node and then execute the CreateChild() method with the appropriate template file as a parameter.

Code snippet (C#):

```
ITcSmTreeItem cpp = systemManager.LookupTreeItem("TIXC");
ITcSmTreeItem cppProject = cpp.CreateChild("NewCppProject", 0, "", pathToTemplateFile);
```

Code snippet (PowerShell):

```
$cpp = $systemManager.LookupTreeItem("TIXC")
$newProject = $cpp.CreateChild("NewCppProject", 0, "", $pathToTemplateFile)
```

To instantiate a driver project, use "TcVersionedDriverWizard" as `pathToTemplateFile`.

Creating new module within a C++ project

Within a C++ project usually a TwinCAT Module Wizard is used to let the wizard create a module by a template.

Code snippet (C#):

```
ITcSmTreeItem cppModule = cppProject.CreateChild("NewModule", 1, "", pathToTemplateFile);
```

Code snippet (PowerShell):

```
$cppModule = $cppProject.CreateChild("NewModule", 0, "", $pathToTemplateFile);
```

As example for instantiating a Cyclic IO module project please use "TcModuleCyclicCallerWizard " as `pathToTemplateFile`.

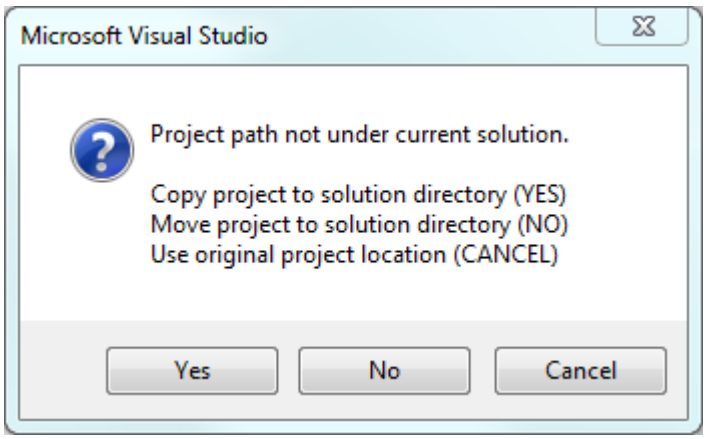
Open existing C++ projects

To open an existing C++ project using Automation Interface, you must navigate to the C++ node and then execute the CreateChild() method with the path of the corresponding C++ project file as a parameter.

You can use three different values as SubType:

- 0: Copy project to Solution directory
- 1: Move project to the Solution directory
- 2: Use the original project location (specify "" as NameOfProject parameter)

Basically, these values represent the functionalities (Yes, No, Cancel) of the following MessageBox in TwinCAT XAE:



Instead of the template file, you must use the path to the C++ project (or its vcxproj file) that must be added. Alternatively, you can also use a C++ project archive (tczip file).

Code snippet (C#):

```
ITcSmTreeItem cpp = systemManager.LookupTreeItem("TIXC");
ITcSmTreeItem newProject = cpp.CreateChild("NameOfProject", 1, "", pathToProjectOrTczipFile);
```

Code snippet (PowerShell):

```
$cpp = $systemManager.LookupTreeItem("TIXC")
$newProject = $cpp.CreateChild("NameOfProject", 1, "", $pathToProjectOrTczipFile)
```

Please note that C++ projects cannot be renamed, so the original project name must be specified. (cf. Renaming TwinCAT C++ projects [▶ 216])

Creating module instances

TcCOM Modules could be created at the System -> TcCOM Modules node. Please see documentation there [▶ 333].

The same procedure could also be applied to the C++ project node to add TcCOM instances at that place (`$newProject` at the code on top of this page.).

Calling the TMC Code Generator

The TMC Code Generator can be called to generate the C++ code according to the changes in the TMC file or the C++ project.

Code snippet (C#):

```
string startTmcCodeGenerator = @"<?xml version=""1.0"" encoding=""UTF-16""?>
<TreeItem>
<CppProjectDef>
<Methods>
<StartTmcCodeGenerator>
<Active>true</Active>
</StartTmcCodeGenerator>
</Methods>
</CppProjectDef>
</TreeItem>";
cppProject.ConsumeXml(startTmcCodeGenerator);
```

Code snippet (PowerShell):

```
$startTmcCodeGenerator = @"<?xml version=""1.0"" encoding=""UTF-16""?>
<TreeItem>
<CppProjectDef>
<Methods>
<StartTmcCodeGenerator>
<Active>true</Active>
</StartTmcCodeGenerator>
</Methods>
</CppProjectDef>
</TreeItem>"
$cppProject.ConsumeXml($startTmcCodeGenerator)
```

Calling the Publish Modules command

The publication covers the development of the project for all platforms. The compiled module is made available for export as described in the section Module handling for C++.

Code snippet (C#):

```
string publishModules = @"<?xml version=""1.0"" encoding=""UTF-16""?>
<TreeItem>
<CppProjectDef>
<Methods>
<PublishModules>
<Active>true</Active>
</PublishModules>
</Methods>
</CppProjectDef>
</TreeItem>";
cppProject.ConsumeXml(publishModules);
```

Code snippet (PowerShell):

```
$publishModules = @"<?xml version=""1.0"" encoding=""UTF-16""?>
<TreeItem>
<CppProjectDef>
<Methods>
<PublishModules>
<Active>true</Active>
</PublishModules>
</Methods>
</CppProjectDef>
</TreeItem>"
$cppProject.ConsumeXml($publishModules)
```

Setting C++ project properties

C++ projects offer various options for the build and deployment process.
These can be set via the Automation Interface.

Code snippet (C#):

```
string projProps = @"<?xml version=""1.0"" encoding=""UTF-16""?>
<TreeItem>
<CppProjectDef>
<BootProjectEncryption>Target</BootProjectEncryption>
<TargetArchiveSettings>
<SaveProjectSources>false</SaveProjectSources>
</TargetArchiveSettings>
<FileArchiveSettings>
<SaveProjectSources>false</SaveProjectSources>
</FileArchiveSettings>
</CppProjectDef>
</TreeItem>";
cppProject.ConsumeXml(projProps);
```

Code snippet (PowerShell):

```
$projProps = @"<?xml version=""1.0"" encoding=""UTF-16""?>
<TreeItem>
<CppProjectDef>
<BootProjectEncryption>Target</BootProjectEncryption>
<TargetArchiveSettings>
<SaveProjectSources>false</SaveProjectSources>
</TargetArchiveSettings>
<FileArchiveSettings>
<SaveProjectSources>false</SaveProjectSources>
</FileArchiveSettings>
</CppProjectDef>
</TreeItem>"
$cppProject.ConsumeXml($projProps)
```

The values "None" and "Target" are valid for the BootProjectEncryption.
Both other settings are "false" and "true" values.

Set up project

To build the project or solution, you can use the corresponding classes and methods of the Visual Studio API.

16.4 Creating and handling TcCOM modules

This chapter explains how to add existing TcCOM modules to a TwinCAT configuration and parameterize them. The following topics will be briefly covered in this chapter:

- Acquiring a reference to “TcCOM Objects” node
- Adding existing TcCOM modules
- Iterating through added TcCOM modules
- Setting CreateSymbol flag for parameters
- Setting CreateSymbol flag for Data Areas
- Setting Context (Tasks)
- Linking variables

Acquiring a reference to “TcCOM Objects” node

In a TwinCAT configuration, the “TcCOM Objects” node is located under “SYSTEM^TcCOM Objects”. Therefore you can acquire a reference to this node by using the method ITcSysManager::LookupTreeItem() in the following way:

Code Snippet (C#):

```
ITcSmTreeItem tcComObjects = systemManager.LookupTreeItem("TIRC^TcCOM Objects");
```

Code Snippet (Powershell):

```
$tcComObjects = $systemManager.LookupTreeItem("TIRC^TcCOM Objects")
```

The code above assumes that there is already a systemManager objects present in your AI code.

Adding existing TcCOM modules

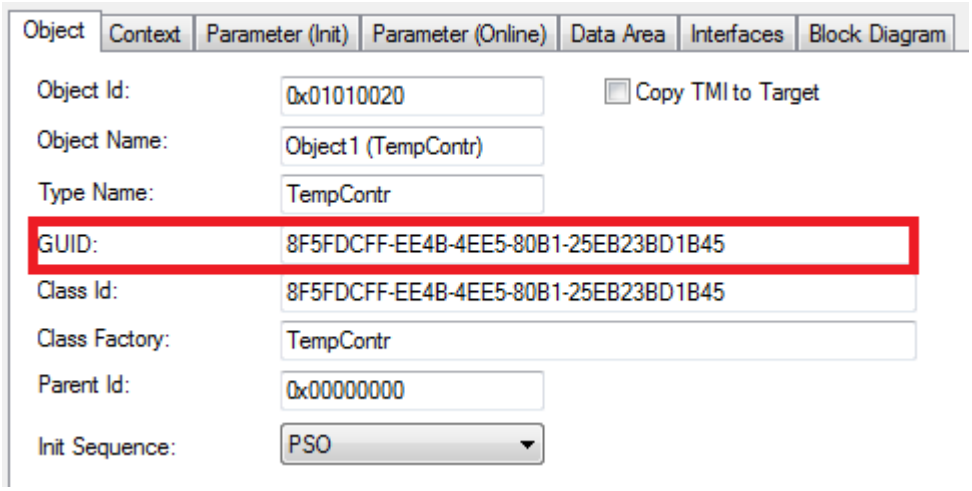
To add existing TcCOM modules to your TwinCAT configuration, these modules need to be detectable by TwinCAT. This can be achieved by either of the following ways:

- Copying TcCOM modules to folder %TWINCAT3.XDIR”\CustomConfig\Modules\
- Editing %TWINCAT3.XDIR”\Config\Io\TcModuleFolders.xml to add a path to a folder of your choice and place the modules within that folder

Both ways will be sufficient to make the TcCOM modules detectable by TwinCAT.

A TcCOM module is being identified by its GUID or name:

- This GUID can be used to add a TcCOM module to a TwinCAT configuration via the ITcSmTreeItem::CreateChild() method. The GUID can be determined in TwinCAT XAE via the properties page of a TcCOM module.



Alternatively, you can also determine the GUID via the TMC file of the TcCOM module.

```
<TcModuleClass>
  <Modules>
    <Module GUID="{8f5fdcff-ee4b-4ee5-80b1-25eb23bd1b45}">
    ...
    </Module>
  </Modules>
</TcModuleClass>
```

Let's assume that we already own a TcCOM module that is registered in and detectable by TwinCAT. We now would like to add this TcCOM module, which has the GUID {8F5FDCFF-EE4B-4EE5-80B1-25EB23BD1B45} to our TwinCAT configuration. This can be done by the following way:

Code Snippet (C#):

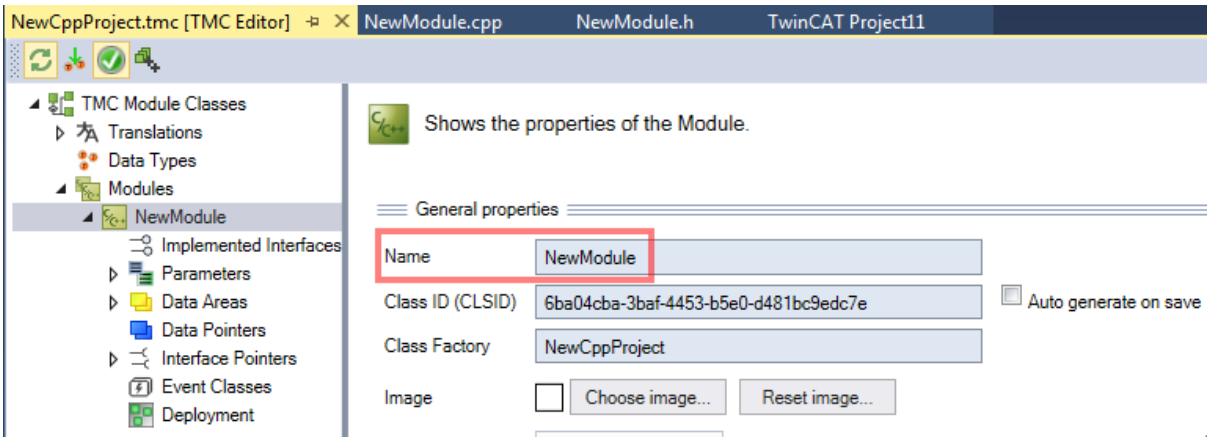
```
Dictionary<string,Guid> tcomModuleTable = new Dictionary<string,Guid>();
tcomModuleTable.Add("TempContr",Guid.Parse("{8f5fdcff-ee4b-4ee5-80b1-25eb23bd1b45}"));
ITcSmTreeItem tempController = tcComObjects.CreateChild("Test", 0, "",
tcomModuleTable["TempContr"]);
```

Code Snippet (Powershell):

```
$tcomModuleTable = @{}
$tcomModuleTable.Add("TempContr", "{8f5fdcff-ee4b-4ee5-80b1-25eb23bd1b45}")
$tempController = $tcComObjects.CreateChild("Test", 0, "", $tcomModuleTable["TempContr"])
```

Please note that the vInfo parameter of the method ItcSmTreeItem::CreateChild() contains the GUID of the TcCOM module which is used to identify the module in the list of all registered TcCOM modules in that system.

- This name can be used to add a TcCOM module to a TwinCAT configuration via the ITcSmTreeItem::CreateChild() method. The name can be determined in TwinCAT XAE via the TMC Editor.



- This can be done by the following way:

Code Snippet (C#):

```
ITcSmTreeItem tempController = tcComObjects.CreateChild("Test", 1, "", "NewModule");
```

Code Snippet (Powershell):

```
$tempController = $tcComObjects.CreateChild("Test", 0, "", "NewModule")
```

Iterating through added TcCOM modules

To iterate through all added TcCOM module instances, you may use the ITcModuleManager2 interface. The following code snippet demonstrates how to use this interface.

Code Snippet (C#):

```
ITcSysManager3 sysManager3 = sysManager as ITcSysManager3;

ITcModuleManager3 moduleManager = sysManager3.GetModuleManager() as ITcModuleManager3;
foreach (ITcModuleInstance2 moduleInstance in moduleManager)
{
  string moduleType = moduleInstance.ModuleTypeName;
  string instanceName = moduleInstance.ModuleInstanceName;
  Guid classId = moduleInstance.ClassID;
```

```
  uint objId = moduleInstance.oid;
  uint parentObjId = moduleInstance.ParentOID;
}
```

Code Snippet (Powershell):

```
$moduleManager = $systemManager.GetModuleManager()
ForEach( $moduleInstance in $moduleManager )
{
  $moduleType = $moduleInstance.ModuleTypeName
  $instanceName = $moduleInstance.ModuleInstanceName
  $classId = $moduleInstance.ClassID
  $objId = $moduleInstance.oid
  $parentObjId = $moduleInstance.ParentOID
}
```

Please note that every module object can also be interpreted as an ITcSmTreeItem, therefore the following type cast would be valid:

Code Snippet (C#):

```
ITcSmTreeItem treeItem = moduleInstance As ITcSmTreeItem;
```

Please note: Powershell uses dynamic data types by default.

Setting CreateSymbol flag for parameters

The CreateSymbol (CS) flag for parameters of a TcCOM module can be set via its XML description. The following code snippet demonstrates how to activate the CS flag for the Parameter “CallBy”.

Code Snippet (C#):

```
bool activateCS = true;
// First step: Read all Parameters of TcCOM module instance
string tempControllerXml = tempController.ProduceXml();
XmlDocument tempControllerDoc = new XmlDocument();
tempControllerDoc.LoadXml(tempControllerXml);
XmlNode sourceParameters = tempControllerDoc.SelectSingleNode("TreeItem/TcModuleInstance/Module/
Parameters");

// Second step: Build target XML (for later ConsumeXml())
XmlDocument targetDoc = new XmlDocument();
XmlElement treeItemElement = targetDoc.CreateElement("TreeItem");
XmlElement moduleInstanceElement = targetDoc.CreateElement("TcModuleInstance");
XmlElement moduleElement = targetDoc.CreateElement("Module");
XmlElement parametersElement = (XmlElement) targetDoc.ImportNode(sourceParameters, true);
moduleElement.AppendChild(parametersElement);
moduleInstanceElement.AppendChild(moduleElement);
treeItemElement.AppendChild(moduleInstanceElement);
targetDoc.AppendChild(treeItemElement);

// Third step: Look for specific parameter (in this case "CallBy") and read its CreateSymbol
attribute
XmlNode destModule = targetDoc.SelectSingleNode("TreeItem/TcModuleInstance/Module");
XmlNode callByParameter = destParameters.SelectSingleNode("Parameters/Parameter[Name='CallBy']");
XmlAttribute createSymbol = callByParameter.Attributes["CreateSymbol"];

createSymbol.Value = "true";

// Fifth step: Write prepared XML to configuration via ConsumeXml()
string targetXml = targetDoc.OuterXml;
tempController.ConsumeXml(targetXml);
```

Code Snippet (Powershell):

```
$tempControllerXml = [Xml]$tempController.ProduceXml()
$sourceParameters = $tempControllerXml.TreeItem.TcModuleInstance.Module.Parameters

[System.XML.XmlDocument] $targetDoc = New-Object System.XML.XmlDocument
[System.XML.XmlElement] $treeItemElement = $targetDoc.CreateElement("TreeItem")
[System.XML.XmlElement] $moduleInstanceElement = $targetDoc.CreateElement("TcModuleInstance")
[System.XML.XmlElement] $moduleElement = $targetDoc.CreateElement("Module")
[System.XML.XmlElement] $parametersElement = $targetDoc.ImportNode($sourceParameters, $true)
$moduleElement.AppendChild($parametersElement)
$moduleInstanceElement.AppendChild($moduleElement)
$treeItemElement.AppendChild($moduleInstanceElement)
$targetDoc.AppendChild($treeItemElement)

$destModule = $targetDoc.TreeItem.TcModuleInstance.Module
```

```
$callByParameter = $destmodule.SelectSingleNode("Parameters/Parameter[Name='CallBy']")

$callByParameter.CreateSymbol = "true"

$targetXml = $targetDoc.OuterXml
$tempController.ConsumeXml($targetXml)
```

Setting CreateSymbol flag for Data Areas

The CreateSymbol (CS) flag for Data Areas of a TcCOM module can be set via its XML description. The following code snippet demonstrates how to activate the CS flag for the Data Area “Input”. Please note that the procedure is pretty much the same as for parameters.

Code Snippet (C#):

```
bool activateCS = true;
// First step: Read all Data Areas of a TcCOM module instance
string tempControllerXml = tempController.ProduceXml();
XmlDocument tempControllerDoc = new XmlDocument();
tempControllerDoc.LoadXml(tempControllerXml);
XmlNode sourceDataAreas = tempControllerDoc.SelectSingleNode("TreeItem/TcModuleInstance/Module/
DataAreas");

// Second step: Build target XML (for later ConsumeXml())
XmlDocument targetDoc = new XmlDocument();
XmlElement treeItem = targetDoc.CreateElement("TreeItem");
XmlElement moduleInstance = targetDoc.CreateElement("TcModuleInstance");
XmlElement module = targetDoc.CreateElement("Module");
XmlElement dataAreas = (XmlElement)
targetDoc.ImportNode(sourceDataAreas, true);
module.AppendChild(dataAreas);
moduleInstance.AppendChild(module);
treeItem.AppendChild(moduleInstance);
targetDoc.AppendChild(treeItem);

// Third step: Look for specific Data Area (in this case "Input") and read its CreateSymbol
attribute
XmlElement dataArea = (XmlElement)targetDoc.SelectSingleNode("TreeItem/TcModuleInstance/Module/
DataAreas/DataArea[ContextId='0' and Name='Input']");
XmlNode dataAreaNo = dataArea.SelectSingleNode("AreaNo");
XmlAttribute createSymbol = dataAreaNoNode.Attributes["CreateSymbols"];

// Fourth step: Set CreateSymbol attribute to true if it exists. If not, create attribute and set
its value
if (createSymbol != null)
string oldValue = createSymbol.Value;
else
{
createSymbol = targetDoc.CreateAttribute("CreateSymbols");
dataAreaNo.Attributes.Append(createSymbol);
}
createSymbol.Value = XmlConvert.ToString(activateCS);

// Fifth step: Write prepared XML to configuration via ConsumeXml()
string targetXml = targetDoc.OuterXml;
tempController.ConsumeXml(targetXml);
```

Code Snippet (Powershell):

```
$tempControllerXml = [Xml]$tempController.ProduceXml()
$sourceDataAreas = $tempControllerXml.TreeItem.TcModuleInstance.Module.DataAreas

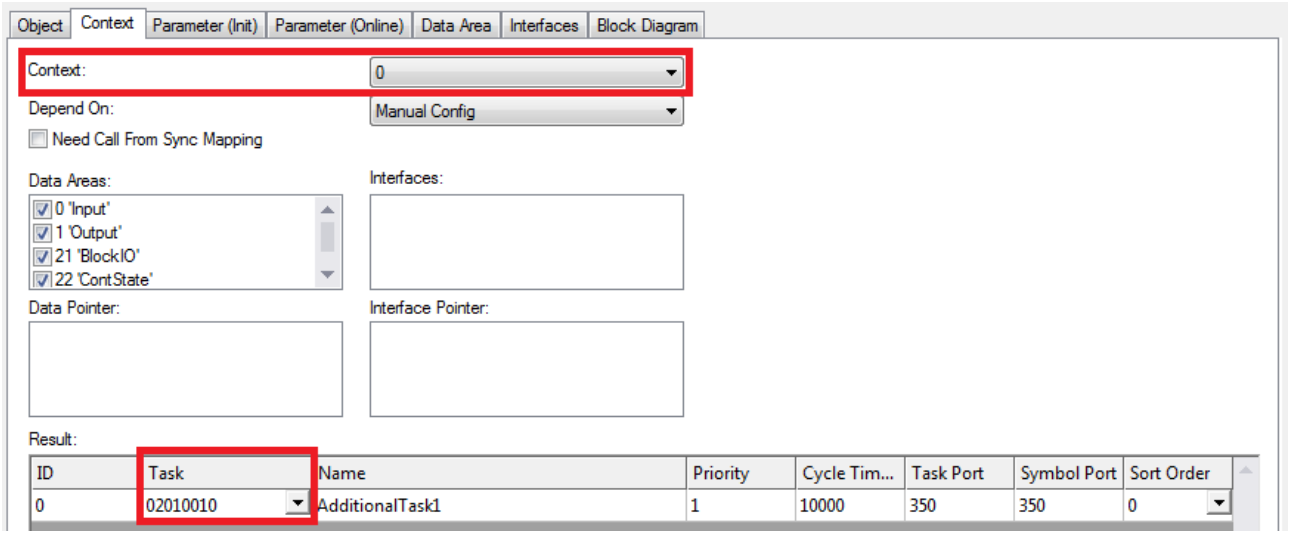
[System.XML.XmlDocument] $targetDoc = New-Object System.XML.XmlDocument
[System.XML.XmlElement] $treeItem = $targetDoc.CreateElement("TreeItem")
[System.XML.XmlElement] $moduleInstance = $targetDoc.CreateElement("TcModuleInstance")
[System.XML.XmlElement] $module = $targetDoc.CreateElement("Module")
[System.XML.XmlElement] $dataAreas = $targetDoc.ImportNode($sourceDataAreas, $true)
$module.AppendChild($dataAreas)
$moduleInstance.AppendChild($module)
$treeItem.AppendChild($moduleInstance)
$targetDoc.AppendChild($treeItem)

$destModule = $targetDoc.TreeItem.TcModuleInstance.Module
[System.XML.XmlElement] $dataArea = $destModule.SelectSingleNode("DataAreas/DataArea[ContextId='0'
and Name='Input']")
$dataAreaNo = $dataArea.SelectSingleNode("AreaNo")
$dataAreaNo.CreateSymbols = "true"
```

```
// Fifth step: Write prepared XML to configuration via ConsumeXml()
$targetXml = $targetDoc.OuterXml
$tempController.ConsumeXml($targetXml)
```

Setting Context (Tasks)

Every TcCOM module instance needs to be run in a specific context (task). This can be done via the ITcModuleInstance2::SetModuleContext() method. This method awaits two parameters: ContextId and TaskObjectId. Both are equivalent to the corresponding parameters in TwinCAT XAE:



Please note that the TaskObjectId is shown in hex in TwinCAT XAE.

Code Snippet (C#):

```
ITcModuleInstance2 tempControllerMi = (ITcModuleInstance2) tempController;
tempControllerMi.SetModuleContext(0, 33619984);
```

You can determine the TaskObjectId via the XML description of the corresponding task, for example:

Code Snippet (C#):

```
ITcSmTreeItem someTask = systemManager.LookupTreeItem("TIRT^SomeTask");
string someTaskXml = someTask.ProduceXml();
XmlDocument someTaskDoc = new XmlDocument();
someTaskDoc.LoadXml(someTaskXml);
XmlNode taskObjectIdNode = someTaskDoc.SelectSingleNode("TreeItem/ObjectId");
string taskObjectIdStr = taskObjectId.InnerText;
uint taskObjectId = uint.Parse(taskObjectIdStr, NumberStyles.HexNumber);
```

Linking variables

Linking variables of a TcCOM module instance to PLC/IO or other TcCOM modules can be done by using regular Automation Interface mechanisms, e.g. ITcSysManager::LinkVariables().

16.5 Third-party components

This software contains third-party components.
Please refer to the license file provided in the following folder for further information:
C:\Program Files(x86)\Beckhoff\LegalTwinCAT-XAE-CppPlatformWizard

C:\Program Files(x86)\Beckhoff\LegalTwinCAT-XAE-PublicSDK

Trademark statements

Beckhoff®, ATRO®, EtherCAT®, EtherCAT G®, EtherCAT G10®, EtherCAT P®, MX-System®, Safety over EtherCAT®, TC/BSD®, TwinCAT®, TwinCAT/BSD®, TwinSAFE®, XFC®, XPlanar® and XTS® are registered and licensed trademarks of Beckhoff Automation GmbH.

Third-party trademark statements

CANopen and CANopen FD are registered trademarks of CAN in AUTOMATION - International Users and Manufacturers Group e.V.

DALI, DALI-2, D4i, DALI+, and DiiA are trademarks in various countries in the exclusive use of the Digital Illumination Interface Alliance.

DSP System Toolbox, Embedded Coder, MATLAB, MATLAB Coder, MATLAB Compiler, MathWorks, Predictive Maintenance Toolbox, Simscape, Simscape™ Multibody™, Simulink, Simulink Coder, Stateflow and ThingSpeak are registered trademarks of The MathWorks, Inc.

Excel, IntelliSense, Microsoft, Microsoft Azure, Microsoft Edge, PowerShell, Visual Studio, Windows and Xbox are trademarks of the Microsoft group of companies.

MAX®, Stratix®, Cyclone®, Altera®, Agilex™, Arria®, Intel, the Intel logo, Intel Core, Xeon, Intel Atom, Celeron and Pentium are trademarks of Intel Corporation or its subsidiaries.

The registered trademark Linux® is used pursuant to a sublicense from the Linux Foundation, the exclusive licensee of Linus Torvalds, owner of the mark on a worldwide basis.

OPC UA is a registered trademark of the OPC Foundation.

More Information:
www.beckhoff.com/te1000

Beckhoff Automation GmbH & Co. KG
Hülshorstweg 20
33415 Verl
Germany
Phone: +49 5246 9630
info@beckhoff.com
www.beckhoff.com